T A R T U Ü L I K O O L

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse instituut

Infotehnoloogia eriala

Rauno Siimann

Continuous Delivery põhimõtete rakendamisest Eesti Töötukassa infosüsteemi arendusprotsessis

Bakalaureusetöö (6 EAP)

Juhendajad: Vambola Leping
Meelis Kull

Autor: …… "….." mai 2012
Juhendaja: ……. "….." mai 2012
Juhendaja: ……. "….." mai 2012

Lubatud kaitsmisele:
Professor: ……. "….." mai 2012

TARTU 2012

Sissejuhatus

Continuous Delivery e. pidev tarne on hulk tarkvara arendamise tavasid ja põhimõtteid, mille eesmärk on parandada tarkvara tarnimise protsessi. Uuritav teema pakub lahendust ühele tüüpilisele pikkade arendustsükilitega tarkvaraprojektides esinevale probleemile, milleks on tähtaegade planeerimine ja nendest kinni pidamine. Nimelt peaks pidev tarnimine võimaldama tarkvara tarnimist kliendile nädalapikkuste arendustsüklite tagant või lausa komponent haaval. Pidev tarnimine võiks pakkuda võrreldes senini kasutatava arendusprotsessiga eeliseid nii tarnijale, kui ka kliendile. Kliendi jaoks on oluliseld kasutegurid tehtava arendustöö läbipaistvus ja pidevalt ning stabiilselt kasvav rakenduse funktsionaalsus. Tarnijale tähendaks pidev tarnimine lihtsamini planeeritavaid tähtaegasid ja pidevat tagasiside voogu, mis omakorda tõstaks toote kvaliteeti.

Employment Information System(edasipidi EMPIS) projekti eesmärk on arendada Eesti töötukassa infosüsteemi. AS Webmedia on tegelenud infosüsteemi arendamisega Eesti Töötukassale alates aastast 2008. Kuna Eesti Töötukassa pakutavad teenused ja äriprotsessid on ajas pidevalt muutuvad ja täienevad vastvalt vajadusele ja ka seadusandlusele, tuleb ka infosüsteemi pidevalt täiendada, et see võimaldaks realiseerida kogu nende äriloogikat.

Pideva tarne põhimõtete teerajajaks on infotehnoloogia- ja juhtimisalase konsultatiooniga tegelev ettevõte ThoughtWorks Inc.[1] Oma uurimustöö kirjutan ma ThougtWorks töötajate Jez Humble'i ja David Farley raamatu põhjal "Continuous Delivery (2010)".[2] Alustuseks kirjeldan uurimuses ära hetkel kasutusel oleva tootmisprotsessi. Uurimuse käigus toon ma välja raamatu autorite poolt ära kirjeldatud vajalikud mõtteviisid, metoodikad, tavad ja tehnoloogiad pideva tarne juurtamiseks ühes tarkvaraprojektis. Järgnevalt tuleb üles loetleda, millised väljatoodud mõtteviisidest, metoodikatest, tavadest ja tehnoloogiatest on meil juba kasutusel ja millised pole. Sellele järgneb analüüs teemal "kas ja kuidas saaksime veel mitte kasutusel olevaid nõuandeid kasutusele võtta?". Antud bakalaureusetöö eesmärk pole mitte luua tegevuskava antud metoodika kasutusele võtmiseks, vaid esialgu ainult uurida selle kasutusele võtu võimalikkust antud projektis.

Kuna tegu on laia valdkonnaga, on minu töö eesmärgiks süveneda pideva tarne arendusega soetud aspekidesse:

- versioonihaldus
- läthtekoodi toimetamine versioonihaldusesse
- konfiguratsioonihaldus
- andmetehaldus
- pidev integreerimine

Sellegipoolest ei saa piirduda ainult tehnoloogiliste aspektidega, vaid tuleb lähemalt rääkida ka metoodika alustalast - tarnimisvoost.

1.Töötukassa infosüsteemi EMPIS ja selle arendusprotsessi tutvustus

Nagu sissejuhtatuses öeldud käsitleb minu töö pideva tarne(PT) mõistet ning selle juurutamist Eesti Töötukassa infosüsteemi(edasipidi EMPIS) arendusprotesessis. Selle peatüki eesmärk on lähemalt tutvustada meie praegust arendusprotsessi, kirjeldada arendusetapid ning tuua välja mõned olemasolevad puudused, millele loodame PT näol lahendust leida.

EMPISe puhul on tegu infosüsteemiga, mis võimaldab Töötukassal pakkuda töötutele teenuseid. EMPISe eesmärgiks on realiseerida Eesti Töötukassa funktsioone ja viia Töötukassa teenused töötuteni. Infosüsteem võimaldab näiteks töötuid arvele võtta, koostada töötutoetuse avaldusi, määrata töötutele töötutoetust. Lisaks sellele peab EMPIS oluliselt lihtsustama Töötukassa konsultatantide tööd, tehes nende eest ära igasugused automaatiseeritavad tegevused. Seadus näeb ette, et töötuna arvel olev isik peab reguaarselt käima Töötukassas konsultantidega kohtumistel ning kandma ette oma tööotsingu tegevustest eelneval perioodil. Kui töötu jätab kokkulepitud kohtumisele tulemata, on tegu rikkumisega, ning sellepuhul peab EMPIS tekitama konsultandile meeldetuletuse tema teenindatava töötu rikkumisest ning pakkuma ka võimalikke karistusi rikkujale. Lisaks lihtsatele protsessidele, pakub Töötukassa töötutele erinevaid teenuseid: koolitused, karjäärinõustamised, Euroopa Liidu teistes liikmesriikides tööotsingu nõustamised, töötoad, seminarid jne. Kuna pidevalt tekib uusi teenuseid ja vanad teenused võivad muutuda, on vaja pidevalt luua EMPISes uusi lahendusi, et see oleks töötukassa äriprotsessile vastav.

EMPISe arendusprotsess käib tsüklite kaupa. Ühte tsüklisse kogutakse kokkuleppeliselt hulk uusi teenuseid, muudatusi vanades teenustes, veaparandusi ja muud, mida soovitakse EMPISe järgmises versioonis näha. Tsüklisse kogutud tööd analüüsitakse ning neile koostatakse vastavad tehnilised spetsifikatsioonid. Edasi annavad arendajad töödele spetsifikatsioonide järgi ajahinnangud. Arendajate antud ajahinnangutele lisandub testimisele ennustav aeg ning ka lisanduv puhveraeg testimisest leitud vigade parandamiseks ja esile kerkinud probleeimde lahendamiseks. Iga arendustsükkel lõppeb tarnega.

Arendustsüklite pikkus võib olla üsna varieeruv sõltuvalt tööde hulgast.Senini on arendustsüklite pikkus jäänud 1,5 kuni 2,5 kuu vahemikku. Arendustsükli pikkus on tarkvara arendusprotsessis oluline faktor. Reeglina kaasnevad pikema tsükliga suuremad riskid. Mida pikem on

arendustsükkel, seda keerulisem on anda teostatavatele töödele täpseid ajahinnaguid. Sellest tulenevalt on ka suurem oht tähtaegadest üle minna ning tarnet lubatud tähtajaks mitte valmis saada. Lisaks on pikema arendustsükli puhul keerulisem näha ette potensiaalseid riske ja ohte projektile. Liiga pikas plaanis on ka riskide hindamine ebatäpsem ja nendega tegelemise meetmete tõhusus ebakindel. PT pakub arendustsükli lühendamiseks lahendusi, mille kasutamise võimalikkust EMPISe projektis kavatsen analüüsida.

Oluline termin tarkvara projektis on projekti töövoog. Mõiste töövoog kirjeldab protsesside järjestust, mille tulemusena valmib töötav ja kvalikteetne tarkvara. EMPISe arendusprotsessi töövoog näeb välja järgnev:

- analüüs
- analüüsi ülevaatus
- arendus
- koodiülevaatus
- testimine

Analüüsiga tegeleb analüütik ning selle protsessi tulemiks on analüüsitud komponent koos spetsifikatsioonide, andmemudeli ja prototüübiga. Analüüsile järgneb analüüsi ülevaatus, mille teostab reeglina testija. Kui leitakse analüüsis vigu, läheb tööülesanne analüütiku kätte tagasi, et vigu parandada. Vastasel juhul liigub ülesanne arendajatele realiseerimiseks.Pärast arendusfaasi antakse testijatele testida juba valmis tarkvara. Kui testimisel leitakse vigasid, suunatakse tööülesanne tagasi arendajate, vastasel juhul loetakse tarkvara verifitseerituks ning tuleb teostada veel koodiülevaatus. Koodi ülevaatuse teostab üks arendaja teisele. Koodi ülevaatuse esmane eesmärk on arendaja tehtud töö hindamine värske pilguga selleks, et ennustada võimalikke probleeme või pakkuda välja paremaid alternatiive. Ülevaatusel kontrollitakse, et kasutataks kokkulepitud programmeerimistavasid ning kirjutatud lähtekood oleks loetav ka teistele arendajatele peale autori enda. Ülevaatuse käigus võivad välja tulla ka võimalikud bugid.

2. Pidev integreerimine, tarnevoog

Enne, teemadesse detailsemalt laskumist on oluline tutvustada mõned PT teemaatikaga seotud põhiterminid.

2.1 Pidev integreerimine (inglise keeles *Continuous Integration* edasipidi CI)

Pidev integreerimine on üks olulisemaid, kui mitte kõige olulisem osa pideva tarnimise protsessist. CI mõiste puudutab lähtekoodi toimetamist versioonihaldusesse, kuid ka lähtekoodist tarkvara valmis ehitamist ja selle käivitamist testimis- ja töökeskkondadesse. CI kujutab endast tarkvara pidevat käivitamist kasutuskeskkonna sarnasesse keskkonda. Kasutuskeskkonnaks nimetame keskkonda, kus klient meie toodetud tarkvara kasutab. CI rakendamiseks on palju erinevaid vahendeid (tööriistu) nt. Hudson, Jenkins, CruiseControl. CI töövahendid jälgivad versioonihalduse serverit ja juhul kui sinna uus versioon toimetatakse, üritab CI vahend sellele lähtekoodi versioonile vastava tarkvara valmis ehitada.

Sellegipoolest on oluline aru saada, et CI ei tähenda töövahendit, mis pidevat tarkvara käivitamist teeb, vaid põhimõtet, mille kohaselt pärast iga lähtekoodile tehtud muutuse toimetamist versioonihaldusesse tuleb tarkvara ka käivitada ja veenduda, et see üldse tööle hakkab. CI puhul on oluline, et sama tsükliga seotud muudatused ja uuendused toimetatakse versioonihalduse serverisse samasse harusse üksteise otsa, mitte ei teha iga muudatuse jaoks eraldi haru. [3] Haru mõiste tähendab versioonihaldussüsteemides kahte eraldi koopiat meie projektist, kuhu võime paralleelselt erinevaid muudatusi tekitada. Harumudelit võib kasutada näiteks mingisuguse uue mooduli arendamiseks kogu ülejäänud projektist sõltumatult. Kui projektis on erinevad arendustsüklid samaaegselt arenduses, võib teha iga tsükli jaoks oma haru niimoodi, et eelneva tsükli muudatused oleksid olemas ka järgneva tsükli harus, aga järgneva tsükli muudatusi eelneva tsükli harus ei ole. PT soovitused ei keela harumudeli kasutamist, vaid piiravad selle kasutamist niimoodi, et ühte kindlasse harusse tehakse ainult vastava arendustsükli muudatused.

Pideva samasse harusse toimetamise tulemusena võib tihti kohata olukorda, kus mõni viimane muudatus on tarkvara ehitamise ära lõhukunud. Seega on oluliseks CI osaks meeskonna kokkulepe, et kui mõni muudatus lõhub tarkvara ehitamise protesessi ära, on kõigi

meeskonnaliikmete prioriteet võimalikult lühikese aja jooksul tekkinud probleem ehitamisega korda teha. Kui probleemi põhjust piisavalt kiiresti ei leita või ilmneb, et selle parandamine võtab kaua aega, tuleks ehitamise ära rikkunud muudatus versioonihaldusest kõrvaldada.

Selleks, et pärast iga muudatust saaks kiiresti ja mugavalt tarkvara ehitada ja käivitada ning olla veendunud, et ilmenud vead on tekkinud lähtekoodile tehtud muudatustest, tuleb kasutada tarkvara ehitamiseks ja käivitamiseks automaatseid skripte. Vastasel juhul võib tarkvara ehitamine ja käivitamine osutada väga ajakulukaks ning tülikaks tegevuseks, mille käigus on väga lihtne inimlikke tähelepanematuse vigu teha. Automaatsed ehitamise ja käivitamise skriptid on möödapääsmatud eriti siis, kui projektis on kastusel CI vahend. CI vahend kasutab samuti tarkvara ehitamiseks ja käivitamiseks, ette antud skripte. Kui skripte pole, ei oska CI vahend tarkvara ehitada ja käivitada.

EMPISe arendusprotsessis on kasutusel Jenkinsi CI server [5]. Jenkinsi CI server jälgib meie versioonihalduse keskkonda ja iga uue versiooni toimetamisel versioonihaldusesse, üritab Jenkinsi server ehitada ja käivitada rakendust. Edukal ehitamisel ja käivitamisel asendab Jenkins viimase oma keskkonnas töötava versiooni rakendusest uue versiooniga. Lisaks sellele käivitab Jenkinsi server ka automaattestid ning testide lõppedes koostab testraporti. CI vahendit on võimalik konfigureerida niimoodi, et kui automaatteste ei läbita, siis CI serveris uut versiooni ei käivtata ning tööle jääb vana viimati töökorras olnud versioon. Meie siiski hetkel oma CI serverit nii karmilt seadistanud ei ole.

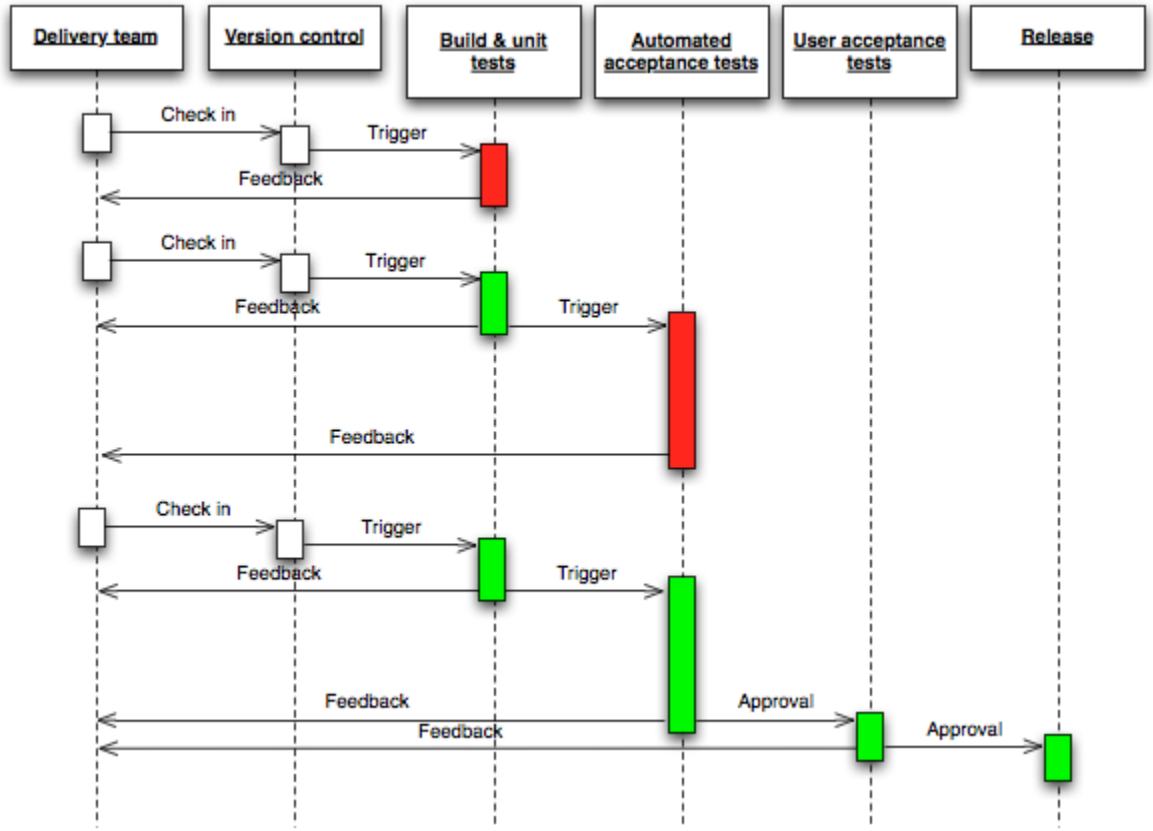
EMPISes on kasutusel 2 Jenkinsi CI keskkonda: empis-latest ja empis-current. Empis-latest on keskkond, kus käivitatakse alati arendadavas tsüklis viimane muudatus. Empis-latest on meie põhiline testimise keskkond. Latestis käivitatakse projekti automaattestid ning viiakse läbi ka manuaalne vastuvõtu testimine. Teine meie arendusmeeskonna hallatav keskkond on empis-current. Empis-currentis töötab alati viimane Töötukassale tarnitud ning Töötukassas kasutuses olev versioon rakendusest. See keskkond on vajalik selleks, et kui kliendi töökeskkonnas leitakse viga, on meil olemas täpselt sama versioon, mis hetkel Töötukassas kasutusel on, ning saaksime seal viga korrata, et selle põhjust otsida ning parandada.

Lisaks meie arendusmeeskonna hallatavatele keskkondadele on veel Töötukassa hallatav keskkond tk-test. Selles keskkonnas toimub kliendi testimine. Tk-test keskkonda anname välja aktiivsele arendustsüklile vastava versiooni rakendusest, mis on stabiilne ja meie

arendusmeeskonna poolt juba piisavalt testitud. Lisaks sellele ei sisalda üleantav versioon tõenäoliselt kõige uuemaid muudatusi.

2.2 Tarnevoog (Inglise k. *Deployment Pipeline*)

Tarnevoog on abstraktne pilt või skeem protsessist, mille tulemusena jõuab tarkvara versioonihaldusest kliendini. Tarnevoog kirjeldab etapid, mis on vaja läbida selleks, et veenduda tarkvara kvalikteedis. Skeem 2.1 kirjeldab ühe näidis tarnevoo ning selles aset leidvad protsessid.



[4] Skeem 2.1 - näidis tarnevoog

Tarnevoog algab enamasti lähtekoodi toimetamisega versioonihalduskeskkonda. Siinkohal on tarvilik etteruttavalt välja tuua fakt, et PT puhul on väga oluline võtmetegur see, et iga eraldi töö ülesanne või isegi väiksem parandus olemasolevale tarkvarale käivitab uue tarnevoo. Näide ühest tarnevoost võib olla järgnev:

- Versioonihaldusesse toimetamine

- Vastuvõtu testide jooksutamine
- Kliendi testimine
- Jõudluse testimine
- Kasutuskeskkonda paigaldamine

Tarnevoo oodatavaks tulemiks on töötav tarkvara ja aruanded tarkvara testimise kohta. Kuna tarnevoog tuleb käivitada iga tarkvarale tehtud väiksemagi muutuse järel, on oluline, et kõik skeemi etapid oleksid võimalikult automatiseeritud. Eeldatud on automaattestide olemasolu ja kõrge koodi kaetus testidega. Juhul kui tarkvara ei ole kõlblik mõnest skeemi etapist läbi laskmiseks, katkestatakse skeemi täitmine ning tagastatakse arendusmeeskonnale aruanne tarkvara vigade kohta. Kui meeskond parandab tarkvaras leitud vead, ei jätkata skeemi läbimist sealt, kust viga leiti, vaid alustatakse täiesti algusest. Sellepärast on korrektne öelda, et iga tehtud muudatus alustab uue tarnevoo. Iga edukalt lõpuni läbitud tarnevoog võib lõppeda tarnega ning tarkvara kliendi töökeskkonda paigaldamisega.

EMPISe arendusprotsessi tarnevoog koosneb jägnevatest etappidest: versioonihaldusesse toimetamine, ühiktestid, vastuvõtu testid, regressioonitestimine, klienditest ja tarkvara üleandmine. EMPISe tarnevoog ei ole vastavuses PT tarnevoo olulisemate põhimõtetega. Kuna meie projektis on automaattestide osakaal üpriski madal, siis ei ole võimalik, et iga tööga tehtud muudatus käivitaks täiesti uue voo, mis igakord algusest lõpuni läbitakse. Hetkel käivitatakse pärast tööülesandega tehtud muudatust ühiktestid ja olemasolevad automaatsed vastuvõtu testid. Suurema osa vastuvõtu testidest teevad testijad käsitsi. Samamoodi teevad testijad käsitsi ka regressioonitestimise. Regressiooni testimine toimub iga arendustsükli lõpus. Kuna käsitsi regressioonitestimise puhul on tegu ajakuluka tegevusega, mis võtab tavaliselt umbes nädal aega, on seda võimatu teha pärast iga muudatust.

Klienditestimine toimub samuti alles tsüklilõpus ning pole protsess, mida saaks iga muudatuse järel teha. Selleks, et meie arendusmeeskonnapoolset testimist täielikult automatiseerida, tuleks terve rakendus katta automaatsete vastuvõtu testidega, et peale iga muudatust oleks võimalik lisaks lisatud või muudetud komponendi testimisele teostada ka regressioontestimist. Kindlasti on see suur väljakutse, sest projekt on juba käinud 4 aastat, mis tähendab, et peaksime nelja aastaga tehtud funktsionaalsusele kirjutama järgi automaattestid. Suure tõenäosusega ei pruugi Töötukassa sellest huvitatud olla. Klienditestimist ei hakatagi tegema pärast igat muudatust, vaid kogutakse mingisugune hulk muudatusi kokku ja siis testitakse need koos. Sama meetodit tasuks

proovida ka meiepoolse testimise puhul. Sisuselist tähendaks selline lähenemine oluliselt väiksema tööde mahuga arendustsükleid ning sagedamini tarnimist.

PT pakub välja mõned tavad, mida tuleks tarnevoo täitmisel rakendada. Lähtekoodist tuleks käivitatav tulem ehitada ainult ükskord kogu skeemi jooksul. Skeemi jooksul ainult ükskord ehitamine on kasulik sellepärast, et skeemi erinevates etappides võivad olla erinevad tingumised nt. erinev kompilaatori versioon. Niimoodi võib juhtuda, et ehitame ühes etapis mingisuguse versiooni tarkvarast ja kinnitame selle vastavas etapis kõlblikuks. Edasi ehitamine järgmises etapis tarkvara uuesti ning arvame, et tegu on sama versiooniga, kuid tegelikult on meil tegu uue versiooniga ning isegi, kui tarkvara tunnistatakse vaadeldavas etapis kõlblikuks, ei saa me kindlad olla, et vaadeldavas etapis ehitatud tarkvara oleks ka eelmises etapis kõlblik olnud, kuna tegu ei ole sama versiooniga, mida eelmisel sammul testiti.

Igas tarnevoo etapis tuleb tarkvara käivitada ühtemoodi. See eeldab, et me kasutame igas keskkonnas sama käivitusskripti. Erineda võivad konfiguratsioonifailid erinevates keskkondades. Samamoodi käivitamine on igas etapis oluline selleks, et oleks võimalik veenduda, et selleks hetkeks, kui hakatakse tarkvara kliendi töökeskkonda käivitama, oleks juba eelnevalt piisavalt palju testinud, et käivitamine töötab edukalt. See tähendab, et pidev tarkvara käivitamine eelnevates etappides on harjutamine tarnimisel edukalt käivitamiseks. EMPISe projektis toimub kõikides senini automatiseeritud etappides käivitamine sama skriptiga. Mingis konkreetses keskkonnas käivitamiseks antakse rakendusele keskkonna konfiguratsiooni andmed ette konfiguratsioonifailist. Konfiguratsiooni failis on näiteks andmebaasi URI, andmebaasi skeemi nimi, andmebaasi parool. Konfiguratsioonifailis on kirjeldatud ka, kuidas toimub andmete initsialieerimine. CI peatükis kirjeldatud empis-latest ja empis-current keskkonnad töötavad mõlemad oma andmebaasiskeemi peal. Nendes keskkonades kirjeldatakse andmetele ligipääsemine omaette konfiguratsiooni failides.

Töökeskkonda käivitamisele eelnevatel tarnevoo sammudel tuleb käivitada alati võimalikult töökeskkonna sarnasesse keskkonda. Kõik testkeskkonnad nii automaatsete ja manuaalsete testide jaoks peavad olema sisuliselt töökeskkonna koopiad. Ainult niimoodi oleme testimisega usaldusväärselt tõestanud, et meie tarkvara ka töökeskkonnas töötab.

3. Konfiguratsioonihaldus

Konfiguratsioonihaldus on termin, mis kirjeldab protsessi, mille järgi kõiki projektiga seotud tulemeid ja ressursid salvestatakse, kätte saadakse, identifitseeritakse ja muudetakse. Kõige mugavam ja ilmsem meetod konfiguratsiooni haldamiseks on selle hoidmine versioonihalduses. On oluline tähtsustada, et versioonihalduses ei hoita ainult lähtekoodi vaid kogu informatsiooni, mis on vajalik, et luua automaatselt keskkond, milles tarkvara peab töötama. Seega tuleks lisaks lähtekoodile versioneerida veel:

- andmebaasi skripte
- automaatteste
- ehitmis- ja käivitusskripte
- rakenduse konfiguratsioonifaile
- kompilaatorit ja tööriistu

Selleks, et konfiguratsiooni versioneerimine efektiivselt töötaks, on oluline rakendada CI-t. CI kasutamine aitab vältida ühilduvuse probleemide tekkimist. Kui probleeme peaks siiski tekkima, avastatakse need vead suure tõenäosusega oluliselt varem, kui alles tarnimisel. Kasulik on ka sisukate kommentaaride kirjutamine lähtekoodi toimetamisel versioonihaldusesse. Hästi kommenteeritud muudatuste järgi on hiljem oluliselt lihtsam tuvastada, millise muudatusega ilmnenud viga võib olla tekkinud. EMPISe projektis lisatakse muudatuste versiooni haldusesse toimetamisel kommentaar seotud ülesande numbri ja kirjeldusega. Lisaks sellele on meil võimalik leida kõik ülesande raames muudetud ja tekitatud failid.

3.1 Konfiguratsioonihalduse vahendid

EMPISe arendusprotsessis kasutame Mercuriali versioonihaldustarkvara. Mercurial [9] on hajus versioonihalduse süsteem. Hajus versioonihalduse süsteem erineb tavaliselt versioonihalduse süsteemist sellepoolest, et arendajad ei toimeta tehtud muudatusi kohe ühte ja samasse tsentraalsesse repositooriumisse, vaid igal arendajal on oma arendusmasinas töötav lokaalne repositoorium, kuhu ta oma muudatused toimetab. Selleks, et tehtud muudatused teiste arendajateni jõuaksid, tuleb lokaalne repositoorium sünkroniseerida tsentraalsega ja laadida muudatused tsentraalsesse repositooriumisse. Hajusa versioonihalduse eeliseks on see, et muudatuste toimetamine versioonihaldusesse on lokaalne operatsioon. See võimaldab arendajal

pidevalt oma muudatusi lokaalsesse repositooriumisse toimetada, mis võimaldab sealt töö viimase eduka seisu taastamist juhul, kui arendaja peaks avastama, et on mingist hetkest alates valesti arendanud või kustutanud midagi, mis hiljem siiski oluliselt osutub. Kui arendaja tehtud muudatused on tarkvara katki teinud, saab ta minna tagasi viimati töötanud versiooni peale ja kustutada halvad muudatused lokaalselt nii et need üldse tsentraalsesse repositooriumisse ei jõuagi. Teine eelis hajusal versioonihaldusel tavalise ees on lähtekoodi ühilduvuse probleemide lahendamine lokaalselt. CI tava järgi tuleb tarkvara pärast iga muudatust ehitada ja käivitada. Et see oleks võimalik peavad kõik arendajad terve arendustsükli jooksul toimetama oma koodi samasse versioonihalduse harusse. Selline tegutsemine soodustab lähtekoodi ühilduvuse konfliktide teket. Hajus versioonihaldus aitab suurte konfliktide tekkimist leevendada kuna, hajusa versioonihalduse puhul on loomulikuks töö osaks oma muudatuste pidev lokaalsesse repositooriumisse toimetamine ja oma repositooriumi sünkroniseerimine tsentraalse repostitooriumiga. Sellise tegutsemise puhul lahendatakse konfliktid pidevalt nende tekkimisel mitte alles siis, kui arendaja on oma töö algusest lõpuni valmis teinud.

Tarkvaraga seotud välised teegid ei pea olema versioonihalduses, aga ka nendega on kasulik pidada ühtlast süsteemi, sellepärast, et suuretõenäosusega on erinevate tarkvara versioonidega seotud erinevad väliste teekide versioonid. Väliseid teeke ei ole kasulik hoida versioonihalduskeskkonnas sest, välised teegid võtavad palju salvestusruumi ja nende allalaadimine koos tarkvara versiooniga pikendab oluliselt tarkvara kätte saamist versioonihaldus keskkonnast. Samas ei tohiks väliseid teeke ka haldama jätta, sest muidu võib juhtuda, et uus meeskonna liige peab meeletult aega kulutama, et internetist õiged väliste teekide versioonid kätte saada. Hea soovitus on hoida väliseid teeke lokaalses asukohas, mis on kõigile projektiliikmetele kättesaadav. Lisaks lokaalsele hoiustamiskohale on soovitatav kasutada vahendeid, mis määravad väliste teekide versiooni ja tarkvara versiooni vastavuse.

Java projektide puhul on toimivad vahendid Apache Ivy ja Maven. Apache Ivy [6] on projekti sõltuvuste haldusvahend. Projekti sõltuvusteks nimetame projekti ehitamiseks vajalikke väliseid ressursse, mis on vajalikud selle ehitamiseks. Maven [7] on projekti ehitamise automatiseerimise vahend, mis võimaldab hallata ka projekti sõltuvusi. Informatsioon tarkvaga seotud teekide kohta sisaldub tarkvaraga seotud vastava vahendi konfiguratsiooni failis. Konfiguratsiooni failis kirjeldatakse ära välised ehitamiseks vajalikud ressursid ja nende asukohad. Seejärel laetakse need alla. Kui konfiguratsiooni faili, mis kirjeldab tarkvaraga seotud sõltuvused, säilitatakse

versioonihaldus keskkonnas, on sellega määratud üheselt tarkvara versioon ja sellega seotud väliste ressursside versioonid.

EMPISe projektis on juba kasutusel Apache Ivy.

3.2 Ehitamis- ja käivitamisskriptid

Kõige triviaalsemate projektide puhul piisab ehitamiseks ja käivitamiseks integreeritud arenduskeskkonna(IDE) võimalustest. Niipea kui projekti maht kasvab suuremaks kui üks arendaja ja ajakulu kasvab suuremaks kui mõned päevad, nõuab projekt rohkem tähelepanu ja hoolt, et see ei kasvaks liiga laialihajuvaks ja haldamatuks. See on eriti oluline skriptide kirjutamise puhul, muidu võib juhtuda nii, et uue meeskonnaliikme arvutis projekti käivitamine võib võtta päevi.

Ehitamisskriptide loomine ei ole tänapäeval kuigi keeruline, sest igal kaasaegsel platvormil on olemas võimalus käsurealt ehituskäske anda. Näiteks Java projektides Apache Ant'i [8] ja Maveni ehitusvahenditega. või.NET'i puhul MsBuild vahendiga. Nende vahenditega ehitamiskäsku võib ka automaatselt käivitada oma CI serverist. Empise projektis on kasutusel Anti skriptid ehitamiseks, käivitamiseks, propagaatori käivitamiseks,ühiktestide ehitamiseks ja käivitamiseks ja lähtekoodi versioonihaldusesse toimetamise tegevusteks. Skriptides on defineeritud kõik eelpool mainitud operatsioonid ning iga operatsiooni juures on võimalik määrata ka need operatsioonid, mis tuleb enne defineeritava operastiooni täitmist ära teha.

Projekti käivitamine ei ole sageli nii lihtne kui ehitamine. Tarkvara käivitamine testimiskeskkonda või töökeskkona on üldjuhul keerulisem, kui lihtsalt lähtekoodist kompileeritud tulemi kopeerimine töökeskkonda. Paljudel juhtudel kaasneb sellega rakenduse konfigureerimine, andmete initsialiseerimine ning lisaks veel operatsioonisüsteemi, infrastruktuuri ja vahevara seadistamine.

Et vigade tekkimist vältida, tuleks vältida üldiseid ehitus-käivitusvahendeid ning kasutada ainult neid, mis on töökeskkonna ja vahevara poolt kindlasti toetatud. Lisaks on selliste skriptide kirjutamisel oluline järgida kindlaid põhimõtteid.

Esiteks tarnevoo iga etapi jaoks peab olema skript. See tähendab, et me tahame, et meie skriptid kataks kõiki protsesse, mida me oma tarnevoo jooksul läbi viime. Skriptide vastavusse viimine tarnevoo etappidega aitab skripte paremini hooldada ja vähendab sõltuvust ehitamise ja käivitamise komponentide vahel. Projekti alguses on mõistlik teha ainult üks skript, mis kirjeldab

kõik tarnevoo operatsioonid ilma neid realiseerimata. Kui hakkame neid operatsioone realiseerima, siis tuleks see realisatsioon viia eraldi skripti. Seega oleks meil töövoogu tutvustava peatüki näite töövoo puhul olema eraldi skriptid. EMPISe projektis on automatiseeritud tarnevoo esimene pool. Olemas on skriptis ehitamiseks ja käivitamiseks ning ühiktestide ehitamiseks ja käivitamiseks. Samade skriptidega saab käivitada ka testimiskeskkondades. Automatiseerimata on aga protsess tarkvara üleandmiseks kliendile ja kliendi töökeskkonnas käivitamiseks. See on osaliselt tingutud sellest, et Töötukassa kasutab oma töökeskkonnana kolmandalt osapoolelt renditavat serverit, kuhu tarkvara iga uus versioon käivitatakse käsitsi kolmanda osapoole tehnikute poolt. Seega on hetkel meie projekti tarne tulemuseks war-fail. War-file [10] on veebirakenduse arhiivfail, mis sisaldab endas Java veebirakenduse sisu: Java servletid, Java klassid, XML konfiguratsioonifailid, rakenduse versioonile vastavad välised teegid ja staatilised HTML-lehed. Seega oleks terve tarnevoo täielikuks automatiseerimiseks vaja töötada välja plaan üheskoos Eesti Töötukassaga, et tarnimine ei sisaldaks kolmandat osapoolt ning kästsi tehtavaid tegevusi.

Igas keskkonnas peab kasutama samu skripte. Oluline on, et igas keskkonnas käivitamine käib samade tegevuste järgi. Ainult niimoodi saame olla veendunud, et ehitamis- ja käivitamstegevused on piisavalt testitud. Reaalselt saavutatakse see niimoodi, et käivitamiseks kasutatakse sama skripti ning keskkkonde erinevused nt. teenuste URL, andmebaasi IP ja autentimisinfo kajastatakse konfiguratsioonis, mida hallatakse eraldi. Konfiguratsioon võib seega olla erinevate keskkondade puhul mõnevõrra erinev. EMPISe projektis on nii arendajate keskkondades kui ka testkeskkonnas käivitamiseks kasutusel samad skriptid. Kliendi töökeskkonda käivitamine ei toimu aga enam skriptidega, vaid nagu eelnevalt öeldud kasutatakse töökeskkonnas käivitamiseks eelnevalt samade skriptidega valmis ehitatud tulemit. Vaatamata sellele, et tulemiks on automaatselt valmis ehitatud *war*-fail, on siinkohal võimalus probleemide tekkeks. Kuna töökeskkonda käivitamine pole automatiseeritud, mis tähendab, et see ei toimu täpselt samas järjekorras samade operatsioonide järgi nagu eelnevates faasides, võib juhtuda, et näiteks konfiguratsiooni info lisamine käivitamisel tekitab probleeme. Kindlasti ei saa väita, et töökeskkonda iga uue versiooni käivitamine oleks tarnevoo käigus pidevalt testitud operatsioon. Seega on töökeskkonda käivitamine koht, mis võib tekitada tarne ajal ebameeldivaid üllatusi.

Käivitamiskeskkond peaks pärast käivitamist jääma samasse olekusse, kus see oli enne käivitamist. Vastasel juhul ei pruugi sama protsessi järgi käivitamine ja ehitamine enam

järgnevatel katsetel õnnestuda. Lisaks ehitamise ja/või käivitamise ebaõnnestumisele raskendab selline olukord oluliselt vea põhjuse leidmist. Kui keskkond jääb alati pärast käivitamist samaks, siis teame, et vea põhjus peab olema rakenduses või selle skriptides. Kui aga keskkond võib muutuda, võib viga tulla hoopis keskkonnast.

Käivitussüsteemi tuleb arendada sammhaaval. Täielikult automatiseeritud käivitamise võlu on selles, et see võimaldab ühe nupu vajutusega tarkvara ehitada ja käivitada. Probleem seisneb selles, et ühe nupuvajutusega kogu projekti ehitamise ja käivitamise saavutamine ei ole lihtne. Kui ühte käivitussüsteemi lähemalt uurida, on näha, et tegu on lihtsate tegevuste kogumiga, mis ajajooksul järjest täienevad. Seega on sageli kasumlikum algul kirjutada skriptid, mis töötavad ainult arenduskeskkonas. Juba sellest on palju kasu, sest säästa alustuseks ainult arendajate aega. Edasi tuleb täiustada skripte, et need töötaksid ka testkeskkonas. Niimoodi tuleb liikudatestide täiendamisel mööda tarnevoogu, kuni lõpuks saab neid kasutada ka töökeskkonda ehitamiseks ja käivitamiseks. EMPISe skriptid on hetkel kasutatvad arendusetapis ja testimisetappidel. Kui meil õnnestub automatiseerida ka kliendile rakenduse üleandmise protsess, siis täiendame kindlasti skripte niimoodi, et need toetaksid ka tarnimiseprotsessi. Kuna projektis on kasutusel Jenkinsi CI server, millel jooksevad meie testimiskeskkonnad, oleks ilmselt üks mugavaim lahendus, et ka Töötukassa töökeskkonnaks oleks Jenkinsi CI server, kus on võimalik kas meie töötajatel või Töötukassa töötajatel käivitada ühe nupu vajutusega viimane meie poolt tarnitud versioon.

4. Andmete haldus

Nagu kõiksugused teised muudatused süsteemi, peaksid ka andmebaasi muudatused olema hallatud automaatsete protsessidega. See tähendab, et igasugused muudatused andmebaasi struktuurile ja andmete migreerimine peaksid olema tehtud skriptidega, mida saab versioonihaldus keskkonnas salvestada. Need skriptid peavad töötama igas andmebaasis, mida me oma tarnevoos kohtame. EMPISe projektis on kasutusel andmebaasi propagaator. Andmebaasi propagaator on vahend, mis jälgib, kas on tekkinud uusi andmebaasi skripte, mida meie rakendusega ühendatud andmebaasis pole veel käivitatud. Kui selliseid skripte on, siis need käivitatakse. Varem juba käivitatud skripte enam rohkem ei käivitata. Propagaatori kasu seisneb selles, et see teab oskab igas erinevas andmebaasi skeemis käivitada skriptid, mis seal veel käivitatud pole, et saavutada viimane versioon.

CI nõuab, et rakendus jääb töökorda pärast iga tehtud muutust. Kuna andmebaasi skeem, muutub koos rakendusega, tekib probleem, sellepärast, et andmebaasi skeem peaks vastama rakenduse versioonile. Kui me teeme vastavalt rakenduse uuele versioonile muudatusi ka andmbaasis, ei teki rakenduse viimase versiooniga probleeme. Probleemid tekivad siis, kui tahame versioonihaldusest rakenduse mõne vanema versiooni välja võtta, sest puudub strateegia andmebaasi vanema versiooni peale tagasi minemiseks.

PT pakub lahenduseks lihtsamate rakenduste puhul samm-haaval andmebaasi muutmist ning igale muudatusele eelnevasse olekusse taastamisskripti tegemise. Andmebaasi struktuuri haldamiseks on selline lahendus piisav, kuid see ei lahenda andmete migreermisega tekkinud probleeme. Oletame, et meil on versioonis n tabelid x ja y. Versioonis n+1 tuleb nõue, et kõik tabeli x kirjed tuleb üle kanda tabelisse y ja pärast seda tabel x kustutada. Selleks saame luua skripti, mis migreerib tabeli x kirjed tabelisse y ja kustutab tabeli x. Tabeli x kustutamisele saame teha ka kustumise pöördoperatsiooni täitva skripti, mis kustutatud tabeli uuesti loob. Märksa keerulisem on aga teha skripti, mis tabelist y suudaks välja eraldada need andmed, mis tabelist x sinna migreeriti ning need uuesti tabelisse x lisaks. Oluliselt lihtsam lahendus oleks siin juhul kustutatud tabel siiski ajutise tabelina koos oma kitsendustega alles hoida. Vanade tabelite alles hoidmisel ja nende peale tagasi pöördumisel võib tekkida selline konflikt, et uude andmebaasi on lisatud andmeid, mis vanade tabelite kitsendusi rikuvad. Vanasse andmebaasi skeemi nende andmete lisamine aga ei õnnestu. Suuremate projektide puhul muutub selline lahendus, kus igale andmebaasi muudatusele, tuleb teha ka tema pöördoperatsioon liiga aega nõudvaks ja kulukaks. Lisaks sellele muutub ka andmebaasi skeem pika sellise arendamise puhul üsna haldamatuks ja korrapäratuks. Sellepärast ei ole selline lahendus EMPISe projektis kõige mõistlikum.

Järgmine pakutav lahendus on, et rakendusest arendada algul versioon, mis töötab uuendustega kui ka uuendusteta andmebaasi skeemiga. Kui on tõestatud, et rakendus töötab täielikult ka uue skeemiga, võib anda välja järgmise rakenduse versiooni, mis töötab ainult viimase andmebaasi skeemi peal. Selline lähenemine annab tagasipöördumise võimaluse ainult ühe versiooni võrra. Kui PT põhimõtted on projektis täielikult kasutusel, pole selle strateegia puhul puhul probleemi, sest ideaalis ei tohikski tarvis olla tagasi pöörduda vanema versioonini kui viimane töötav versioon. Viimane strateegia on piisav olukorras, kus iga kõlblikuks tunnistatud tarkvara versioon, tarnitakse otse kliendile. Kuna EMPISe projektis veel täielikult PT’t ei rakendata, ei

oleks selline lähenemine meile piisav, kuna võib tekkida vajadus käivitada ka viimasest töötavast versioonist veel vanemaid versioone. Sellise rakenduse versiooni loomine, mis kahe erineva andmebaasi skeemiga töötaks, tekitab oluliselt lisatööd ning ei pruugi osades olukordades üldsegi võimalik olla. Teiselt poolt ei esine EMPISe projektis suuri andmebaasi muudatusi niivõrd tihti, et sellist strateegiat võimatu kasutusele oleks võtta.

Kui rakenduse uus versioon on vaja niimoodi käiku lasta, et kahe versiooni vahetamisel ei oleks tööseisakut, on võimalik rakendada sinise ja rohelise väljaande praktikat. Olgu meil 2 töötavat keskkonda rakendusega. Üks neist viimane töötav versioon. Nimetame seda roheliseks väljaandeks. Kopeerime rohelise versiooni ka teise keskkonda. Nimetame teise keskkonna siniseks väljaandeks. Nüüd käivitame rohelise väljaande peal skriptid, mis viivad läbi uued muudatused. Kui rohelises keskkonnas peaks läbi viidud muudatuste tõttu midagi katki minema, saame selle asendada kohe sinises keskkonnas töötava eelmise versiooniga. Kirjeldatud lähenemist ei pea kastuama ainult andmebaasi muudatuste läbi viimiseks. Selline lähenemine sobib rakenduse iga uue versiooni käiku laskmiseks sõltumata mis tüüpi muudatusi tehti.

5. Versioonihalduskeskkonda toimetamise faas

Versioonihalduskeskkonda toimetamise faas(edasipidi toimetamisfaas) algab sellise muudatusega projektile, mis kujutab endast muudetud lähtekoodi toimetamist versioonihalduse keskkonda. Toimetamisfaasi lõpuks loetakse ebaõnnestumise puhul veaaruaneid või õnnestumise puhul valmis ehitatud, käivitatavat tulemit, mida hakatakse käivitama järgnevates tarnevoo etappides. Toimetamisfaas tähistab tarnevoo algust. Juba CI rakendamisest üksinda piisab selleks, et tarnevoos oleks olemas toimetamisfaas. Esimese sammuna tarnevoos on toimetamisfaas olulise tähtsusega vähendamaks projektis kuluvat aega lähtekoodi muudatuste integreerimisele. Kuna terve tarnevoo eesmärgiks on leida ja kõrvaldada sellised tulemid, mis ei kõlba tarnimiseks, siis võib öelda, et toimetamisfaas on skeemis kõige esimene proovikivi, mis üritab kõrvaldada tahtmatud tarnekandidaadid enne, kui need meile probleeme jõuavad tekitada. Selleks, et seda etappi võimalikult hästi realiseerida tuleb jälgida kindlaid tavasid ja põhimõtteid. Toimetamisfaas peab andma kiiret ja kasulikku tagasisidet tarkvara kohta. Toimetamisetapp peaks läbi kukkuma järgnevatel juhtudel: kompileeritavate keelte puhul on tehtud süntaksiviga ning tarvara annab kompileerimisvea, muudatus on tekitanud rakendusse semantilise vea ning

esmased testid kukuvad läbi, rakenduse konfigratsioonis või töökeskkonnas on tekkinud viga. Igal juhul peaks toimetamisfaas ebaõnnestumise korral andma meile tagasisidet, millise eelpool toodud põhjusega meil tegu peaks olema. Mida varem vead avastatakse, seda lihtsam ja odavam on neid parandada. Kui arendaja teeb mingisuguse muutuse, mis selles faasis testid katki teeb ja vea põhjus kohe välja ei paista, on loomulik lahendus hakata vaatama üle muudatusi, mis tehtud pärast viimast töötavat versiooni. Et selline lähenemine võimalikult kiiresti ja mugavalt toimiks tasub hoida korraga tehtud muudatuste maht väike.

Oluline on mõelda läbi, mis on toimetamisfaasi läbimiskriteeriumid. Traditsiooniliselt kontrollitakse toimetamisfaasis ainult seda, et kõik testid edukalt läbitaks. Siis tuleks ka arvestada seda, milline on testide kvalikteet ning kui suurel määral on lähtekood testidega kaetud. Kui kõik testid edukalt läbitakse, aga koodikattuvus on ainult 10%, ei tohi testide edukas läbimine panna meid arvama, et meie koodis vigu pole, kuna suure tõenäosusega neid lihtsalt ei kontrollitud. Soovitatav on kehtestada mingisugused lävendid, koodi kattuvuse protsendile, millest alla poole langemise korral etapp läbikukkunuks loetakse. Samuti tasuks ka jälgida kompileerimishoiatuste hulka. Hoiatustele tasuks samamoodi kehtestada maksimaalne piir, mille ületamisel toimetamisfaasi ebaõnnestunuks loeme.

Toimetamisfaasis on peale lähtekoodi olulisteks teguriteks ka ehitamisskriptid, testide käivitamise skriptid, koodi analüüsivahendid ja toimetamistestid. Neid tulemeid tuleb samamoodi pidevalt hooldada ja parandada nagu muid rakenduse osasid. Hooldamine ja parandamine peaks toimuma pidevalt, sest kui need korra käest lasta, siis hiljem on nende parandamine juba oluliselt keerulisem ja aeganõudvam. Lõpuks võib see hakata kulutama tõsiselt aega arendamisele. Nagu eelpool mainitud peavad skriptid olema jagatud eraldi osadeks, millest igaüks tegeleb oma konkreetsete operatsioonidega. Kui on näiteks üks ainuke mitmetuhanderealine skript, siis on seal oluliselt keerulisem tekkinud viga tuvastada, kui juhul, kus iga tegevuse jaoks on eraldi oluliselt väiksemamahuline skript. Niimoodi saab kiiremini tuvastada, kus täpselt viga tekkis. Võit tuleb ka sellest, et skriptide jaotamisel iseseisvateks juppideks tekib modulaarsus, mis tähendab, et iga skript on omaette olem ning ei ole läbinisti seotud teiste skriptidega. EMPISe projektis on olemas teatud skriptide jaotus, kuid seda oleks võimalik tublisti parandada. Hetkel on ühes Ant-skriptis defineeritud operatsioonid rakenduse ehitamiseks, andmebaasi propagaatori käivitamiseks ja testide käivitamiseks. Sellele lisandub eraldi moodul Mercuriali versioonisehalduskekskonda toimetamisskripte. Testide ehitamine on

küll rakenduse enda ehitamisest eraldi operatsioon, kuid tõenäoliselt oleks lihtsam ja arusaadavam, kui kõik testimisetapiga seotud skriptid eraldi moodulis asuksid. Rohkem lihtsust võiks tuua ka andmebaasiga seotud operatsioonide eraldamine rakenduse ehitamise operatsioonidest.

Arendajad peavad oma tehtud muudatuste eest vastutama. Osades arendusmeeskondades on määratud eraldi ametikohad, millel töötavate spetsialistide ülesanne on tarnevoo jälgimine ja sellega seotud probleemide tuvastamine. Selline lähenemine on ebaefektiivne, sellepärast, et see tekitab lisa barjääri arendajatele tegemaks kiireid parandusi ja muudatusi. Lisaks tekib sellise lähenemise puhul olukord, kus tekkinud probleeme võidakse edasi lükata ning CI keskkonda jääb katkine versioon. Probleemide lahendamine lükkub siinjuhul edasi selle tõttu, et selleks ajaks, kui tarnevoogu jälgiv spetsialist vea avastab ning selle eest vastutava arendaja tuvastab, on vastutav arendaja juba liikunud edasi järgmiste ülesannetega ning võib nendega pikalt hõivatud olla enne, kui saab vea korda teha. Ebaefektiivne lahendus oleks ka see, et tarnevoogu jälgiv spetsialist ise hakkaks viga parandama, sest kuna see viga on kellegi teise tehtud, on selle vea autor ilmselt kontekstiga oluliselt paremini kursis ning suudab vea oluliselt kiiremini parandada, kui seda suudaks teha isik, kellele kontekst võõras on. Sellepärast peaks olema iga arendaja enda ülesanne kontrollida, et tema tehtud muudatused toimetamisfaasi läbiksid ja vigade ilmenemisel need otsekoheselt ära parandaksid.

Töötukassa projektis on kokkulepitud, et arendaja jälgib pärast oma muudatuse jõudmist Jenkinsi CI keskkonda, kas see ka seal probleemideta valmis suudetakse ehitada ja üles käivitada. Kui arendaja tehtud muudatus, Jenkinsis rakenduse ära lõhub, teeb ta selle ka korda. Meie meeskonna puhul ei ole see isegi niivõrd raudne reegel, vaid pigem hea tava, kuid sellest peetakse siiski tublisti kinni. Juhuks, kui arendaja ise unustab kontrollida, et tema muudatus Jenkinsis rakendust ära ei lõhkunud, saadab Jenkinsi server tervele arendusmeeskonnale ka e-kirju juhul, kui Jenkins leiab, et lisatud muudatused pole aktsepteeritavad.

Suuremate arendusmeeskondade puhul tasub pidada korrapidajat (inglise keeles *Build Master*). Korrapidaja ülesanne ei peaks olema teiste vigade parandamine, vaid jälgimine, et uued muudatused oleksid kvalikteetsed ning rakendust ära ei lõhuks. Vigade tekkimise korral peaks korrapidaja ülesandeks olema vastutava arendaja teavitamine tekkinud probleemist ning tema innustamine tekkinud probleemi lahendama. Kuna selline roll on kohati tüütu ja ebahuvitav, ei peaks ükski meeskonna liige seda pidevalt täitma. Korrapidaja roll peaks meeskonnasiseselt

teatud ajavahemiku tagant vahetuma. Korrapidaja kohustus võiks ühe meeskonnaliikme jaoks kesta optimaaselt üks nädal. EMPISe projektis hetkel otseselt sellist inimest ei ole, kelle ainuke või kõige olulisem ülesanne oleks CI-keskkonnas pidevalt silma peal hoida. EMPISe projektis rakendatakse küll korrapidaja ametit, kuid see roll on meie meeskonnas oluliselt erinev, kui PT puhul kasutatav *Build Masteri* oma. Meie projektis pole veel niivõrd suur arendusmeeskond, et oleks vajadus sellise rolli järele. Meie projektis on korrapidaja ülesandeks lähtekoodi rakenduse korrastustööde tegemine: refaktoorimise ülesanded, ühiktestide hooldus ja parandamine, staatiliste lähtekoodi analüüsivahendite käivitamine ja nende koostatud aruannete järgi koodis paranduste tegemine.

Kokkuvõte

Antud lõputöö eesmärk oli viia ennast kurssi pideva tarnimise arendusega sotud põhimõtetega ja tavadega ning analüüsida võimalikkust ja vajalikkust kasutada neid meie projektis. Tekkis ka ülevaade meie projekti töövoost ning projektis juba kasutusel olevatest vahenditest või pakutud vahendite alternatiividest meie projektis.

Selgus, et pideva tarnimise põhimõtete hulgas on soovitusi, mida tahaksime omaks võtta või vähemalt mõningat aega katsetada. Valitud soovituste juurutamised ja katsetamised toimuvad juba väljaspool selle töö skoopi. Leidus ka selliseid põhimõtteid, mis pole meie tööprotsessi puhul sobilikud ning ei tooks meile piisavalt kasu või loovad koguni piiranguid meie kasutusel olevale tööprotsessile.

Implementing the Principles of Continuous Delivery in the Development Process of Estonian Unemployment Insurance Fund's Information System

Bachelor Thesis

Rauno Siimann

Summary

Continuous Delivery is a set of principles and patterns of realeasing software. The subject of Continuous Delivery aims to solve a common problem in software developing process which is planning the deadlines and adhering to them. Continuous Delivery should enable weekly releases of software to the user. Continuous Delivery is considered to be mutually beneficial to developing team as well to the user beacause it introduces incremental changes, stable growth of functionality and transparecny to the development process.

The topic of Continuous Delivery was analysed on the basis of Estonian Unemployment Insurance Fund's information system called EMPIS(Employment Information System). The development of EMPIS has been going on since 2008. The aim of this thesis is to analyse the possibility of implementing the princibles and patterns of Continuous Delivery in the development process of EMPIS. This thesis does not provide a step-by-step plan on how to employ principles and recommendations of Continuous Delivery in our development process, but it provides an analysis whether they are beneficial and employable to our project at all.

Material on Continuous Delivery derives from the book "Continuous Delivery (2010)" written by Jez Humble and David Farley of ThoughtWorks Inc.

Viited

1. http://www.thoughtworks.com/
2. Humble, Jez; Farley, David (2010). *Continuous delivery* (1. print. ed.).
3. Haru mõiste http://www.ericsink.com/scm/scm_branches.html
4. Jez Humble's work blog. *Näidis skeem tarnevoost* http://continuousdelivery.com/2010/02/continuous-delivery/, 14.05.2012.
5. Jenkins CI . *Meet Jenkins* https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins, 14.05.2012.
6. The Apache Ant project. *Apache Ivy documentation* http://ant.apache.org/ivy/history/latest-milestone/index.html, 14.05.2012.
7. Apache Maven. *Wikipedia* http://en.wikipedia.org/wiki/Apache_Maven, 14.05.2012.
8. Apache Ant http://ant.apache.org/
9. Mercurial. *Understanding Mercurial* http://mercurial.selenic.com/wiki/UnderstandingMercurial, 14.05.2012.
10. War file. *Wikipedia* http://en.wikipedia.org/wiki/WAR_file_format_%28Sun%29, 14.05.2012.