Vesa Vilkman

JATKUVA INTEGRAATIO OHJELMISTOKEHITYKSESSÄ

Tietojärjestelmätieteen kandidaatintutkielma

1. kesäkuuta 2010

JYVÄSKYLÄN YLIOPISTO
TIETOJENKÄSITTELYTIETEIDEN LAITOS

TIIVISTELMÄ

Vilkman, Vesa Ilmari

Jatkuva integraatio ohjelmistokehityksessä / Vesa Vilkman

Jyväskylä: Jyväskylän yliopisto, 2010, 36 s.

Kandidaatintutkielma

Kandidaatintutkielman tarkoituksena on esitellä lukijalle jatkuva integraatio-käytänne ja sen käyttöönottoa harkittaessa huomioonotettavat seikat. Tutkielmassa käsitellään jatkuva integraatio käytänteen ominaispiirteitä, samalla verraten sitä muihin integraatiostrategioihin. Tutkielman päämääränä on vastata tutkimus-ongelmaan: ”Millaisia vaikutuksia jatkuva integraatio-periaatteen käyttöönotolla on ohjelmistokehitysprosessiin?”

Tutkielma on tehty tutustumalla aiempiin tutkimuksiin ja kirjallisuuteen kirjallisuuskatsauksen muodossa. Aluksi tutkielmassa esitellään erilaiset lähestymistavat integraatioprosessiin, jonka jälkeen esitellään jatkuvan integraa-tion keskeiset käytänteet. Tämän jälkeen esitellään jatkuvan integraation käyttöön-oton tuomia etuja ja haasteita.

Lopuksi yhteenvedossa kerrataan tutkielman sisältö ja arvioidaan jatkuvan integraation käyttöönoton tuomia etuja ja käyttöönottoprosessin haasteita. Johto-päätöksenä tuodaan esille, että jatkuvan integraation tuomat edut puhuvat puoles-taan, mutta ongelmatekijöiden minimoimista ei tule unohtaa.

Avainsanat: jatkuva integraatio, integraatio, koostamisprosessi, ohjelmistokehitys

SISÄLTÖ

1 JOHDANTO

Nykyaikaiset ohjelmistotuotteet koostuvat yleensä sadoista, jopa tuhansista erillisistä komponenteista (Kim, Park, Yun & Lee, 2009). Komponentilla tarkoitetaan tämän tutkielman yhteydessä Binderin (1999, 1080) määritelmän mukaisesti mitä tahansa ohjelmiston osakokonaisuutta, joka on nähtävissä kehitysympäristössä (*development environment*), kuten esimerkiksi metodeja, luokkia, olioita, funktioita, moduuleja tai alijärjestelmiä. Ohjelmistotuotteen sisältämien komponenttien kehitysprosessit ovat usein toisistaan riippumattomia ajan, paikan ja henkilöstön suhteen. Komponentit voivat olla esimerkiksi yhtiön sisällä kehitettyjä tai avoimen lähdekoodin (*open source*) komponentteja. Ohjelmistotuotteen komponenttien moninaisuus asettaa monia haasteita ohjelmistokehitysprosessille. Toiminnallisuuden muutokset komponenteissa voivat aiheuttaa ennalta arvaamattomia ongelmia, joten päivitetyn tai kokonaan uuden komponentin integraatio toimivaan ohjelmistotuotteeseen on tehtävä tarkasti kontrolloidusti. (Kim ym., 2009.)

Ohjelmistokehitysprosessin suurimpia riskejä ovat ongelmat, jotka syntyvät kun toisistaan erillään kehitetyt komponentit pyritään yhdistämään, eikä yhdistetty ohjelmisto toimi halutulla tavalla. Myöhäisessä vaiheessa ilmitulleiden virheiden johdosta ohjelmiston toimintaa voidaan joutua muuttamaan laajalti, jotta korjaukset ovat mahdollisia. (McConnell, 1996b, 406.) Suurten ja monimutkaisten ohjelmistojen tapauksessa, joissa kriittisten osien kehitys ja testaus suoritetaan erillään muista osista, integraatiovaihe voi muodostua kohtalokkaaksi (Larsson, Crnkovic & Ekdahl, 2004). Äärimmäisissä tapauksissa integraatio-ongelmat ovat aiheuttaneet jopa projektien peruuttamisia. (McConnell, 1996b, 406.)

Mitä myöhemmäksi integraatioprosessia lykätään, sitä todennäköisemmin sen aikana tulee ongelmia. Myös virheiden selvitystyö vaikeutuu muutosjoukon kasvaessa. Olisikin tärkeää, että komponenttien integraatio suoritetaan mahdollisimman varhain (van der Storm, 2007). Perinteisissä vaihejakoisissa prosessimalleissa integraatio ja testaus sijoittuvat ohjelmistokehitysprosessin loppuvaiheeseen (Haikala & Märijärvi, 2001. 37). Myöhäisessä vaiheessa suoritettava integraatio on usein pitkä ja ennustamaton prosessi (Fowler, 2006). Usein ohjelmiston varsinainen koostamis- ja integraatioprosessi toteutetaan vasta hieman ennen projektin virstanpylväitä, jolloin esiin tulevat ongelmat ovat monimutkaisia ja niiden ratkaisu vie paljon resursseja (Duvall, Matyas & Glover, 2007, 66).

Ohjelmiston koostamisella tarkoitetaan koostamisprosessin (*build process*) suorittamista, jonka tarkoituksena on tuottaa suorituskelpoinen ohjelmisto eli ohjelmistokooste (*software build*). Koosteprosessiin kuuluvia toimia ovat lähdekoodi- ja binääritiedostojen valitseminen, kääntäminen ja linkittäminen sekä komponentin sisällä, että useiden komponenttien välillä. (Binder, 1999. 629.)

Stavridoun (1999) määritelmän mukaan integraatioprosessi on menettely, jossa kootaan yhteen ohjelmiston osajärjestelmät, tarkoituksena luoda yksi, eheä ohjelmisto, joka täyttää organisaation sille asettamat vaatimukset. Binderin (1999, 629) mukaan Stavridoun määritelmässä kyse on vain ohjelmiston koostamisesta, integraation ollessa laajempi prosessi, johon kuuluu myös komponenttien välisen toiminnan varmistaminen. Tässä tutkielmassa noudatetaan em. Binderin määritelmää, jonka mukaisesti integraatiolla tarkoitetaan ohjelmiston koostamista ja testaamista.

Integraatioprosessin helpottamiseksi Kent Beck (2000, 97–99) esitteli Extreme Programming -menetelmän yhteydessä nk. jatkuva integraatio (*continuous integration*) -käytänteen. Jatkuvan integraation tarkoituksena on helpottaa ohjelmistokehittäjän työtä ja parantaa ohjelmistotuotteen laatua. Käytännössä jatkuvan integraation mukaisesti toimiva kehitystiimi yhdistää työnsä usein, vähintään päivittäin. Jokaisen integraatiokerran jälkeen ohjelmistokooste luodaan ja testataan automaattisesti. Tavoitteena on koostaa koko ohjelmisto heti muutoksen lisäyksen yhteydessä, jotta virheet löydettäisiin mahdollisimman ajoissa. Tämä vähentää mm. kehitysprojektin loppuvaiheessa ilmeneviä integraatio-ongelmia ja niiden selvittelyyn kuluvaa aikaa. (Fowler, 2006.)

Tässä kandidaatintutkielmassa esitellään jatkuva integraatio -käytänne sekä syvennytään sen ohjelmistokehitysprosessiin tuomiin etuihin ja haasteisiin. Tutkimusongelma on muotoiltu seuraavasti: ”Millaisia vaikutuksia jatkuva integraatio -periaatteen käyttöönotolla on ohjelmistokehitysprosessiin?”

Jäsentelyn helpottamiseksi on tutkimusongelma jaettu seuraaviin tutkimuskysymyksiin:

- Mitä tarkoitetaan jatkuva integraatio -periaatteella ja mistä osista se koostuu?
- Millaisia etuja jatkuvan integraation käyttöönotto tarjoaa ohjelmistokehitysprosessiin?
- Mitkä haasteet ja riskit tulee ottaa huomioon jatkuvan integraation käyttöönottoa harkittaessa?

Tutkielman tavoitteena on luoda jäsennelty, kattava kuvaus jatkuva integraatio -periaatteesta ja sen käyttöönoton vaikutuksista. Tutkielman tuloksia voidaan

käyttää esimerkiksi harkittaessa jatkuvan integraation käyttöönottoa ohjelmisto-tuotantoprojektissa. Tutkimus toteutetaan kirjallisuuskatsauksena.

2 AIEMMAT LÄHESTYMISTAVAT INTEGRAATIOON

Erilaiset integraatiostrategiat voidaan jakaa niiden suoritusajankohdan perusteella kahteen ryhmään: erillisessä integraatiovaiheessa tapahtuvaan integraatioon ja kehityksen yhteydessä tapahtuvaan esimerkiksi päivittäin suoritettavaan integraatioon. Tämän luvun aliluvuissa esitellään nämä kaksi tapaa ja tärkeimmät niihin liittyvät edut ja haitat.

2.1 Integraatiovaihe

Vaihejakoisissa kehitysmalleissa ohjelmistokehitysprosessi jaetaan erillisiin vaiheisiin kuten määrittely, suunnittelu, toteutus sekä integraatio ja testaus. Integraatiovaihe on tällöin eritelty omaksi vaiheekseen, jonka suoritus alkaa vasta toteutusvaiheen päätyttyä. (Cusumano & Selby, 1997.)

Pressmanin (2005, 366) mukaan integraatiovaiheesta pyritään liian usein suoriutumaan kertarysäyksellä (*big bang*), eli koostamalla ohjelmiston kaikki osat kerralla. Luotua ohjelmistokoostetta pyritään testaamaan kokonaisuutena, jolloin virheiden selvitystä vaikeuttaa se, ettei niiden syitä voida eritellä selvästi ja korjausyritykset yleensä synnyttävät vain lisää virheitä. (Pressman, 2005, 366.)

Inkrementaaliset integraatiostrategiat ovat kertarysäys-strategiaa paremmaksi havaittu ratkaisu. Ne mahdollistavat systemaattisen lähestymistavan testaukseen, sillä integraatio ja testaus toteutetaan vähitellen pienissä osissa. Inkrementaalisia strategioita ovat esimerkiksi osittava (*top-down*) tai kokoava (*bottom-up*). Kokoavasti integroitaessa edetään kutsuhierarkiassa alimmalta tasolta ylöspäin, kun osittaen integroitaessa suunta on päinvastainen. Inkrementaalisten integraatiostrategioiden mukaan toimimalla ohjelmisto testataan osissa, jolloin osa-alueiden

virheet voidaan eristää ja ne ovat helpommin löydettävissä. (Pressman, 2005. 366–368.)

2.2 Päivittäinen integraatio

Päivittäisen koostamisen ja savutestauksen (*daily build & smoke test*) tapauksessa luodaan päivittäin ohjelmistokooste, jonka toimivuus pyritään testaamaan pääpiirteittäin. Päivittäinen koostaminen ja savutestaus suoritetaan luomalla keskitetty ohjelmistokooste erillisessä koostamisympäristössä. Päivittäin suoritettaviin toimiin kuuluvat ohjelmiston lähdekoodien kääntäminen ja osien linkittäminen sekä savutestin suorittaminen. Koostamisprosessia pidetään onnistuneena, jos kaikki vaiheet suoritetaan onnistuneesti. Onnistuneen ohjelmistokoosteen tuottamisen päivittäin tulisi olla projektin pääprioriteetti. (McConnell, 1996b. 407–414.)

Jotta päivittäisen koostamisen ja savutestauksen suorittaminen olisi mielekästä, on koostamisprosessin oltava nopeasti ja helposti toistettavissa. (Karlsson, Andersson & Leion, 2000.) Testien suorittaminen päivittäin vaatii niiden automatisoimista, muutoin tarvittavan työn määrä kasvaa liian suureksi. (Koroorian & Kajko-Mattsson, 2008.) Päivittäisen koostamisen ja savutestien avulla ohjelmiston tilasta ja projektin etenemisestä saadaan tietoa tasaiseen tahtiin. (Pressman, 2005, 369.) Savutestien tarkoituksena on etsiä sellaiset virheet, jotka estävät ohjelmiston suorittamisen. Savutestien läpäisy osoittaa ohjelmiston olevan valmis tarkempaa testausta varten. (McConnell, 1996b. 408.)

Koostamisprosessin ja savutestien tulee kehittyä kehitettävän ohjelmiston mukana. (McConnell, 1996b. 414.) Koostamisprosessin ja testien ylläpitoon tulee nimittää vastuuhenkilö, jonka vastuulla on koostamisympäristön ja -skriptien ylläpito,

koostamisprosessin suorittaminen ja kehittäjien informoiminen savutestin löytämistä virheistä. (Koroorian & Kajko-Mattsson, 2008.) Laajemmissa projekteissa koostamisprosessin ja testien ylläpito voi vaatia useamman kuin yhden ihmisen täyden työpanoksen. (McConnell, 1996b. 408.)

2.3 Lähestymistapojen edut ja ongelmat

Erilliseen integraatiovaiheeseen siirretyn integraation ongelmana on vaiheen keston ennustaminen ja etenemisen seuranta. (Fowler, 2006.) Berczukin ja Appletonin (2002, 55) mukaan kertarysäyksellä tapahtuvan integraation kulut ylittävät ennakkoarviot lähes poikkeuksetta.

Integraatiovaiheessa ilmitulleet ongelmat johtuivat Herbslebin ja Grinterin (1999) tutkimuksen mukaan puutteellisesta suunnitteludokumentaatiosta. Koska dokumentaatio ei ollut yksiselitteistä, kehittäjät tekivät erilaisia olettamuksia halutusta toiminnasta. Tästä aiheutui integraatio-ongelmia, sillä valmiit osat eivät olleet yhteensopivia keskenään, vaikka täyttivätkin niille asetetut vaatimukset. Ongelmia aiheutti mm. se, että osat testattiin kehitysvaiheessa erillään toisistaan. (Herbsleb & Grinter, 1999.)

Virheiden selvitys integraatiovaiheessa on poikkeuksetta monimutkaisempaa kuin tapauksessa jossa ne olisi huomattu aiemmin. Epäonnistuneen integraatiovaiheen johdosta jopa lähes valmiina pidettyjä projekteja on peruutettu. (McConnell, 1996b. 405–406.)

Päivittäin tapahtuvan koostamisen ja savutestauksen avulla voidaan integraatio-ongelmien laajuutta rajoittaa ja pitää ne helpommin hallittavina. Ohjelmakoodiin tehdyt muutokset yhdistetään päähaaraan päivittäin, jolloin voidaan vähentää

rinnakkain tehtyjen, päällekkäisten muutosten yhdistämisestä aiheutuvia nk. yhdistämiskonflikteja (*merge conflicts*). (Karlsson ym., 2000.) Suurta muutosjoukkoa integroitaessa on virheen aiheuttaneen muutoksen löytäminen hankalaa. Päivittäin koostettaessa ja testattaessa voidaan osoittaa vuorokauden tarkkuudella, milloin virheen aiheuttanut muutos on tehty. (McConnell, 1996b. 406.)

Päivittäisen koostamisen ja savutestien avulla voidaan havaita suuri osa integraatio-ongelmista jo kehitysvaiheen aikana. Näin saadaan selville ongelmien lähteet välittömästi ja siten varmistettua että projekti etenee haluttuun suuntaan. (Koroorian & Kajko-Mattsson, 2008.) Integraatiovaiheen täydellinen epäonnistuminen voidaan siis torjua, samalla helpottaen virheiden selvitys- ja korjaustyötä sekä parantaen ohjelmakoodin laatua. (McConnell, 1996b. 405. ks. Pressman, 2005. 370.) Päivittäistä ohjelmistokoostetta ja savutestiä on onnistuneesti käytetty laajoissakin projekteissa: Microsoft on käyttänyt menetelmää useissa projekteissa, mm. käyttöjärjestelmien ja toimisto-ohjelmistopakettien kehitystyössä. Windows 95:n tapauksessa kehitystiimin koko oli yli 200 henkeä ja koodirivejä oli yli 11 miljoonaa (Cusumano & Selby, 1997).

Koroorian ja Kajko-Mattsson (2008) huomioivat päivittäisen koostamisen vaikutuksen motivaatioon: kehittäjä motivoituu tekemään laadultaan parempaa lähdekoodia, sillä työstä saadaan päivittäin palautetta koostamisprosessin tuloksen muodossa. Tämän palautteen puute johtaa työmoraalin ja vastuuntunnon laskemiseen (Koroorian & Kajko-Mattsson, 2008). Karlsson ym. (2000) huomioivat, että keskityttäessä liikaa toimivien ohjelmistokoosteiden tuottamiseen koodin laatu ja ohjelmiston arkkitehtuuri itse asiassa kärsivät. Virhetilanteet selvitetään vippaskonstein ja myöhemmissä vaiheissa joudutaan selvittämään entistä vaikeampia ongelmia. (Karlsson ym., 2000.)

Päivittäinen koostaminen vaatii organisaatiokulttuurin ja kehitysprosessin sopeutumista uuteen toimintatapaan, sillä sen esivaatimuksena on koostamiskelpoinen ohjelmisto jo kehitysvaiheen aikana. Tämä saattaa vaatia suuriakin muutoksia vaihejakoista kehitysprosessia noudattavaan organisaatioon, jossa on totuttu tuottamaan ohjelmistokoosteita esimerkiksi kuukausittain tai vasta projektin päätteeksi. Liian nopeatempoinen ja huonosti valmisteltu siirtyminen päivittäisen koostamisen käyttöön onkin omiaan aiheuttamaan ongelmia. (Koroorian & Kajko-Mattsson, 2008.)

3 JATKUVA INTEGRAATIO

Ohjelmiston testauksen ja usein tapahtuvan koostamisen arvo ymmärretään alan harjoittajien keskuudessa, mutta prosessien työläyden takia ne usein jätetään vähemmälle huomiolle. Näiden tehtävien automatisointia vältellään sillä perusteella, että ohjelmiston kehitysprosessi on luonteeltaan monimutkainen. Yleensä ei huomioida sitä, että kehitysprosessi pitää sisällään sellaisia tehtäviä joiden automatisoiminen on kannattavaa - jopa välttämätöntä. (Duvall ym., 2007. 65–66).

Jatkuva integraatio on joukko ohjelmistotuotannon alalla hyväksi havaittuja käytänteitä, joiden avulla voidaan vähentää integraatioon kuluvaa aikaa ja siten nopeuttaa ohjelmiston toimitusaikoja. (Stolberg, 2009.) Jatkuvan integraation päämääränä on pyrkiä vähentämään integraatio-ongelmia ja niiden selvitystyöhön kuluvaa aikaa (Fowler, 2006.).

Jatkuva integraatio voidaan nähdä edistyneempänä versiona aiemmin esitellystä päivittäisestä koostamisesta ja savutestauksesta. Jatkuvassa integraatiossa painotetaan selvästi enemmän tehtävien automatisointia, testausta ja ohjelmistokoosteiden luomista jokaisen muutospäivityksen yhteydessä. (vrt. Fowler, 2006 ja McConnell, 1996a.) Duvallin ym. (2007. 13) visio hyvin toimivasta jatkuvasta integraatiosta on näppäimen painalluksella käynnistyvä aktiviteetti jossa kehittäjän tekemät muutokset yhdistetään päähaaraan, ohjelmisto koostetaan, testataan ja otetaan käyttöön.

Seuraavaksi käydään läpi jatkuvan integraation keskeiset käytänteet: tiedostojen keskittäminen, päivittäinen muutosten päivittäminen, ohjelmistokooste, testaus, käyttöönotto ja tiedon jakaminen.

3.1 Tiedostojen keskittäminen

Ohjelmistoprojektit käsittävät suuren määrän erilaisia tiedostoja: lähdekoodia, kolmannen osapuolen tuottamia kirjastoja, asetustiedostoja ja niin edelleen. Näistä tiedostoista organisoidusti yhteen kerättynä lopulta muodostuu ohjelmistotuote. Tiedostojen hallinta vaatii suuria ponnisteluja, erityisesti kun kehitystyössä on mukana useita henkilöitä. (Fowler, 2006.)

Tiedostojen hallintaa helpottamaan Berczuk & Appleton (2002. 79–86.) esittelevät Tietovarasto-konfiguraationhallintamallin (*Repository-pattern*), jonka mukaisesti kaikki ohjelmistoon liittyvät tiedostot keskitetään versionhallintajärjestelmään. Tietovarasto-mallissa määritellään tarvittaviksi tiedostoiksi muokattavan lähdekoodin lisäksi mm. muut tarvittavat komponentit, kolmannen osapuolen tuottamat komponentit, konfiguraatiotiedostot, sovelluksen alustamiseksi tarvittavat tiedostot, ohjelmistokoosteen luomiseen tarvittavat tiedostot ja testitapaukset. Tietovarasto-konfiguraationhallintamallin käyttöönotto mahdollistaa nopeasti tapahtuvan työtilojen (*workspace*) monistamisen sekä helpottaa ohjelmistokoosteen luomista. (Berczuk & Appleton 2002. 79–86.)

Fowlerin (2006) mukaan ohjelmistokehityksen tulisi pohjautua nk. päähaara-malliin (*mainline pattern*), jonka mukaisesti toimimalla voidaan välttää haarautumisen (*branching*) ja yhdistämisen (*merging*) aiheuttamia ongelmia. Päähaara-mallin mukaisessa ohjelmistokehityksessä pyritään lähdekoodista ylläpitämään vain yhtä päähaaraa, johon kaikki tarvittavat muutokset yhdistetään mahdollisimman aikaisin (Berczuk & Appleton, 2002. 54).

Tiedostojen keskittämisellä versionhallintajärjestelmään voidaan vähentää myös sellaisia ongelmatilanteita, joissa huomataan että tehdyt muutokset toimivat vain

yhdessä ympäristössä (Duvall ym., 2007, 74). Ihannetilanteessa kuka tahansa voisi tuoda nk. neitsytkoneen (*virgin machine)*, jolle ei ole asennettu kuin minimaalinen määrä tarvittavia työkaluja, ja ladata siihen versionhallinnasta kaikki kehitettävään ohjelmaan liittyvät tiedostot (Fowler & Foemmel, 2001).

3.2 Päivittäinen muutosten päivitys

Nykyaikaisen, yleensä rinnakkain tapahtuvan ohjelmistokehityksen ongelmaksi muodostuu eri tahoilla tehtyjen muutosten yhdistäminen. Perryn, Siyn & Vottan. (2001) tutkimuksen mukaan jopa 45 prosentista tiedostoja oli 2-16 erilaista, rinnakkaista versiota. Vaikka versionhallintajärjestelmät tukevatkin rinnakkaisten versioiden yhdistämistä, joudutaan päällekkäisten muutosten tapauksissa tekemään työ manuaalisesti. Yhdistämällä muutokset usein voidaan välttyä suurelta määrältä manuaalista työtä, joka muuten kuluisi syntaksivirheiden korjaamiseen. Tästä syystä varsinkin laajamittaisten projektien yhteydessä muutosten koordinointi on erityisen tärkeää. (Perry ym. 2001.)

Jatkuvan integraation tapauksessa muutospäivitysprosessi on nelivaiheinen (Fowler, 2006.):

1. Oman tehtävänsä suoritettuaan kehittäjä noutaa muiden tekemät muutokset versionhallintajärjestelmästä.
2. Kehittäjä luo yksityisen ohjelmistokoosteen omassa kehitysympäristössään.
3. Yksityisen koostamisprosessin onnistuessa kehittäjä lisää omat muutoksensa versionhallintajärjestelmään.
4. Suoritetaan keskitetty koostamisprosessi erillisellä integraatiokoneella.

Fowlerin (2006) mukaan on tärkeää, että jokainen siirtää tekemänsä muutokset versionhallintaan vähintään päivittäin. Mitä useammin muutokset päivitetään versionhallintajärjestelmään, sitä vähemmän ongelmia niiden yhdistäminen aiheuttaa. (Larsson, Myllyperkiö & Ekdahl. 2007). Koroorian ja Kajko-Mattsson (2008) huomioivat tutkimuksessaan, että vaatimusta päivittäin tapahtuvasta muutosten päivityksestä versionhallintaan noudatettiin liiankin kirjaimellisesti: kehittäjät päivittivät versionhallintaan myös keskeneräiset muutokset, ja siksi koostamisprosessi epäonnistui. Tämänkaltaisen tilanteen torjumiseksi Duvall ym. (2007. 40–41) ohjeistavat jakamaan tehtävät pienempiin osiin ja päivittämään muutokset jokaisen osatehtävän valmistuttua.

3.3 Ohjelmistokooste

Yhdistämällä tehdyt muutokset päivittäin voidaan torjua rinnakkain kehitetyn lähdekoodin yhdistämisestä syntyneiden syntaksivirheiden korjaukseen kuluvaa työtä. Virheet jotka piilevät ohjelmiston osien varsinaisessa toiminnallisuudessa eivät kuitenkaan tule esille, ellei ohjelmistoa koosteta ja testata (Perry ym., 2001).

Ohjelmiston luominen lähdekoodeista ja muista komponenteista käyttökelpoiseksi ohjelmistotuotteeksi on monimutkainen prosessi, johon liittyy lähdekoodien kääntämistä, tiedostojen siirtelyä sekä kantakaavojen lataamista tietokantoihin. Erilaisten komentojen syöttäminen ja valintaikkunoiden läpikäyminen on virhealtista ja yksitoikkoista työtä. (Fowler, 2006). Tällaisten tehtävien automatisoiminen skriptejä käyttäen ei ainoastaan nopeuta työskentelyä, vaan poistaa inhimillisten virheiden, kuten esimerkiksi virhepainallusten mahdollisuuden. (Duvall ym., 2007. 52, 67.)

Ohjelmistokoosteen luominen on usein mahdollista tehdä suoraan ohjelmointiympäristössä (*development environment*) pelkällä napin painalluksella, mutta Fowler (2006) painottaa alustariippumattoman koosteskriptin tärkeyttä. Alustariippumattoman koosteskriptin avulla ohjelmistokoosteen luominen tapahtuu aina samalla tavalla, riippumatta koostamisprosessin suorittajasta, kehitystyökalun versiosta tai koostamisprosessin suoritusympäristön parametreista ja mahdollistaa ohjelmistokoosteen luomisen puhtaalle koneelle. (Duvall ym., 2007. 68–69.)

Fowlerin (2006) mukaan yksityisen koostamisprosessin suorittamisen lisäksi tulee jokaisen muutoksen jälkeen muodostaa päähaarasta keskitetty ohjelmistokooste erillisellä integraatiokoneella, jotta voidaan katsoa kehittäjän suorittaneen muutostenpäivitysprosessin hyväksytysti. Keskitetyn ohjelmistokoosteen luominen harvemmin kuin jokaisen muutospäivityksen yhteydessä ei palvele jatkuvan integraation periaatetta, jonka mukaan virheet pyritään löytämään ja poistamaan järjestelmästä mahdollisimman nopeasti (Fowler, 2006).

Keskitetty koostamisprosessi voidaan suorittaa integraatiokoneella joko manuaalisesti, tai automaattisesti käyttäen erillistä integraatiopalvelinohjelmistoa (*continuous integration server*). Manuaalisesti tehtynä ohjelmistokoosteen luominen on pitkälti samankaltainen prosessi kuin muutostenpäivitysprosessi, joka kuvattiin aiemmassa aliluvussa: kehittäjä lataa versionhallinnasta uusimman päähaaran integraatiokoneelle ja käynnistää integraatiokoneella keskitetyn koostamisprosessin, jonka onnistuessa voidaan muutostenpäivitysprosessia pitää onnistuneena. (Fowler, 2006.) Automaattista keskitetyn ohjelmistokoosteen luomista varten on saatavilla lukuisia valmiita integraatiopalvelinohjelmistoja. Integraatiopalvelin tarkkailee versionhallinnan tapahtumia ja muutosten ilmetessä käynnistää koostamisprosessin automaattisesti. Koostamisprosessin lopputulos voidaan

ilmoittaa kehittäjille esimerkiksi sähköpostitse, tekstiviestillä, pikaviestimellä tai vaikkapa vaihtamalla yhteisessä työtilassa olevan valaisimen väriä. (Duvall ym., 2007, 85, 210–221.)

Suorittamalla ohjelmiston koostamisprosessi jokaisen muutoksen yhteydessä syntyy luottamus siihen, että kehitystyötä tehdään vakaalla pohjalla, sillä jokaisen koosteprosessin lopuksi saadaan palaute siitä, onnistuiko kooste ja läpäistiinkö vaaditut testit. Mainitun luottamuksen muodostumisen edellytyksenä on, että sitoudutaan korjaamaan koostamisprosessin esiintuomat virheet mahdollisimman nopeasti. (Fowler, 2006.)

Jotta ohjelmistokoosteen muodostaminen jokaisen muutospäivityksen yhteydessä olisi mielekästä, tulee koostamisprosessin suoritusajan olla lyhyt (Fowler, 2006). Pitkäkestoinen koostamisprosessi vähentää jatkuvan integraation hyötyjä, sillä kehittäjän tulee jokaisen muutospäivityksen yhteydessä odottaa keskitetyn koostamisprosessin lopputulos ennen muihin tehtäviin siirtymistä (Shore & Warden, 2007, 177). Koostamisprosessin ollessa liian hidas kehittäjät reagoivat päivittämällä muutokset versionhallintaan harvemmin. (Duvall ym., 2007, 87.) Koostamisprosessin maksimisuoritusaika on aina tapauskohtainen, mutta esimerkiksi Shore ja Warden (2007, 186) esittävät nyrkkisääntönä ettei ohjelmistokoosteen luomisen ja testauksen tulisi ylittää 10 minuuttia.

Sekä Fowler (2006) että Duvall ym. (2007, 92–96) tuovat esiin mahdollisuuden vaiheistaa ohjelmistokoosteen luominen. Koostamisprosessia vaiheistettaessa jaetaan ohjelmistokoosteiden luominen ensisijaiseen muutostenpäivitys-koostamisprosessiin ja toissijaisiin koostamisprosesseihin. Ensisijainen koostamisprosessi suoritetaan kevyellä testikuormalla, sen onnistuessa pidetään

muutostenpäivitysprosessia onnistuneena ja käynnistetään toissijaiset koostamisprosessit, joissa painopiste on testauksella. Toissijaisten ohjelmistokoosteiden luominen voidaan jakaa useampaan osaan esimerkiksi testauslajien mukaan: komponenttien toiminnan testaava koostamisprosessi, järjestelmätason toimivuuden testaava koostamisprosessi tai suorituskykyä testaava koostamisprosessi. (Duvall ym., 2007, 92–96, ja Fowler, 2006).

3.4 Testaus

Perinteisesti ohjelmiston koostamisella tarkoitetaan lähdekoodien kääntämistä, linkittämistä ja muita toimia, joilla ohjelmisto saadaan koottua suorituskelpoiseksi. Ohjelmisto voidaan siis suorittaa, mutta sen toiminnasta ei ole käytännössä varmaa tietoa. (Fowler, 2006.) Hyvä tapa löytää virheet nopeammin ja varmemmin, on ottaa automaattiset testit osaksi koostamisprosessia. Integraatiotestijoukon (*integration test suite*) päämääränä on varmistaa ohjelmiston toimivuus kokonaisuutena. (Fowler, 2006.) Pelkän savutestin suorittaminen koostamisprosessin yhteydessä ei varsinaisesti ole integraatiotestausta, vaan lisäksi tarvitaan myös joukko tarkempia testejä (Binder, 1999. 707, 711).

Jatkuvan integraation mukaisen koostamisprosessin yhteydessä suoritettava testijoukko voi koostua esimerkiksi yksikkötason, komponenttitason tai järjestelmätason testeistä. Tärkeintä koostamisprosessin yhteydessä suoritettavissa testeissä on niiden toistettavuus ja automaattisuus. (Duvall ym., 2007. 141–143.) Käytännössä koostamisprosessin yhteydessä tapahtuva testaus on regressiotestausta, jonka avulla pyritään varmistamaan se, ettei lisättyjen muutosten johdosta synny virheitä aiemmin toimivaksi havaittuihin ominaisuuksiin (Binder, 1999. 755–756.). Ilman jokaisen koostamisprosessin yhteydessä suoritettavia automaattisia testejä

menetetään suuri osa jatkuvan integraation tuomista eduista, sillä kehittäjien on mahdotonta luottaa tehtyjen muutosten toimivuuteen (Duvall ym., 2007. 15). Toisaalta Elssamadisyn (2007, 91) mukaan jo pelkkä koostamisprosessin automatisointi tekee jatkuvan integraation käyttöönotosta kannattavaa.

Liian hidasta koostamisprosessia voidaan usein nopeuttaa sekä tehostamalla että karsimalla siihen liitettyä testausta. Usein vaaditaankin tasapainoilua testien kattavuuden ja koostamisprosessin nopeuden välillä. (Fowler, 2006.) Duvall ym. (2007, 76–77) huomioivat suoritusjärjestyksen tärkeyden: mitä nopeammin yksittäisellä testillä voidaan osoittaa koostamisprosessin epäonnistuminen, sitä aiemmin se tulisi suorittaa.

Tärkeä osa ohjelmiston lopullista toimintaa on tuotantoympäristö, jossa ohjelmistoa tullaan lopulta suorittamaan. Onkin tärkeää huomioida, että kaikki eroavaisuudet testausympäristön ja tuotantoympäristön välillä ovat potentiaalisia esteitä virheiden löytymiselle. Testausta suunnitellessa päämääränä tulisi olla testausympäristö, joka mahdollisimman hyvin vastaa tuotantoympäristöä. Täydellisesti tuotantoympäristön kaltaisesti toimivien testausympäristöjen luominen on useassa tapauksessa käytännössä mahdotonta, mutta ympäristöjen välillä vallitsevat eroavaisuudet tulisi pyrkiä ottamaan huomioon ja dokumentoimaan. (Fowler, 2006.)

3.5 Käyttöönotto

Useissa tapauksissa jatkuva integraatio vaatii useiden erilaisten testausympäristöjen käyttöä: ensisijaisen koostamisprosessin testaus voidaan suorittaa omassa testausympäristössään ja toissijaisia koostamisprosesseja varten voidaan tarvita useita erilaisia testausympäristöjä. Tavallista useampia testausympäristöjä vaadi-

taan erityisesti useilla alustoilla toimivissa ohjelmistoissa. Tavoitteena on, että ohjelmistoa pyritään testaamaan koostamisprosessin yhteydessä tuotantokäyttöä vastaavasti, ja silloin jokaiselle käyttöalustalle tulisi olla oma testausympäristönsä. (Duvall ym., 2007. 191–194.) Koska ohjelmistokoosteita otetaan käyttöön näissä ympäristöissä useita kertoja päivässä, on käyttöönottoprosessi (*deployment process*) käytännössä välttämätöntä automatisoida, esimerkiksi skriptejä käyttäen. Käyttöönottoprosessista johtuvien virheiden minimoimiseksi tulisi sen olla mahdollisimman yhtenäinen kaikille järjestelmille. Tuotantoympäristön käyttöönottoprosessia tuskin suoritetaan päivittäin, mutta myös sen automatisointi on järkevää, sillä siten voidaan sekä vähentää toisteista työtä että estää inhimillisten virheiden aiheuttamia ongelmia. (Fowler, 2006)

Tuotantoympäristön käyttöönottoprosessissa on syytä panostaa varmatoimiseen automaattiseen palautukseen (*rollback*), jonka avulla voidaan palata viimeiseen toimivaksi todettuun tilaan (Duvall ym., 2007. 18). Automaattinen palautus vähentää käyttöönottoprosessiin liittyvää jännitystä ja epävarmuutta, siten rohkaisten suorittamaan sen useammin. Tuotantoympäristön automatisoidun käyttöönottoprosessin ja automaattisen palautusmahdollisuuden avulla voidaan käyttäjille tuottaa ominaisuuksia nopeammalla tahdilla. (Fowler, 2006.)

3.6 Tiedon jakaminen

Jotta kehitystyötä tekevien kesken voi syntyä luottamus siihen, että työtä tehdään vakaalla pohjalla, on kehitettävän ohjelmiston tilasta ja tehdyistä muutoksista kommunikoitava ryhmän kesken. Integraatiopalvelinohjelmistojen tuottamat verkkosivut mahdollistavat tarvittavan informaation jakamisen automaattisesti. Näiltä verkkosivuilta voidaan nähdä mm. viimeisimmän koostamisprosessin tila

(kesken/valmistunut), lopputulos (onnistunut/epäonnistunut), tehdyt muutokset ja muutosten tekijä. Verkkosivujen avulla myös sellaiset henkilöt, jotka ovat fyysisesti erillään muusta ryhmästä saavat automaattisesti ajan tasalla olevaa tietoa koostamisprosessin tilasta. Tätä kommunikaatiota ei tule unohtaa, vaikka käytössä olisi manuaalinen koostamisprosessi. (Fowler, 2006.)

Tieto koostamisprosessin epäonnistumisesta on jaettava mahdollisimman nopeasti, jotta ongelmiin voidaan reagoida heti ja jatkuvasta integraatiosta olisi todellista hyötyä (Duvall ym., 2007. 203). On kuitenkin tärkeää huomata myös se seikka, että liiallinen informaation jakaminen saattaa johtaa tärkeän informaation sivuuttamiseen. Duvallin ym. (2007, 205–209.) mukaan informaation jakamisen tuleekin tapahtua niin, että oikea informaatio jaetaan oikeille ihmisille oikeaan aikaan ja oikealla tavalla.

4 JATKUVAN INTEGRAATION KÄYTTÖÖNOTON TUOMAT EDUT

Edellisessä luvussa esiteltiin jatkuvan integraation mukaisen toimintatavan vaatimat käytänteet. Uuden toimintatavan käyttöönottoa harkittaessa tärkeässä roolissa ovat sen käyttöönotosta syntyvät vaikutukset. Tässä luvussa esitellään jatkuvan integraation käyttöönoton tuomat keskeisimmät edut ohjelmistokehitysprosessiin.

Ohjelmistoprojektin onnistumisessa riskien ennakoiminen ja hallinta ovat tärkeässä roolissa. (Boehm, 1991). Suuri osa ohjelmistoprojektiin liittyvistä riskeistä on sellaisia, joiden vaikutukset paljastuvat vasta integraatioprosessin yhteydessä (Larsson & Crnkovic, 2005). Jatkuvan integraation mukainen, usein tapahtuva koostaminen ja testaaminen helpottavat integraatioon liittyvien riskien hallintaa mm. seuraavasti:

- Ohjelmiston puutteet havaitaan ja korjataan aikaisessa vaiheessa (Holck & Jørgensen, 2003).
- Toteutuksen laadusta saadaan tietoa jokaisen muutoksen yhteydessä testauksen ja katselmointien avulla (Karlsson ym., 2000).
- Olettamukset jotka liittyvät testausympäristön asetuksiin, testauksen suoritustapaan ja koosteen luomiseen vähenevät – ne suoritetaan aina automatisoidusti samalla tavalla. (Fowler, 2006).

Jatkuvan integraation mukaisen, jokaisen muutospäivityksen yhteydessä tapahtuvan koostamisprosessin avulla voidaan osoittaa selvästi, mikä muutospäivitys on aiheuttanut ilmenneen virhetilanteen (Smart, 2009). Pienestä muutosjoukosta virheen aiheuttajan löytäminen on yksinkertaisempaa, sillä suuri osa tutkittavasta alueesta voidaan heti rajata pois (McConnell, 1996a). Toinen usein tapahtuvan

koostamisen selkeä etu on se, että tehty työ on vielä tuoreena muistissa: kehittäjän on huomattavasti kivuttomampaa palauttaa mieleensä sellainen tehtävä jonka parissa on työskennellyt saman päivän aikana, kuin tehtävä jonka valmistumisesta voi olla jopa useita kuukausia. (Fowler, 2006).

Jotta voidaan aikaansaada jatkuvaa integraatiota vastaava manuaalinen integraatioprosessi, tulisi muutoksia päivittävän kehittäjän suorittaa samat toimet jokaisen muutospäivityksen yhteydessä manuaalisesti. Muutospäivitysten määrän kasvaessa kuluu prosessin suorittamiseen manuaalisesti enemmän aikaa kuin integraatiopalvelimen käyttöönottoon ja käytön aiheuttamien ongelmien selvittämiseen. Vaihtoehtoinen ratkaisu on yhdistää muutokset harvemmin, mutta muutosjoukon kasvaessa virheiden selvittäminen vaikeutuu, projektin näkyvyys (*project visibility*) heikentyy ja yhdistämiskonfliktit yleistyvät. (Miller, 2008.)

Millerin (2008) keräämien tietojen perusteella 12 henkilön kehitystiimissä jatkuvan integraation käyttöönotto toi vähintään 40 prosentin säästön integraatioon käytettyyn aikaan ja laajempien projektien tapauksessa säästöt vielä kasvavat entisestään. Knibergin ja Farhangin (2008) artikkeli tukee Millerin havaintoa: Siirtymällä jatkuvan integraation käyttöön koostamisprosessiin liittyvä, kehittäjän suorittama viiden tunnin manuaalinen työ saatiin korvattua 20 minuuttia kestävällä automaattisella koostamisprosessilla.

Viivästettyyn, erillisessä integraatiovaiheessa tapahtuvaan integraatioon liittyy epävarmuus siitä, kuinka kauan vaiheen läpikäymiseen kuluu aikaa. Jatkuvan integraation tapauksessa riski on selvästi pienempi, sillä integraatioon liittyvät toimet suoritetaan useita kertoja päivittäin (Fowler, 2006).

Automaattinen ohjelmiston käyttöönottoprosessi helpottaa julkistusten (*release*) luomista, joten jatkuvaa integraatiota hyödyntäen voidaan ohjelmistosta julkistaa pieniäkin versiopäivityksiä varsin kivuttomasti (Smart, 2009). Jatkuvan integraation yhdistäminen pieniin julkistuksiin mahdollistaa Lindvallin, Muthigin, Dagninon, Wallinin, Stupperichin Kieferin, Mayn ja Kähkösen (2004) mukaan toimivan ohjelmiston toimittamisen aina tarvittaessa, joka on yksi ketterien menetelmien vaalimista periaatteista (Beck, Beedle, van Bennekum, Cockburn, Cunningham, Fowler, Grenning, Highsmith, Hunt, Jeffries, Kern, Marick, Martin, Mellor, Schwaber, Sutherland & Thomas, 2001). Knibergin ja Farhangin (2008) kokemusten perusteella manuaalisesti tapahtuva käyttöönotto oli sekä hidasta että ongelmille altista. Kun käyttöönottoprosessi automatisoitiin, julkistuksia kyettiin tuottamaan luotettavasti tasaisella tahdilla. Tämä mahdollisti uusien ominaisuuksien tuottamisen käyttäjille aiempaa nopeammin (Kniberg & Farhang, 2008.)

Mitä nopeammin käyttäjille voidaan tuottaa uusia ominaisuuksia, sitä nopeammin ominaisuuksista voidaan saada palautetta ja saatuun palautteeseen reagoida. (Cusumano & Selby, 1997). Käyttäjäpalautteen avulla voidaan kehitysprojektin ajautuminen sivuraiteille havaita ja torjua jo aikaisessa vaiheessa. (Richardson, 2006). Käyttökelpoisen julkistuksen saatavilla olon ansiosta projektin näkyvyys paranee, kun asiakkaalle ja muille kiinnostuneille sidosryhmille voidaan selvästi osoittaa, mitä on jo toteutettu ja mitä on toteuttamatta. Tämä on omiaan lisäämään sekä asiakastyytyväisyyttä että asiakkaan luottamusta projektin etenemiseen. (McConnell, 1996b. 407).

Noudattamalla jatkuvan integraation periaatteita kehitettävän ohjelmiston edistymisestä ja ohjelmakoodin laadusta saadaan palautetta jokaisen muutospäivityksen yhteydessä (Duvall ym., 2007. 203). Palaute työn laadusta lisää kehittä-

jien työskentelymotivaatiota ja pyrkimystä korkealaatuisen ohjelmakoodin tuottamiseen (Holck & Jørgensen, 2003). Laadusta saatavan palautteen ei tarvitse olla monisanaista: motivaatiota parantamaan riittää Kajko-Mattssonin, Jonsonin, Koroorianin ja Westinin (2004) artikkelin mukaan jo tieto siitä, onnistuiko koostamisprosessi vai ei.

Usein tapahtuva integraatio ja siihen yhdistetty testaus edellyttää testijoukon ja ohjelmakoodin rinnakkaista kehittämistä. Testijoukon jatkuva kehittäminen vaatii lisätyötä, mutta on samalla erittäin tehokas tapa paljastaa virheitä. Kehittäjien sitoutuessa jatkuvan integraation periaatteiden noudattamiseen tavoitteeksi muodostuu oikean, toimivan järjestelmän kehittäminen, pelkän järjestelmän parissa työskentelyn sijaan (Binder, 1999. 711.)

5 JATKUVAN INTEGRAATION KÄYTTÖÖNOTON HAASTEET

Jatkuvan integraation käyttöönottoa harkittaessa on tärkeää ymmärtää ne haasteet, jotka keskeisesti vaikuttavat käyttöönoton onnistumiseen. Vähintään yhtä tärkeää on ymmärtää millaisia ongelmia jatkuva integraatio voi aiheuttaa, ellei ongelmien torjumiseen kiinnitetä huomiota. Tässä luvussa esitellään jatkuvan integraation käyttöönotossa ilmenneitä ongelmia ja tärkeitä seikkoja, joihin tulee kiinnittää huomiota jatkuvan integraation käyttöönottoa harkittaessa.

Siirryttäessä viivästetystä integraatiosta usein tapahtuvaan integraatioon ovat tarvittavat muutokset kehitysprosessiin huomattavat: sekä koostamisprosessin, testijoukon että ohjelmiston varsinaisen lähdekoodin tulee olla integraatiohetkellä sellaisessa tilassa, että ohjelmistokoosteen luominen ja testaaminen on mahdollista. (Binder, 706.) Tämän mahdollistamiseksi on välttämätöntä jakaa kehitystyö pienempiin osiin kuin viivästetyn integraation tapauksessa. (Karlsson ym., 2000.)

Elssamadisy (2007, 92) huomioi, että jatkuva toimivien ohjelmistokoosteiden tuottaminen vaatii huomattavan määrän työtä ja edellyttää jokaisen kehittäjän sitoutumisen yhteiseen päämäärään: toimivien ohjelmistokoosteiden tuottamiseen. Duvall ym. (2007, 25) tukevat edellä mainittua ja tuovat esille tärkeän huomion: vastuuta jatkuvan integraation ja koostamisprosessin onnistumisesta ei voida siirtää vain yhden henkilön harteille, vaan on tärkeää ymmärtää että jatkuva integraatio vaikuttaa jokaisen kehitysprosessiin osallistuvan päivittäiseen työhön. Mooren ja Spensin (2008) artikkelin mukaan yksilötason sitoutumisen puute toimivan ohjelmistokoosteen tuottamiseksi aiheutti ongelmia: jouduttiin nimittämään erillinen integraatiotiimi, jonka vastuulla oli ohjelmistokoosteiden luominen ja ilmi tulleiden virheiden korjaus.

Koska jatkuvan integraation käyttöönotto koskettaa käytännössä jokaista ohjelmistoprojektin parissa toimivaa, on tärkeää panostaa siihen liittyvään ohjeistukseen. Selkeiden ohjeiden puute aiheuttaa ongelmia esimerkiksi koostamisprosessin epäonnistumisien määrän kasvuna. Hyvät perustelut uuden toimintamallin tuomista eduista ja vaatimuksista vaikuttavat positiivisesti sen käyttöönottoon vähentämällä muutosvastarintaa. (Koroorian & Kajko-Mattsson, 2008.)

Myös organisaation kyky omaksua uusi toimintamalli joutuu koetukselle jatkuvaa integraatiota käyttöönotettaessa. Motorolalla ongelmia aiheutui muutosten-hallintakomitean (*change control board*) roolin ja jatkuvan integraation yhteentörmäyksen vuoksi: pienikin muutos suunnitelmiin tuli hyväksyä muutostenhallintakomiteassa joka kokoontui kerran viikossa ja siitä johtuen tarvittavia muutoksia ei saatu toteutettua tarpeeksi nopeasti ja niiden yhdistämisessä päähaaraan ilmeni ongelmia. (Lindvall ym. 2004.) Toisaalta jos ryhmälle annetaan vapaat kädet tehdä suunnitelmista poikkeavia muutoksia, aiheutuu siitä sekä arkkitehtuurin rappeutumista että laadun heikkenemistä. Kehittäjät työskentelevät lyhytnäköisesti, tavoitteenaan onnistua seuraavan ohjelmisto-koosteen luomisessa ja testaamisessa, jolloin tehdään pikaisia korjauksia ajattelematta niiden todellisia, kauaskantoisia vaikutuksia. (Holck & Jørgensen, 2003.)

Jatkuvan integraation käyttöönoton alkuvaiheessa on luonnollista, että joudutaan tekemään runsaasti töitä toimivien ohjelmistokoosteiden tuottamiseksi. Projektin edetessä tulee koostamisprosessissa epäonnistumisen kuitenkin olla säännön sijaan poikkeus. Vaikka ohjelmistokoosteiden ylläpitoon ja korjaukseen kuluukin arvokkaana pidettävää työaikaa, ei jatkuvaa integraatiota tule hylätä paineen alla. Vaikeudet ohjelmistokoosteen luomisessa kertovat yleensä ongelmista kehitettä-

vän ohjelmiston toteutuksessa – eivät ongelmista koostamisprosessissa. (McConnell, 1996b. 414.) Jos koostamisprosessi onnistuu jokaisella suorituskerralla, on vaarallista tuudittautua siihen uskoon, ettei ohjelmiston toteutuksessa olisi lainkaan virheitä. Tuleekin ymmärtää se tosiasia, ettei käytännössä ole mahdollista luoda sellaista testijoukkoa, joka paljastaisi kaikki mahdolliset virheet. Koostamisprosessin jatkuva onnistuminen onkin yleensä osoitus siitä, että käytetyssä testijoukossa on parantamisen varaa. (Binder, 1999. 711.)

Jokaisen muutospäivityksen yhteydessä tapahtuva ohjelmistokoosteen luominen asettaa koostamisprosessin kestolle nopeusvaatimuksia: liian hidas koostamisprosessi ei palvele jatkuvan integraation tarkoitusta. Nopeusvaatimukset asettavat siten myös rajat jatkuvan integraation hyväksikäytölle laajoissa projekteissa. Esimerkiksi Windows NT:n tapauksessa täydellisen ohjelmistokoosteen luominen kesti 19 tuntia käyttäen yhtäaikaisesti useita koneita, joten on selvää että niin laajan ohjelmiston koostaminen jokaisen muutospäivityksen yhteydessä ei ole kannattavaa. (Maraia, 1999. 5.) Magennisin (2007) mukaan liian hitaan koosteprosessin ongelmat voidaan ohjelmiston koosta riippumatta välttää käyttämällä oikeita arkkitehtuuriratkaisuja ja jakamalla ohjelmistokoosteiden luonti useisiin rinnakkaisiin koostamisprosesseihin.

6 YHTEENVETO

Integraatio-ongelmien on havaittu olevan merkittävä riskitekijä ohjelmistokehitysprojektin onnistumiselle. Erilaisia lähestymistapoja integraatioon on lukuisia. Tässä tutkielmassa pureuduttiin niistä yhteen, jatkuvaan integraatioon.

Tutkielman tavoitteena oli esitellä jatkuva integraatio-käytänne ja käsitellä sen käyttöönoton tuomia etuja sekä haasteita, tarkoituksena vastata tutkielman tutkimusongelmaan. Aiheeseen tutustuttiin aikaisemman kirjallisuuden perusteella, kirjallisuuskatsauksen muodossa.

Jotta lukijalle olisi syntynyt kuva vaihtoehtoisista, aiemmista lähestymistavoista integraatioon, esiteltiin tutkielman aluksi kaksi erilaista integraatiostrategiaa: erilliseen integraatiovaiheeseen siirretty integraatio ja päivittäin suoritettava integraatio. Seuraavassa luvussa esiteltiin keskeiset jatkuvan integraation käytänteet, jotta lukijalle voisi syntyä kuva siitä, mitä jatkuvalla integraatiolla tarkoitetaan ja mitä sen käyttöönotto edellyttää. Näitä käytänteitä olivat tiedostojen keskittäminen, päivittäinen muutosten päivittäminen, ohjelmistokooste, testaus, käyttöönotto ja tiedon jakaminen. Tämän luvun perusteella jatkuva integraatio voidaan tiivistää seuraavasti: jatkuva integraatio on joukko hyväksi havaittuja käytänteitä, jonka mukaan toimimalla valmistuneet muutokset päivitetään versionhallintaan ja ohjelmiston toiminnallisuus varmistetaan kokonaisuutena jokaisen muutospäivityksen.

Kolmannessa luvussa tuotiin aiemman kirjallisuuden pohjalta esille ne edut, joita jatkuvan integraation on havaittu tuovan ohjelmistokehitysprosessiin. Keskeisimmiksi eduiksi voidaan mainita virheiden selvityksen helpottuminen, vähentyneet

integraatioriskit, kyky tuottaa julkistuksia nopeasti, parantunut kommunikaatio ja kehittäjien motivaation kasvu.

Viimeisessä luvussa käsiteltiin niitä haasteita ja esiteltiin sellaisia ongelmatilanteita, joita jatkuvan integraation käyttöönotto tuo mukanaan. Haasteita olivat mm. työtapojen muuttuminen, organisaation ja henkilöstön kyky omaksua uusi toimintamalli ja jatkuvasti epäonnistuva ohjelmistokoosteprosessi. Jatkuvan integraation käyttöönotosta aiheutuneita ongelmia olivat liian vahva luottamus siihen, että jatkuva integraatio paljastaa kaikki virheet, ja ohjeistuksessa epäonnistuminen. Esille tuotiin myös se seikka, ettei jatkuva integraatio välttämättä sovellu kaikkein laajimpiin ohjelmistokehitysprojekteihin.

Tutkielma vastasi sekä asetettuun tutkimusongelmaan että sen tueksi esitettyihin tutkimuskysymyksiin. Tutkielmassa luotiin lukijalle kuva jatkuvasta integraatiosta ja sen käyttöönoton mukanaan tuomista eduista ja haasteista. Vaikka jatkuvan integraation tuomat edut puhuvat puolestaan, tulee muistaa että käyttöönotolla saattaa olla myös negatiivisia vaikutuksia, erityisesti jos ongelmatekijöiden minimoiminen tai kunnollisen ohjeistuksen laatiminen laiminlyödään. Kandidaatintutkielman rajoitetun pituuden vuoksi katsaus aihepiiriin oli varsin suppea.

Aiempaa kirjallisuutta jatkuvan integraation todellisiin sovelluskohteisiin liittyen löytyi varsin niukasti. Aihepiirin jatkotutkimuksen kannalta voisi olla mielenkiintoista selvittää, kuinka tunnettu käsite jatkuva integraatio on alan harjoittajien piirissä, ja kartoittaa minkälaisten ohjelmistotuotteiden kehitystyössä sitä on hyödynnetty.

7 LÄHDELUETTELO

Beck, K. 2000 Extreme Programming Explained: Embrace Change. Addison-Wesley Longman.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D. 2001. Manifesto for Agile Software Development[online], [viitattu 5.5.2010]. Saatavilla www-muodossa <http://agilemanifesto.org/>

Berczuk S, P., Appleton B. 2002. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Addison-Wesley Longman.

Binder, R. V. 1999 Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman.

Boehm, B. W. 1991. Software Risk Management: Principles and Practices. IEEE Software. 8, 1 (Jan. 1991), 32-41.

Cusumano, M. A., Selby R. W. 1997. How Microsoft builds software. Communications of the ACM, 40, 6, 53-61.

Duvall P., Matyas S., Glover, A. 2007. Continuous integration: improving software quality and reducing risk. Addison-Wesley Professional.

Elssamadisy, A. 2007. Agile Patterns: The Technical Cluster. C4Media.

Fowler, M. 2006. Continuous Integration[online], [viitattu 10.5.2010]. Saatavilla www-muodossa <http://martinfowler.com/articles/continuousIntegration.html>

Fowler, M., Foemmel M. 2001. Continuous Integration (original version)[online], [viitattu 10.5.2010]: Saatavilla www-muodossa <http://martinfowler.com/articles/originalContinuousIntegration.html>

Herbsleb, J. D., Grinter, R. E. 1999. Splitting the organization and integrating the code: Conway's law revisited. Teoksessa Proceedings of the 21st international Conference on Software Engineering (Los Angeles, California, United States, May 16 - 22, 1999). ICSE '99. ACM, New York, NY, 85-95.

Holck, J., Jørgensen, N. 2003. Continuous Integration and Quality Assurance: a case study of two open source projects. Australasian Journal Of Information Systems, 11(1).

Kajko-Mattsson, M., Jonson, M., Koroorian, S., Westin, F. 2004. Lesson Learned from Attempts to Implement Daily Build. Teoksessa Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (Csmr'04) (March 24 - 26, 2004). CSMR. IEEE Computer Society, Washington, DC, 137.

Karlsson, E., Andersson, L., Leion, P. 2000. Daily build and feature development in large distributed projects. Teoksessa Proceedings of the 22nd international Conference on Software Engineering (Limerick, Ireland, June 04 - 11, 2000). ICSE '00. ACM, New York, NY, 649-65.

Kim, S., Park, S., Yun, J., Lee Y., 2008. Automated Continuous Integration of Component-Based Software: An Industrial Experience. In Proceedings of the 2008 23rd IEEE/ACM international Conference on Automated Software Engineering (September 15 - 19, 2008). Automated Software Engineering. IEEE Computer Society, Washington, DC, 423-426.

Kniberg, H., Farhang, R. 2008. Bootstrapping Scrum and XP under Crisis - A Story from the Trenches. Teoksessa Proceedings of the Agile 2008 (August 04 - 08, 2008). AGILE. IEEE Computer Society, Washington, DC, 436-444.

Koroorian, S., Kajko-Mattsson, M. 2008. A Tale of Two Daily Build Projects. Proceedings of the Third International Conference on Software Engineering Advances, ICSEA 2008, October 26-31, 2008, Sliema, Malta, 245-251.

Larsson, S., Crnkovic, I. 2005. Case Study: Software Product Integration Practices, Teoksessa Product Focused Software Process Improvement: 6th International Conference, PROFES 2005, Oulu, Finland, Springer, Lecture Notes in Computer Science, Volume 3547 / 2005, 272-285.

Larsson, S., Crnkovic, I., Ekdahl, F. 2004. On the Expected Synergies between Component-Based Software Engineering and Best Practices in Product Integration, Teoksessa Proceedings of the 30th EUROMICRO Conference (August 31 - September 03, 2004). EUROMICRO. IEEE Computer Society, Washington, DC, 430-436.

Larsson, S., Myllyperkiö, P., Ekdahl, F. 2007. Product integration improvement based on analysis of build statistics. Teoksessa Proceedings of the the 6th

Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (Dubrovnik, Croatia, September 03 - 07, 2007). ESEC-FSE '07. ACM, New York, NY, 505-508

Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J., Kähkönen, T. 2004. Agile Software Development in Large Organizations. Computer 37, 12 (Dec. 2004), 26-34.

Magennis, T. 2007. Continuous Integration and Automated Builds at Enterprise Scale[online] [viitattu 10.5.2010] Saatavilla www-muodossa <http://blog.aspiring-technology.com/?tag=/automated+builds >

Maraia, V. 2005. The Build Master: Microsoft's Software Configuration Management Best Practices. Addison Wesley Professional.

McConnell, S. 1996a. Daily Build and Smoke Test. IEEE Softw. 13, 4 (Jul. 1996), 144.

McConnell, S. 1996b. Rapid Development: Taming Wild Software Schedules. Microsoft Press.

Miller, A. 2008. A Hundred Days of Continuous Integration. Teoksessa Proceedings of the Agile 2008 (August 04 - 08, 2008). AGILE. IEEE Computer Society, Washington, DC, 289-293.

Moore, E. Spens, J. 2008. Scaling Agile: Finding your Agile Tribe. Teoksessa Proceedings of the Agile 2008 (August 04 - 08, 2008). AGILE. IEEE Computer Society, Washington, DC, 121-124.

Perry, D. E., Siy, H. P., Votta, L. G. 2001. Parallel changes in large-scale software development: an observational case study. ACM Trans. Softw. Eng. Methodol. 10, 3 (Jul. 2001), 308-337.

Pressman R, S., 2005. Software engineering: a practitioner's approach. 6. uusittu painos. New York: McGraw-Hill.

Richardson, J. 2006. Continuous Integration: A Cornerstone of a Great Shop. Methods & Tools, 14, 1, 20-27. Martinig & Associates.

Shore, J., Warden, S. 2007. The Art of Agile Development. O'Reilly Media

Smart, J. 2009. Where To Now with Build Automation? [online], [viitattu 23.4.2010] Saatavilla www-muodossa: <http://www.infoq.com/articles/build-automation-ci-atlassian>

Stolberg, S. 2009. Enabling Agile Testing trough Continuous Integration. AGILE '09: Proceedings of the 2009 Agile Conference. Washington: IEEE Computer Society, 369-374.

van der Storm, T. 2007. The Sisyphus Continuous Integration System. Teoksessa Proceedings of the 11th European Conference on Software Maintenance and Reengineering (March 21 - 23, 2007). CSMR. IEEE Computer Society, Washington, DC, 335-336.