TARTU ÜLIKOOL

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse instituut

Tarkvaratehnika eriala

Mirjam Rauba

Pideva tarnimise strateegia rakendamise analüüs AS Webmedia Group projektis EMPIS

Magistritöö (30 EAP)

Juhendajad: dotsent Helle Hein

Artur Assor, AS Webmedia Group

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

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

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

Lubada kaitsmisele

Professor ....…………………….……. "….." mai 2012

TARTU 2012

Sisukord

1 Sissejuhatus 4
2 Pideva tarnimise strateegia *(Continuous Delivery)* 6
3 Põhimõtted ja praktikad pideva tarnimise strateegia rakendamiseks 9
3.1 Lahendused, mida tarkvara tarneprotsessis vältida 10
3.2 Kuidas saavutada eesmärk 11
3.3 Pideva tarnimise strateegia rakendamisest saadav kasu 12
3.4 Toodangu kandidaat 13
3.5 Tarkvara üleandmise/tarnimise põhimõtted 14
3.6 Pidev integratsioon *(Continuous Integration)* 16
4 Konfiguratsiooni haldamine 18
4.1 Versioonihaldus 18
5 Testimisstrateegia väljatöötamine 21
5.1 Testide tüübid 22
6 Paigalduskonveier 25
6.1 Paigalduskonveieri praktikad 28
6.2 Kehtestusfaas *(Commit stage)* 30
6.3 Automaatsete vastuvõtutestide etapp 31
6.4 Järgnevad testimise etapid 32
6.4.1 Käsitsi testimine 33
6.4.2 Mittefunktsionaalne testimine 33
6.5 Väljalase 33
7 EMPIS arendusprotsess 35
7.1 Arendusprotsessi kirjeldus 35
7.2 Versioonihaldus 38
7.2.1 Mercurial 38

7.2.2 Andmebaasi propagaator ... 38
7.3 Jenkins ... 39
7.4 Testimisstrateegia ... 41
7.4.1 Automaattestid ... 42
7.4.2 Testimiskeskkonnad ... 42
8 Pideva tarnimise strateegia rakendamine EMPISes ... 44
8.1 Tarkvara tarnimise põhimõtted ... 44
8.2 Versioonihaldus ... 47
8.3 Testimisstrateegia ... 48
8.4 Paigalduskonveier ... 49
8.5 Mida võiks EMPISes kasutusele võtta? ... 50
8.5.1 EMPIS paigalduskonveier ... 51
8.5.2 Automaatsed vastuvõtutestid ... 52
9 Kokkuvõte ... 53
Summary ... 54
Sõnastik ... 55
Allikad ... 57

1 Sissejuhatus

Tarkvaraarenduse maailm muutub ja areneb pidevalt, mistõttu tarkvaraarendusega tegelevad ettevõtted ja õppeasutused peaksid neid muutuseid jälgima ja nendega kohanduma. Üha vähem kasutatakse tarkvara arendamiseks koskmudelit ja üha enam võetakse kasutusele agiilseid arendusmetoodikaid. Selline tendents on tingitud eelkõige asjaolust, et tarkvara tellija nõudmised muutuvad pidevalt ning soovides konkurentsis püsida, tuleb neile muutustele kiiresti reageerida.

Üks uudsemaid lähenemisi tarkvaraarenduses on pideva tarnimise strateegia *(continuous delivery)* rakendamine. Seda strateegiat levitab tarkvaraarenduse ja konsultatsiooniga tegelev ettevõte ThoughtWorks Ltd. [1], mille praegune töötaja Jez Humble ja endine töötaja David Farley kirjutasid antud teemal ka raamatu. Raamat anti välja aastal 2010 ning raamatu pealkirjaks on *„Continuous Delivery“* [2]. Lisaks raamatule ja ThoughtWorks Ltd. poolt pakutavatele konsultatsioonidele on pideva tarnimise strateegiat tutvustatud ka erinevatel ülemaailmsetel tarkvaraalastel konverentsidel. Näiteks, Jez Humble tegi ettekande suurimal agiilse tarkvaraarenduse teemalisel konverentsil Agile2010 [3] ja Agile2011 [4]. David Farley aga tutvustas pideva tarnimise strateegiat Java arenduse teemalisel konverentsil Devoxx 2011 [5].

Pideva tarnimise starteegia on kogum põhimõtteid ja praktikaid, mille õigel rakendamisel on võimalik saavutada olukord, kus iga rakenduse versioon, mis läbib kõik tarnimisprotsessi sammud, on paigaldatav toodangukeskkonda. Põhirõhk on automatiseerimisel, mis muudab tarneprotsessi kiiremaks, korratavaks ja seeläbi usaldusväärsemaks. Samas ei tehta järeleandmisi kvaliteedis. Eesmärk on pakkuda kliendile pidevalt uuenevat ja kvaliteetset tarkvara.

Antud töö eesmärgiks on uurida pideva tarnimise strateegia rakendamise võimalikkust AS Webmedia Group projektis EMPIS (*Employment Infosystem*). Selleks uuritakse pideva tarnimise strateegia põhimõtteid ja praktikaid. Samuti kirjeldatakse EMPISes kasutusel olevat arendusprotsessi ning võrreldakse seda pideva tarnimise strateegia tavadega, et leida võimalusi arendusprotsessi täiustamiseks. Soov protsessi muutmiseks tuleneb väga pikast regressioontestimise faasist, mis kolme testijaga kestab kuni kaks nädalat. See tähendab, et arendustsüklid on mitme kuu pikkused, kuna soovitakse kliendile tarnida mõistlikus mahus

uut funktsionaalsust ning mitte kulutada liiga suurt osa ajast regressioontestimise peale. Ideaalis võiksid arendustsüklid kesta paar nädalat, et tagada kliendile pidevalt uuenev tarkvara. Seeläbi oleks kliendi rahulolu suurem, kuna erinevad uuendused ja täiendused jõuaksid kasutajateni kiiremini. Arendusprotsessi on vaja muuta tarnete kvaliteedi tagamiseks. Pideva tarnimise strateegia võib olla lahendus, kuid selle rakendamine eeldab põhjalikku analüüsi.

Lisaks üldisele arendusprotsessi täiustamisele keskendutakse antud töös ka testimisprotsessi efektiivsemaks muutmisele, kuna antud töö autor on ametilt testija. Arendusprotsessi muutmisel tuleb arvestada ka kliendi soovide ja võimalustega, kuna enne uue tarkvaraversiooni üle andmist läbib see ka klienditestimise. Seega tuleb arvestada, et kui anda tihedamini üle uusi tarkvara versioone, peab ka klient leidma tihedamini aega testimise jaoks, samas on ühekordne testimise maht väiksem.

Töö tulemusena valmib EMPISe arendusprotsessi parendamise juhend, mis on eelkõige kasulik EMPISe arendusmeeskonnale. Samas võiks antud töö huvi pakkuda ka teistele projektimeeskondadele Webmedias, erinevatele tarkvaraarenduse ettevõtetele ja tarkvaraarendust õpetavatele asutustele.

2 Pideva tarnimise strateegia *(Continuous Delivery)*

Aastal 2001 kohtus grupp iteratiivsetest ja agiilsetest meetoditest huvitatud tarkvaraarendajaid, kes leppisid kokku ühistes põhimõtetes. Nad võtsid kasutusele termini *„agile“*, moodustasid *„Agile Alliance“* [6], koostasid manifesti ja panid kirja 12 agiilse tarkvaraarenduse põhiprintsiipi. Pideva tarnimise strateegia on nime saanud agiilse tarkvaraarenduse esimese printsiibi järgi: „Kõige olulisem on tagada kliendi rahulolu, tarnides talle vajalikku tarkvara võimalikult kiiresti ja tihti“ *(ingl.k Our highest priority is to satisfy the customer through early and continuous delivery of valuable software)*. Mõiste võtsid kasutusele Jez Humble ja David Farley oma raamatu *„Continuous Delivery“* pealkirjana.

Pideva tarnimise strateegia (CD) on kogum põhimõtteid ja praktikaid, mida tarkvaraarenduse maailmas üha laialdasemalt kasutusele võetakse, seda sooviga vähendada aega ja riske, mis on seotud tarkvara uue versiooni üleandmisega kliendile. Selle saavutamiseks suurendatakse tarkvara hetkeseisu kohta saadavat tagasisidet, tõhustades suhtlust arendajate, testijate ja ülejäänud meeskonnaliikmete vahel, kes on seotud tarkvara tarnimisega kliendile. Need tehnikad kindlustavad, et tarkvara muudatustele, veaparandustele või uue funktsionaalsuse realiseerimiseks kuluv aeg on minimaalne ja probleemsed kohad leitakse varakult, kui neid on lihtne parandada.

Mary ja Tom Poppendieck esitasid oma raamatus „*Implementing Lean Software Development“* küsimuse: „Kui kaua kulub sinu ettevõttel aega muudatuse paigaldamiseks, kui muudatus sisaldab ainult ühte koodirida? Kas seda tehakse korrataval ja usaldusväärsel moel?“[7]. Seda küsimust võib esitada igale tarkvaraarendusmeeskonnala ning sealt tuleneb ka üks tähtsamaid meetrikaid/mõõdikuid tarkvaraarenduses – tsükliaeg. See on aeg, mis kulub otsusest lisada täiendus kuni selle uuenduse toodangukeskkonda jõudmiseni. Pideva tarnimise strateegiat rakendades üritatakse vähendada tsükliaega ja muuta tarneid kvaliteetsemaks. Selle saavutamiseks kasutatakse täielikult automatiseeritud, korratavat ja usaldusväärset tarneprotsessi, mis koosneb järgnevatest arendusetappidest: ehitus*(build)*, paigaldus *(deploy)*, testimine ja väljalase *(release)*. Peamiseks märksõnaks on automatiseerimine, mille abil on testijatel ja arendajatel võimalik teostada oma igapäevaselt korratavaid ülesandeid lihtsalt ja kiiresti, ilma et kuluks päevast päeva aega ühe ja sama tegevuse jaoks. Automatiseerimine on kasulik pidevalt korratavate tegevuste

puhul, sest masin suudab teha rutiinseid tegevusi alati ühtemoodi, aga inimesed võivad tähelepanu hajumise või hooletuse tõttu teha vigu, mis võivad ka lihtsate tegevuste puhul kaasa tuua üleliigse ajakulu.

Pideva tarnimise strateegia püüab järgida ka DevOps liikumise [8] põhimõtet: „Meeskond peab hoidma kokku ja teadvustama endale, et kõik töötavad ühise eesmärgi nimel: luua kvaliteetset, usaldusväärset tarkvara, mis rahuldab kliendi vajadusi ja loob ärilist kasu *(bussiness benefit)*". Ühine on ka vastutus kvaliteetse tarkvara tarnimise eest, see ei ole ainult testijate ülesanne. Tegemist on pigem käitumusliku muutusega, mida soovitakse tarkvara arenduse meeskondades juurutada.

„Continuous Delivery" raamatu autorid on *lean* ja iteratiivse tarkvaraarenduse taustaga, seetõttu on paljud kirjeldatud tehnikad ja printsiibid mõeldud suurte agiilsete projektide jaoks, kuid need on kohandatavad ka kõigile teistele projektidele. Põhifookuseks on suhtluse suurendamine, tagamaks protsesside läbipaistvust ja tihedamat/kiiremat tagasisidet. See on positiivne iga projekti puhul, isegi kui tegemist pole iteratiivse tarkvaraarendusega.

Pideva tarnimise strateegia abil üritatakse leida moodus, kuidas panna kõik muutuvad osad kokku sobima: konfiguratsioonihaldus, automaattestimine, pidev integreerimine ja paigaldus, andmete, keskkondade ja väljalasete haldus. *Lean movement* õpetab optimeerima kogu tervikut. Selleks aga on vaja holistilist lähenemist, mis seoks üheks tervikuks kõik tarnimisprotsessi osad ja inimesed, kes nendega on seotud [2].

CD tuleks võtta kasutusele aegamisi, pidevalt midagi paremaks muutes ja tehes pidevalt retrospektiive. Pideva tarnimise strateegia puhul on oluline protsessi pidev arendamine. Kogu arendusmeeskonda kaasates tuleks tarnimise protsessis leida kitsaskohti ning väikeste realiseeritavate tükkidena protsessi pidevalt täiustada.

Kui projektis on kasutusel pideva tarnimise strateegia, siis ideaalis peaks olema võimalik igat uut tarkvaraversiooni, mis läbib testimise, paigaldada kohe ka toodangukeskkonda. Samas võib pidevaks tarnimiseks nimetada ka seda, kui toodangukeskkonda tarnitakse tihti. Põhjus, miks tarnida tihedamini – saada kasutajatelt tagasisidet. Kasutajatelt saadav tagasiside on väga oluline veendumaks, et toodetav tarkvara ei vasta mitte ainult tarkvaraspetsifikatsioonile, vaid ka kasutajate ootustele.

Tarkvaraarenduses on viimastel aastatel levinud lähenemine, mille kasutamisel paigaldatakse iga versioon, mis läbib kõik testimisetapid, kohe toodangukeskkonda. Seda lähenemist nimetatakse pidevaks paigaldamiseks (*continuous deployment*) [9]. Oma sisult on see üsna sarnane pideva tarnimise strateegiale ja ka terminid on väga sarnased. Seetõttu aetakse neid tihti segi. Põhiliseks erinevuseks on paigalduse tegemine toodangukeskkonda. Kui pideva paigalduse strateegia puhul on reegel, et iga versioon, mis läbib testimise paigaldatakse kohe toodangusse, siis pideva tarnimise strateegia puhul on olemas lihtsalt selline võimalus. Pideva paigalduse strateegiat järgivad arendusmeeskonnad tarnivad uusi versioone toodangukeskkonda mitu korda päevas.

Jez Humble toob artiklis *„Continuous Delivery: The Value Proposition"* välja pideva tarnimise strateegia kolm alustala. Nendeks on: konfiguratsioonihaldamine, agiilne testimine ja paigalduskonveier (*deployment pipeline*) [10]. Käesoleva töö järgnevates peatükkides 3 – 6 kirjeldatakse pideva tarnimise strateegia kasutuselevõtmise üldiseid põhimõtteid ja praktikaid, konfiguratsioonihaldamist, testimisstrateegia väljatöötamist ja paigalduskonveierit. Nendes peatükkides väljatoodud põhimõtete ja praktikate kirja panekul on lähtutud eelkõige Jez Humble'i ja David Farley poolt raamatus *„Continuous Delivery"*[2] kirja pandut.

3 Põhimõtted ja praktikad pideva tarnimise strateegia rakendamiseks

On olemas palju erinevaid tarkvaraarenduse metodoloogiaid, mis keskenduvad põhiliselt nõuete haldusele ja selle mõjule tarkvaraarendamisel. Paljud allikad selgitavad detailselt erinevaid lähenemisi tarkvara disaini, arenduse ja testimise osas, kuid tegelikult katavad need vaid väikse osa tarkvara kasuvoost *(value stream),* mis loob otsest väärtust kliendile, kes tarkvara tellis.

Pideva tarnimise strateegia keskendub ehitus- *(build)*, paigaldus- *(deploy)*, testimis- ja väljalaske *(release)* protsessidele, millest on varem küllaltki vähe kirjutatud. Vaadeldakse aspekte, mis juhtub siis, kui nõuded on kogutud, lahendused disainitud, arendatud ja testitud. Kuidas need tegevused omavahel kokku sobivad ning kuidas on omavahel koordineeritud, et arendusprotsess oleks võimalikult efektiivne ja usaldusväärne? Mida teha, et arendusmeeskonna koostöö oleks efektiivne? Millised tehnikad ja praktikad aitavad pideva tarnimise strateegiat ellu viia?

Üheks põhiliseks mustriks pideva tarnimise strateegia rakendamisel on paigalduskonveier *(deployment pipeline).* Paigalduskonveier on olemuselt rakenduse ehitus-, paigaldus-, testimis- ja väljalaskeprotsessi automatiseeritud implementatsioon ning see on iga projekti puhul pisut erinev sõltuvalt projekti kasuvoost. Paigalduskonveieri näidis on esitatud Joonisel 1.

Joonis 1. Paigalduskonveier [2].

Lühidalt öeldes on paigalduskonveier tarkvara tarnimisprotsessi elus mudel. Iga muudatus, mis tehakse tarkvara konfiguratsioonis, lähtekoodis, keskkondades või andmetes, läheb läbi paigalduskonveieri. Pärast muudatuse salvestamist versioonihaldusvahendisse ehitatakse rakendus ning käitatakse sellel automaatteste, et veenduda vastava versiooni toodangukõlbulikkuses. Iga test, mille toodangu kandidaat *(release candidate)* läbib, annab arendusmeeskonnale juurde kindlust, et just see kombinatsioon lähtekoodist,

konfiguratsioonist, keskkonnast ja andmetest töötab õigesti. Kui toodangu kandidaat läbib kõik testid, võib vastava versiooni kliendile üle anda.

Paigalduskonveier on oluline kolmel põhjusel:

- Muudab ehitus-, paigaldus-, testimis- ja tarnimisprotsessi kogu tarkvaraarendusmeeskonna jaoks läbipaistvaks, parandades seeläbi meeskonnaliikmete omavahelist suhtlust.
- Parandab tagasisidet süsteemi kohta, aidates probleeme varakult tuvastada ja seeläbi ka varem parandada.
- Võimaldab arendusmeeskonnal paigaldada ja tarnida suvalise tarkvaraversiooni ükskõik millisesse keskkonda läbi täielikult automatiseeritud protsessi.

Paigalduskonveieri praktikaid ja põhimõtteid kirjeldatakse põhjalikumalt peatükis 6.

3.1 Lahendused, mida tarkvara tarneprotsessis vältida

Tarkvara tarnimine on protsess, kus on võimalik teha palju vigu. Kui ükskõik milline samm tarneprotsessis läheb valesti, ei pruugi rakendus korrektselt töötada. Samas ei pruugi olla üheselt selge, kus viga peitub või milline samm ebaõnnestus. Järgnevalt on välja toodud mõned tihti kasutusel olevad lahendused, mis välistavad tarneprotsessi usaldusväärsuse, kuid samas on saanud normiks tarkvaratööstuses.

Lahendused, mida tuleks vältida:

- Tarkvara käsitsi paigaldamine.
- Tarkvara paigaldamine toodangulaadsesse keskkonda alles pärast arenduse lõppu.
- Toodangu keskkondade konfiguratsiooni haldamine käsitsi.

Mida saaks teha paremini?

Pideva tarnimise strateegia rakendamisel on eesmärk muuta tarneprotsess igavaks. Tarneprotsess võiks ja peaks olema madala riskiga, tihti esinev, odav, kiire ja ettearvatav protsess.

Pideva tarnimise strateegia rakendamisel on oluline osa paigalduskonveieri kasutusele võtmisel, kasutades selleks kõrgel tasemel automatiseerimist nii testimiseks, kui ka paigalduseks ja kõikehõlmavat konfiguratsiooni haldust. Eesmärk on muuta tarkvara versiooni väljalase võimalikult lihtsaks, ühe nupuvajutusega tehtavaks tegevuseks, mis toimuks ühtemoodi olenemata, millisesse keskkonda rakendust paigaldada soovitakse.

3.2 Kuidas saavutada eesmärk

Iga tarkvaraarendusmeeskonna eesmärk on tarnida kasutajatele kasulikku ja töötavat tarkvara nii kiiresti kui võimalik. Selleks, et täita neid eesmärke, st toota lühikese tsükliajaga kvaliteetset tarkvara, peavad tarkvara väljalasked olema automatiseeritud ja sagedased. Järgnevalt püüame eelöeldut põhjendada.

- **Automatiseeritud.** Kui ehitus-, paigaldus-, testimis- ja väljalaskeprotsess ei ole automatiseeritud, ei ole see ka korratav. Iga kord, on see erinev, sest lähtekood, keskkond, süsteemi konfiguratsioon ja väljalaske protsess on muutunud. Kuna kõik sammud sooritatakse käsitsi, on lihtne eksida ja hiljem pole võimalik välja selgitada, mida täpselt tehti. Seega pole võimalik omada täielikku kontrolli väljalaskeprotsessi üle ning ei saa rääkida selle kõrgest kvaliteedist.
- **Sagedased.** Kui väljalaskeid teha tihti, on kahe tarnitud versiooni erinevus väike. See vähendab oluliselt väljalaskega seotud riske ja lihtsustab vanale versioonile tagasiminekut. Sagedased väljalasked võimaldavad kiirema tagasiside saamist tarkvara muudatuste kohta. Tagasiside saamine on üks olulisemaid aspekte pideva tarnimise strateegia rakendamisel. Tagasiside abil on võimalik veenduda, et loodud tarkvara vastab kliendi soovidele ja ootustele.

Automatiseeritud ja sagedaste tarnete puhul on oluline roll tagasisidel. Et tagasisidest oleks võimalikult palju kasu, peab see vastama kolmele järgnevale kriteeriumile:

- **Iga muudatusega peab kaasnema tagasisideprotsess.** Tagasisideprotsess hõlmab iga muudatuse testimist nii põhjalikult kui võimalik. Testimiseks tuleks kasutada nii automaatseid, kui ka traditsioonilisi testimismeetodeid. Testid võivad erinevate süsteemide puhul olla väga erinevad, aga peaksid sisaldama järgnevaid kontrolle: koodi töötamine, ühiktestide edukas läbimine, kvaliteedi kriteeriumite täitmine,

funktsionaalsete vastuvõtutestide läbimine, mittefunktsionaalsete testide läbimine, uurivtestimise (*exploratory testing*) edukas läbimine ja kliendi rahulolu rakenduse esitlusega.

- **Tagasiside andmine/saamine nii kiiresti kui võimalik.** Kiire tagasiside tagab automatiseeritud protsess, mille puhul on ainsaks takistuseks kasutatava riistvara võimsus.

- **Tarnete eest vastutav meeskond peab saama tagasisidet ja tegutsema sellele vastavalt.** On ilmne, et kõik, kes on seotud tarkvara arendusprotsessiga, on osalised ka tagasiside protsessis, sealhulgas arendajad, testijad, andmebaasi administraatorid, infrastruktuuri spetsialistid ja juhid. Tagasisidet antakse pidevalt ja samuti peab oskama seda vastu võtta ning vastavalt sellele tegutseda. Tagasisidest pole mingit kasu, kui ei tegutseta vastavalt saadud infole.

Sellist protsessi võib pidada idealistlikuks, kuid see toimib ja sobib nii suurtele kui ka väikestele meeskondadele. Selle tõestamiseks on vaja seda proovida, st võtta kasutusele praktikad, mis meeskonna tööd tõhustavad ja jätta kõrvale need, mis antud meeskonna puhul kasutamiseks ei sobi. CD kasutavad arendamisel näiteks flickr, facebook [11] ja LMAX [5].

3.3 Pideva tarnimise strateegia rakendamisest saadav kasu

Pideva tarnimise strateegiat rakendades on võimalik luua tarneprotsess, mis on korratav, usaldusväärne ja etteaimatav. Kokkuvõttes võimaldab see vähendada tsükliaega ja seeläbi jõuavad uus funktsionaalsus ja veaparandused kasutajateni kiiremini. Samas on ka mitmeid teisi kasulikke omadusi:

- meeskonna liikmete vastutus suureneb ja seeläbi paraneb nende töö kvaliteet ning rakenduse üldine kvaliteet;

- vigade hulk väheneb;

- stressi hulk väheneb;

- paigalduste tegemine muutub paindlikumaks.

Paigalduse tegemine kõikidesse keskkondadesse peaks toimuma alati ühtemoodi. Nii saab olla kindel, et paigaldusmehhanism töötab õigesti iga kord kui versioon mingisse keskkonda paigaldatakse.

3.4 Toodangu kandidaat

Traditsioonilises lähenemises tarkvaraarendusele nimetatakse toodangu kandidaadiks rakenduse versiooni, mis on läbinud testimise ja millest võib saada järgmine väljalastav versioon. Joonisel 2 on näha, et „toodangu kandidaat“ on testimisprotsessi eraldi samm.

Joonis 2. Traditsiooniline lähenemine toodangu kandidaatidele [12].

Toodangu kandidaadiks nimetamine toimub arendusprotsessi lõpus, kui rakenduse versioon on läbinud alfa ja beeta testimise ning versioon on tunnistatud piisavalt kvaliteetseks ning funktsionaalsus on valmis.

Pideva tarnimise strateegia puhul on lähenemine toodangu kandidaadile veidi erinev. Keskkonnas, kus kasutatakse automaatset ehitamist ja paigaldamist ning millele järgneb laialdane automaattestimise tsükkel, ei ole mõtet jätta testimist projekti/iteratsiooni lõppu. Vastupidiselt, venitades testimise alustamisega ning tehes seda alles pärast arendusfaasi lõppu, on sellel kvaliteedile pigem negatiivne mõju. On parem, kui vead leitakse ja parandatakse võimalikult kiiresti, vastasel juhul on nende parandamine oluliselt kallim. Arendajad mäletavad tööülesande sisu ja realisatsiooni paremini, kui vead raporteeritakse võimalikult kiiresti peale muudatuse sulgemist. Kui vigu hakatakse parandama oluliselt hiljem, võib olla vahepeal funktsionaalsus muutunud ja arendaja ei pruugi mäletada, mida ta tegi. Kui testimine jäetakse arendustsükli lõppu, võib see tähendada ka seda, et leitud vigade parandamiseks pole aega või parandada jõuab ainult väikse osa vigadest. Pideva tarnimise strateegia puhul on eesmärk leida ja parandada vead nii kiiresti kui võimalik. Selle saavutamiseks peavad arendajad ja testijad tihedalt koostööd tegema ning ühes tempos püsima. Testijatele ei tohi koguneda suurt hulka järelkontrolli ootel olevaid tööülesandeid, vaid nad peaksid jõudma võimalikult kiiresti peale funktsionaalsuse arenduse lõppu kõik uued tööülesanded üle vaadata.

Pideva tarnimise strateegia järgi on toodangu kandidaat iga muudatus, mis kompileeritakse pideva integratsiooni *(continuous integration,* edaspidi ka CI) keskkonnas edukalt. CD põhimõtete kohaselt peab iga uus versioonihaldusesse lisatud muudatus läbima kõik automaattestid. Pideva integratsiooni keskkonna eesmärgiks on see eeldus ümber lükata ja näidata, et käesolev toodangu kandidaat ei ole sobiv toodangukeskkonda paigaldamiseks.

3.5 Tarkvara üleandmise/tarnimise põhimõtted

Jez Humble ja David Farley toovad oma raamatus „Continuous Delivery" välja 8 tarkvara tarnimise põhimõtet, ilma milleta nad ei kujuta ette, et tarneprotsess võiks olla efektiivne [2]:

- **Luua korratav ja usaldusväärne tarkvara väljalaske protsess.** Tarkvara väljalase peaks olema lihtne. See peaks olema lihtne juba sellepärast, et tarkvara arenduse jooksul tehakse palju väljalaskeid ning seetõttu tuleks protsess muuta sama lihtsaks nagu nupuvajutus. Tarkvara väljalaske korratavuse ja usaldusväärsuse tagavad kaks järgnevat põhimõtet: maksimaalne automatiseerimine ja kõige, mida on vaja ehitada, paigaldada, testida ja tarnida, versioonihalduses hoidmine.

- **Automatiseerida niipalju kui võimalik**. Kõike pole võimalik kunagi automatiseerida. Näiteks uuriva testimise teostamiseks on vaja kogenud testijaid ja juba töötava tarkvara esitlust kliendile ei saa teha arvutid. Samas on palju tegevusi, mida on võimalik automatiseerida. Kõik, mis ei vaja otsest inimese sekkumist ja otsuste langetamist, tuleks automatiseerida. Automatiseerida tuleks tarkvara paigaldus ja tegelikult kogu väljalaskeprotsess. Samuti on võimalik automatiseerida vastuvõtutestid *(acceptance tests)*, andmebaasi uuendamine ja keskkondade konfiguratsioon. Automatiseerimine on ka üheks eelduseks paigalduskonveieri rakendamiseks. Kogu automatiseerimist pole oluline teha korraga, vaid tuleb leida esmased pudelikaelad erinevates protsessides ja seejärel neid järjest kõrvaldada automatiseerimise teel.

- **Hoida kõike versioonihalduses.** Kõike, mida on vaja rakenduse puhul ehitada, paigaldada, testida ja tarnida, tuleb hoida versioonihalduses. See sisaldab nõuete dokumente, testiskripte, automaattestide testjuhtumeid, keskkondade

konfiguratsiooniskripte, paigaldusskripte, andmebaaside loomis-, uuendus- ja initsialiseerimisskripte, tarkvarateeke jne. Samuti võiks olla selgesti nähtav, milline rakenduse versioon on projekti erinevatesse keskkondadesse paigaldatud. Versioonihaldus on väga oluline ülevaate tagamiseks ning uute inimeste projekti kaasamiseks.

- **Probleemide ennetamiseks tuleks veaohtlikke tegevusi teha tihedamini.** See on väga üldine põhimõte ja tegemist on tarkvara üleandmise protsessi olulise heuristikaga. Kui testimine on arenduse puhul kujunenud väga valulikuks protsessiks, mida tehakse vahetult enne väljalaset, siis tuleks tagada, et see ei toimuks arendustsükli lõpus, vaid testima peaks pidevalt kogu arendustsükli jooksul. Kui aga tarkvara väljalase on osutunud valulikuks protsessiks, tuleks proovida tarnida pidevalt, kasvõi pärast iga muudatuse salvestamist versioonihaldusesse. Kui pole võimalik tarnida toodangukeskkonda, võiks pidevalt tarnida toodangulaadsesse keskkonda.

- **Kvaliteedi tagamine.** Mida varem vead avastada, seda odavam on neid parandada. Kõige odavam on vigu parandada siis, kui need ei jõua testimiskeskkondadesse. On olemas erinevaid tehnikaid, nagu pidev integreerimine, automaattestimine, automatiseeritud paigaldus jne, mis võimaldavad vigu tarkvara arendusprotsessis võimalikult vara avastada (st rakendatakse eelmist põhimõtet „*Bring the pain forward*“). Järgmine samm kvaliteedi tagamisel on vigade parandamine, mida tuleks teha kohe, peale avastamist. Lisaks tuleb jälgida kahte olulist mõtet. Esiteks, testimine pole eraldi faas, mis tuleb pärast arendust. Kui testimine jätta arendusprotsessi lõppu, ei jää enam aega vigade parandamiseks. Teiseks, kvaliteedi eest ei vastuta ainult testijad, vaid kogu arendusmeeskond.

- **Tehtud/valmis tähendab, et on kliendile üle antud**. Kui arendaja saab kasutusloo realiseeritud ja ütleb, et see on valmis, siis tegelikult on ainult arendus lõpetatud. Valmis saab funktsionaalsuse kohta öelda alles siis, kui antud arendus jõuab toodangukeskkonda. Ideaalis tähendab valmis seda, et arendus on jõudnud toodangukeskkonda ja kasutajad kasutavad seda. Suuremates projektides pole enamasti võimalik tarkvarasse muudatuse tegemine ühe inimese poolt, selleks on

vaja kogu arendusmeeskonna koostööd. Analüütikud, arendajad ja testijad peavad kõik andma oma panuse, et muudatus jõuaks kasutajateni.

- **Kogu meeskond vastutab eduka tarneprotsessi eest.** Ideaalis töötavad arendusmeeskonna kõik liikmed ühiste eesmärkide nimel ning tegema koostööd, et ühiselt nende eesmärkide poole püüelda. Lõppkokkuvõttes ollakse edukas või põrutakse kogu meeskonnaga, mitte individuaalselt. Samas on paljudes arendusmeeskondades välja kujunenud olukord, kus arendajad annavad töö üle testijatele mõttega, et nende töö on nüüd tehtud. Ning testijad kuhjavad tööd kokku „üleandmise ootel" olevasse kuhja, omamata plaani tarne tegemiseks. Kõik meeskonnaliikmed peaksid omavahel suhtlema, omama ülevaadet rakenduse hetkeseisust ning teadma, millal on plaanis teha tarne.

- **Tarnimisprotsessi pidev täiendamine.** Rakenduse esimese versiooni üleandmine kliendile on lihtsalt selle elutsükli esimene etapp. Rakendus täieneb pidevalt ja seda tarnitakse korduvalt. Seejuures on oluline, et ka tarneprotsessi pidevalt täiustataks. Selleks peaks aeg-ajalt kogu arendustiim kokku tulema ja pidama tarneprotsessi tagasivaatekoosolekuid, kus arutatakse, mis on vahepeal läinud hästi ja mis halvasti. Jagatakse ideid ja vahetatakse mõtteid, kuidas asju paremaks muuta. Igale uuele ideele määratakse omanik, kes vastutab, et see ellu viiakse. Järgmisel koosolekul uuritakse, kuidas uue idee elluviimine õnnestus. See on tuntud ka kui *deming cycle* [13]: planeeri, vii läbi, õpi, tegutse (*plan, do, study, act*). Oluline on, et kõik meeskonnaliikmed oleksid kaasatud, et optimeerida kogu tarneprotsessi, mitte vaid üksikuid osasid.

3.6 Pidev integratsioon *(Continuous Integration)*

Pideva tarnimise strateegia aluseks on pidev integratsioon (CI) [2]. Pideva tarnimise strateegia kasutuselevõtmisel on esimeseks sammuks CI kasutusele võtmine.

CI eeldab rakenduse ehitamist ja automaattestide käivitamist iga kord, kui versioonihaldusesse salvestatakse uus muudatus. Kui rakenduse ehitamine või sellel käivitatud testid ebaõnnestuvad, on muudatuse teinud arendaja kohustuseks koheselt probleem kõrvaldada. Eesmärgiks on hoida rakendus kogu aeg töötavana.

Rakenduse ehitamine (*build*) on protsess, mis koosneb koodi kompileerimisest, testimisest, inspekteerimisest ja paigaldusest [14].

4 Konfiguratsiooni haldamine

Konfiguratsioonihalduse strateegia fikseerib, kuidas hallatakse muudatusi, mis projekti jooksul toimuvad. Samuti tuleb need muudatused, mis süsteemis ja rakendustes tehtud on, kirja panna. Konfiguratsioonihalduse strateegia kirjeldab, kuidas projekti meeskond omavahel suhtleb. Hea strateegia puhul vastab projekt järgnevatele tingimustele[2]:

- Kõik keskkonnad on reprodutseeritavad, kaasa arvatud operatsioonisüsteem, selle võrgu konfiguratsioon, paigaldatud tarkvara ja rakendused ning nende konfiguratsioon.
- Iga konfigureeritavat üksust on võimalik muuta igas keskkonnas eraldi või kõigis korraga.
- Iga tehtud muudatust on võimalik näha ja kiiresti tuvastada, mis muudatus see oli ja kes ning millal selle tegi.
- Iga meeskonnaliige omab ligipääsu talle vajalikule informatsioonile ja saab teha talle vajalikke muudatusi.

Esimeseks sammuks konfiguratsioonihalduse strateegia väljatöötamisel on versioonihaldusevahendi kasutusele võtmine. Kuna antud töö keskendub pigem testimist puudutavale pideva tarnimise strateegia osale, siis kirjeldatakse konfiguratsioonihaldusest ainult versioonihalduse osa, mis on kõige otsesemalt seotud testimisega.

4.1 Versioonihaldus

Versioonihaldusvahend on süsteem, mis võimaldab hoida alles ühe ja sama faili erinevaid versioone. Kui faili muudetakse, on endiselt kättesaadav see versioon failist, mis oli kasutusel enne muudatuste sisse viimist. Samuti salvestatakse versioonihalduses iga faili külge metaandmeid – infot salvestatud andmete kohta. Versioonihaldusvahendit kasutatakse tarkvaraarenduses tihti ka erinevate osapoolte vaheliseks suhtluseks.

Versioonihalduse kasutamine võimaldab[2]:

- Saada teada, mida sisaldab mingi kindel tarkvara versioon ning kuidas reprodutseerida toodangu keskkonnas olevat tarkvara seisu.

- Saada teada, kes, millal ja mida muutnud on. See pole vajalik mitte ainult siis, kui midagi on katki, vaid on oluline ka üldise ülevaate saamiseks tarkvara arendamisel.

Versioonihalduse efektiivseks kasutamiseks tuleks järgida järgnevaid põhimõtteid [2]:

- **Hoida kõike versioonihalduses.** Versioonihalduses tuleks hoida kõiki artifakte, mis on seotud tarkvara arendamisega. Arendajad peaksid hoidma versioonihalduses nii lähtekoodi, teste, andmebaasiskripte, ehitus- ja paigaldusskripte, rakenduse dokumentatsiooni ja konfiguratsioonifaile, kasutusel olevat kompilaatorit ja tööriistu ja muud taolist. See on oluline, et lihtsustada uute meeskonnaliikmete töö alustamist. Analüütikud peaksid hoidma versioonihalduses nõuete dokumente ja andmebaasi arhitektuuri. Testijad peaksid hoidma versioonihalduses testjuhtumeid, -skripte ja -plaane; projektijuhid aga projektiplaani, iteratsiooniplaani ja muid vajalikke dokumente. Kokkuvõttes peaks iga meeskonnaliige hoidma versioonihalduses kõiki dokumente või faile, mis on projektiga seotud.
- **Teha sissekandeid regulaarselt.** Tihti on tarkvaraarenduses nii, et kui arendaja töötab mingi keerulisema komponendi või ülesande kallal, siis ta ei soovi koodi enne versioonihaldusesse panna, kui ta on antud komponendi või ülesande valmis saanud. Arendaja soovib veenduda, et kood on korrektne ja ei mõju ülejäänud funktsionaalsusele halvasti. See aga võib viia olukorda, kus arendaja ei salvesta oma tööd teistele nähtavaks (*commit*) mitu päeva või isegi terve nädal. Sellisel juhul on aga põhiharuga ühendamine üsnagi keerukas. Kui teha sissekandeid regulaarselt, on koodis tehtud muudatused teisele kiiresti kättesaadavad ja on selgesti näha, et muudatus ei lõhkunud rakendust ning ühendamised on väiksed ja lihtsasti hallatavad. Minimaalselt peaks sissekandeid tegema vähemalt korra päevas, aga oleks parem, kui neid tehtaks tihedamini.
- **Kasutada sisukaid *commit* sõnumeid.** Iga versioonihaldussüsteem võimaldab *commit*'ile lisada kirjeldavat kommentaari muudatuse sisu kohta, kuid samas ei ole see enamasti kohustuslik ja seetõttu ei kasutata seda alati. Kui aga jätta *commit* sõnum kirjutamata, on hiljem üsna keeruline tuvastada, mida vastav muudatus sisaldas ja miks see lahendati just nii. Isegi kui kommentaar on kirjutatud, aga see on stiilis „parandasin vea", siis pole sellest endiselt kasu. Võimalus korral võiks

commit sõnum sisaldada infot selle kohta, mida ja miks tehti, et teistel oleks hiljem võimalikult lihtne tehtud lahendust parandada või täiendada. Samuti võiks *commit* olla seotud kasutusel oleva projektihaldusvahendi tööülesandega, et oleks võimalik tuvastada, mille kallal arendaja parasjagu töötas.

Eelnevad punktid toovad välja olulised tegevused, mida peaksid arendajad igapäevaselt teostama. Samas on nende rakendamisest tulenev kasu oluline kogu meeskonnale.

5 Testimisstrateegia väljatöötamine

Paljudes projektides loodetakse ainult manuaalsetele vastuvõtutestidele, et verifitseerida tarkvara vastavust selle funktsionaalsetele ja mittefunktsionaalsetele nõuetele. Isegi kui kasutatakse automaatteste, on need tavaliselt kehvasti hallatud ja aegunud ning täiendavalt on vaja ikkagi kasutada ka manuaalset testimist.

Üks W. Edwards Deming'u neljateistkümnest punktist on: „Lõpeta sõltumine massilisest ülevaatamisest, et saavutada kvaliteeti. Paranda protsessi ja ehita kvaliteet tootese algusest peale sisse."[15]. Testimine on tegevus, mis hõlmab kogu meeskonda ja mida peaks tegema pidevalt projekti algusest alates. Kvaliteedi tagamiseks tuleb kirjutada mitmel tasemel automaatteste (ühiktestid, komponentestid ja vastuvõtutestid) ning neid käitada osana paigalduskonveierist. Manuaalsel testimisel on samuti oluline roll kvaliteedi tagamisel. Kliendile tuleb pidevalt näidata töötavat tarkvara (*showcases*) ja läbi viia kasutatavuse (*usability*) ning uurivat (*exploratory*) testimist. Kvaliteedi sisseehitamiseks tuleb pidevalt täiustada projekti automaatse testimise strateegiat.

Ideaalses projektis kirjutavad testijad automaatteste projekti algusest saadik. Selleks suhtlevad testijad arendajate ja kasutajatega/kliendiga. Need testid tuleks kirjutada enne, kui arendaja alustab vastava kasutusloo arendamist, mida see test testib. Ühiselt moodustavad need testid käitatava spetsifikatsiooni, mis edukal läbimisel näitab, et kliendi poolt soovitud funktsionaalsus on valmis ja töötab õigesti. Kokkupandud testikomplekti käitatakse pideva integratsiooni serveris peale igat rakendusse tehtud muudatust. Seega on tegemist kogumikuga regressioontestimise jaoks. Pidev testide käitamine tagab kliendi nõuete täitmist takistavate probleemide leidmise varajases staadiumis, kui nende parandamine on odavam.

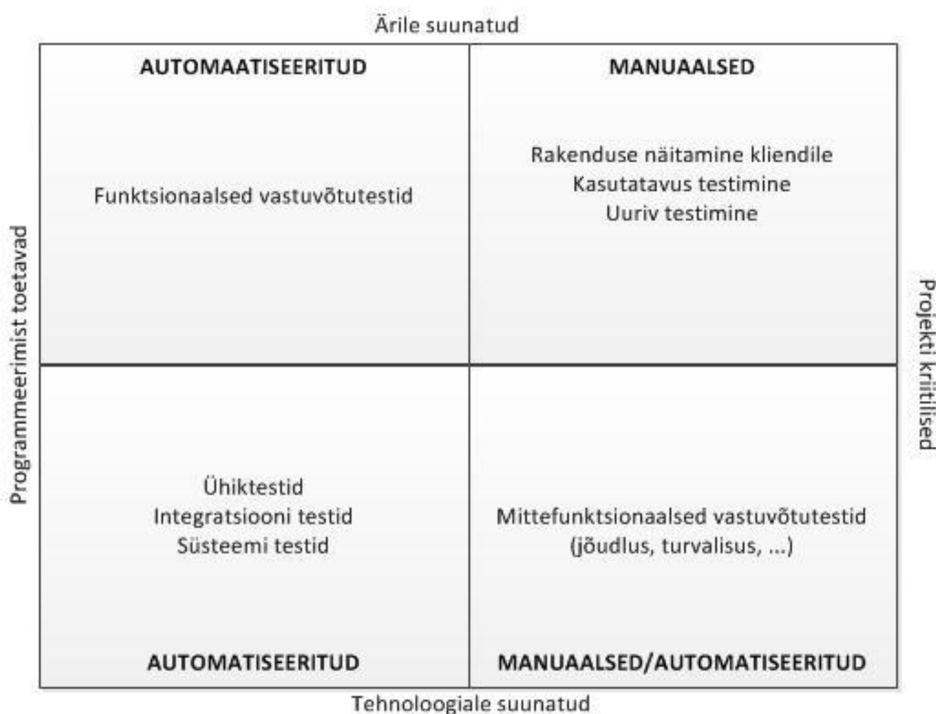
Sellise ideaalpildi saavutamine projektides on täielikult võimalik, kui vastav distsipliin võetakse kasutusele projekti algusfaasis. Juba käimasolevas projektis sama olukorra saavutamine on aga keerulisem, kuna automaattestidega katvuse kõrge taseme saavutamine nõuab palju aega ja hoolikat planeerimist, et tagada arenduse jätkumist samal ajal automaattestimise kasutuselevõtuga.

Testimise strateegia kavandamine on protsess, mille käigus tuvastatakse ja pannakse tähtsuse järjekorda projekti riskid ning otsustatakse, mida teha, et neid vähendada. Heal

testimisstrateegial on projektile mitmekülgne positiivne mõju. Testimine suurendab kindlustunnet, et tarkvara töötab nii nagu peab. See tähendab, et tarkvaras on vähem vigu, väheneb vajadus kasutajatoele ja paraneb reputatsioon. Laialdane automaattestide komplekt pakub hetkeseisule vastavat dokumentatsiooni käitatava spetsifikatsiooni näol, mis ei vasta ainult tingimustele, kuidas tarkvara peaks käituma, vaid näitab ära, kuidas tarkvara tegelikult käitub.

5.1 Testide tüübid

Kasutusel on väga erinevaid viise testimiseks. Brian Marick esitas skeemi (Joonis 3) [16], mida käesoleval ajal kasutatakse laialdaselt erinevat tüüpi testidest mudeli loomisel, mis peaksid olema kasutusel, et tagada kõrge kvaliteediga tarkvara tarnimine kliendile. Joonisel on kujutatud agiilse testimise sektoreid (kvadrante) [2,17]. Iga kvadrant kajastab erinevaid põhjuseid, miks tarkvara testitakse. Maatriksi ühel teljel on testid, mis toetavad programmeerimist ja testid, mis kritiseerivad projekti. Teisel teljel jaotatakse need aga ärile suunatud ja tehnoloogiale suunatud testideks.



Joonis 3. Testimise kvadrandid [16].

Joonisel vasakule poole jäävad sektorid sisaldavad teste, mis toetavad meeskonda tarkvaraarenduse käigus. Nende sektorite eesmärk on võimaldada kiiret tagasisidet

meeskonnale rakenduse hetkeolukorra kohta. Selleks kasutatakse peamiselt automaatteste, mida käitatakse peale iga muudatuse tegemist koodis. Eelkõige on need testid suunatud rakenduse funktsionaalsusele ning mõeldud ennetamaks refaktoreerimise vajadust.

Paremale poole jäävad sektorid sisaldavad teste, mis kritiseerivad projekti. Antud juhul ei kasutata sõna „kritiseerima" negatiivses tähenduses, vaid see sisaldab nii protsessi heade kohtade välja toomist, kui ka ettepanekuid parandamiseks.

Pannes kokku mõlemal teljel olevad testide eristused saadakse järgnevad kvadrandid:

- **Ärile suunatud testid, mis toetavad arendusprotsessi** (*Bussiness-facing tests that support the development process*). Sellesse kvadranti kuuluvaid teste võib nimetada funktsionaalseteks vastuvõtutestideks. Nende abil kontrollitakse kliendile ärilise kasu ehk väärtuse loomist. Samas aitavad need arendajatel paremini rakendust mõista. Ärile suunatud teste kirjutatakse kliendi abiga, et need vastaksid täpselt kliendi soovidele ja vajadustele. Enamus sellese sektorisse kuuluvaid teste tuleks ka automatiseerida, et saada kiiret tagasisidet rakenduse seisukorra kohta.
- **Tehnoloogiale suunatud testid, mis toetavad arendusprotsessi** *(Technology-facing tests that support the development process).* Sellesse kvadranti kuuluvad testid esindavad ühte agiilse tarkvaraarenduse põhilist praktikat – testidest juhitud arendust (*test-driven development*). Ühiktestid, komponenttestid ja süsteemitestid on kõik testid, mida tavaliselt kirjutavad ja haldavad arendajad. Nende testide kirjutamine aitab arendajatel oma koodi paremini disainida ning annavad arendajale kindlustunde kirjutatud koodi arhitektuuri ja disaini korrektsuses. Tegemist on automaattestidega, mis on kirjutatud samas programmeerimiskeeles, milles testitav rakenduski. Nendega testitakse koodi sisemist kvaliteeti ja tavaliselt selles osas kliendi arvamust ei küsita, vaid see on puhtalt programmeerijate vastutada.
- **Ärile suunatud testid, mis kritiseerivad projekti** *(Bussiness-facing tests that critique the project).* Ärile suunatud ja projekti kritiseerivate testide tegemiseks tuleb teha tegevusi, mida lõppkasutajad rakendusega tegema hakkavad. See on võimalik läbi manuaalse testimise, mida saavad teha

ainult inimesed. Antud juhul ei kontrollita mitte ainult rakenduse vastavust spetsifikatsioonile, vaid ka spetsifikatsiooni vastavust reaalsetele vajadustele. Tihti tulevad toodangukeskkonnast välja vead just nendes situatsioonides, mida keegi kunagi varem läbi proovinud pole. Siin on abiks uurivtestimine, klienditestimine ja rakenduse läbivaatamine koos kliendiga, mis võimaldavad läbi käia ja testida palju erinevaid stsenaariume. Kõigi nende stsenaariumite katmine automaattestidega poleks aga mõeldav, sest see on liialt aeganõudev ja kulukas. Sellesse kvadranti kuuluvad ka kasutatavustestid, mida tuleb samuti manuaalselt sooritada, kuna arvutid veel ei oska hinnata rakenduse kasutusmugavust ning meeldivust.

- **Tehnoloogiale suunatud testid, mis kritiseerivad projekti** *(Technology-facing tests that critique the project).* Antud tehnoloogiale suunatud testid on mõeldud hindama rakenduse selliseid omadusi nagu jõudlus, kättesaadavus ja turvalisus. Sellesse kvadranti kuuluvaid teste võib nimetada ka mittefunktsionaalseteks vastuvõtutestideks. Nende testide läbiviimiseks võib olla vaja kasutada spetsiaalset keskkonda, töövahendeid ning see nõuab ka nende läbiviijatelt vastavaid ekspertteadmisi. Olenevalt projektist võib mittefunktsionaalsete nõuete testimise tähtsus olla väga erinev. Näiteks veebipoe puhul on väga oluline kiirus. Kui iga lehe laadimine võtab aega rohkem kui minuti, siis on selge, et klient valib pigem sellise veebipoe, kus reaktsiooniaeg on kiirem. Kuid kindlasti on iga projekti puhul oluline vähemalt mingil määral mittefunktsionaalsete nõuete testimine.

Enamiku tarkvaraprojektide puhul on vajalikud kõik neli eelpool kirjeldatud testimise kategooriat veendumaks, et tarnitav tarkvara loob õiget väärtust. Iga arenduses olev kasutuslugu ei pruugi vajada näiteks turvalisuse testimist, kuid selle vahelejätmine pole mõeldav põhjusel, et ei tuldud lihtsalt selle peale. Programmeerimist toetavad testid aitavad määratleda, millal mingi funktsionaalsuse kohta saab öelda „valmis“ ning projektikriitilised testid aitavad kindlustada, et puudujääk funktsionaalsuses leitakse üles. Kõigi nelja kvadrandi koos kasutamisel on võimalik kogu meeskonnal määratleda, millal iga kasutuslugu täidab kliendi poolt seatud kriteeriume funktsionaalsusele ja kvaliteedile.

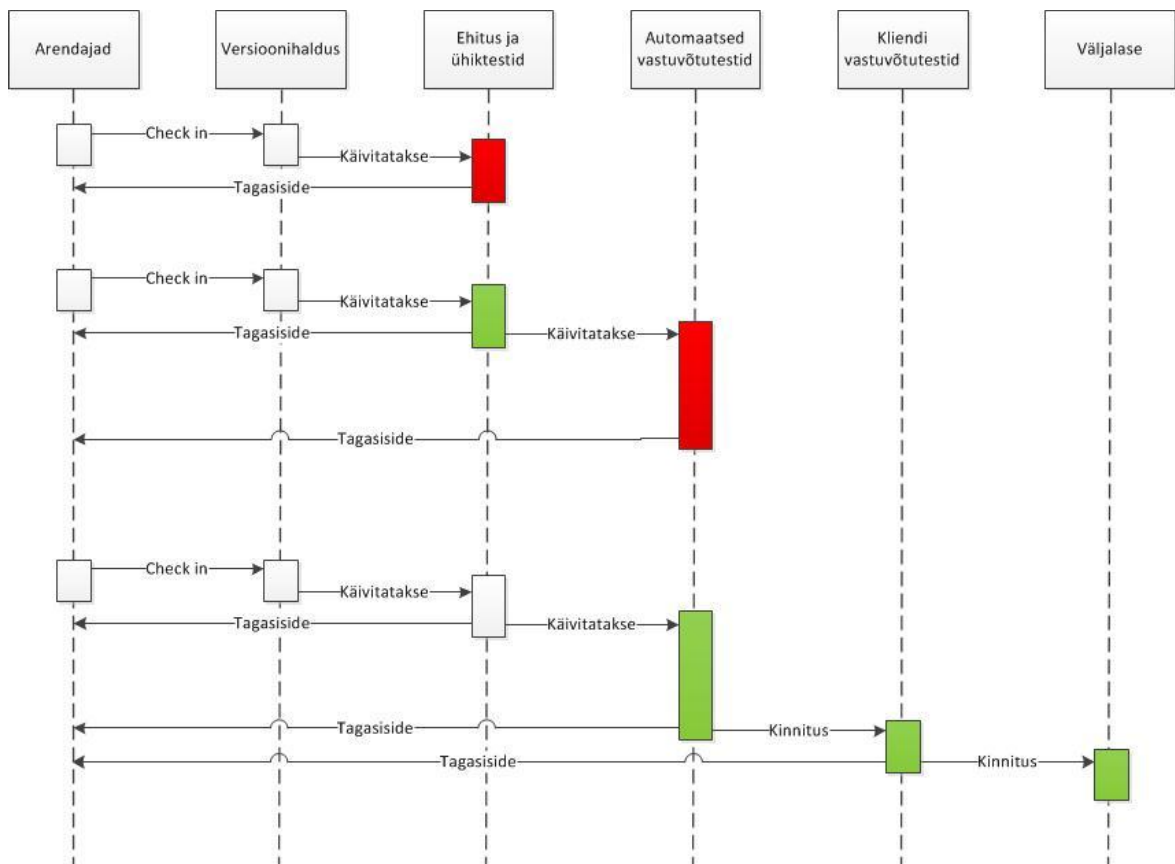
6 Paigalduskonveier

Terminit „paigalduskonveier“ (*deployment pipeline*) mainis esimesena Sam Newman [18].

Tarkvaraarendusprotsess algab traditsiooniliselt tellija poolt arendussoovi esitamisega ja lõppeb lahenduse jõudmisega kasutajateni. Paigalduskonveier on tarkvara arendusprotsessi automatiseeritud mudel, mis algab koodi versioonihaldusesse jõudmisega ja lõppeb tarkvara üleandmisega kliendile. Iga muudatus, mis tarkavaras tehakse, läbib keeruka protsessi enne kui see kliendile üle antakse. See protsess hõlmab endas koodi kokku ehitamist, mis seejärel läbib rea erinevaid testimise etappe ja paigaldust erinevatesse keskkondadesse.

Üks võimalus mõista, mis on paigalduskonveier ja kuidas muudatused seda läbivad, on kujutada seda järgnevusskeemil (*sequence diagram*) nagu on näha ka Joonisel 4. Idee paigalduskonveieri selliseks kujutamiseks on saadud Christopher Read'i ettekandest „CI, Pipelines and Deployment“ [19].

Joonisel kujutatud paigalduskonveieri sisendiks on iga uus sissekanne versioonihaldusesse. Iga muudatuse peale tekitatakse uus rakenduse eksemplar, mis peab läbima rea teste, et näidata selle versiooni sobivust toodangukeskkonda paigaldamiseks. Kasutatav protsess koosneb erinevatest testimise etappidest, millest iga etapp hindab versiooni sobivust erineva nurga alt ja mis algatatakse versioonihaldusesse uue muudatuse sissekandmisel. Idee poolest on tegu CI protsessiga, kuid see on jagatud väiksemateks osadeks.



Joonis 4. Muudatuste liikumine läbi paigalduskonveieri [20].

Mida rohkem teste protsessi käigus läbitakse, seda enam kasvab kindlustunne, et tegemist on toodanguks sobiva versiooniga. Seega, mida kaugemale jõuab muudatus paigalduskonveieris, seda enam ollakse nõus sellele ressursse kulutama ning seda sarnasemaks muutuvad keskkonnad, mida see läbib, toodangu keskkonnale. Eesmärk on elimineerida mittesobivad toodangu kandidaadid võimalikult protsessi alguses. Ning versiooni mittesobivusel anda meeskonnale võimalikult varakult tagasisidet leitud vigade kohta. Seetõttu ei tohi ühtegi muudatust, mis kukub mingis paigalduskonveieri etapis läbi, lubada edasi protsessi järgmisse etappi.

Sellise protsessi rakendamise olulisteks tagajärgedeks on:

1) Toodangukeskkonda ei jõua versioonid, mis pole täielikult testitud. Välditakse regressiooni vigade esinemist ja seda eriti juhul, kui on vaja teha kiireid parandusi toodangukeskkonnas, kuna kõik versioonid lähevad ühtemoodi läbi paigalduskonveieri.

2) Kui erinevatesse keskkondadesse paigaldamine on automatiseeritud, on protsess lihtsamini korratav ning vajadusel on võimalik seda tihedamini läbi viia. Samuti peaks lisama võimaluse minna lihtsasti tagasi eelnevale versioonile ning vajadusel võimaldama kiiret versiooni uuendust. Kui need võimalused on loodud, on tarkvara tarnimine riskivabam. Kriitilise vea leidmisel saab kiiresti minna tagasi eelneva versiooni kasutamisele.

Projekti/rakenduse viimiseks seisu, kus automaattestide läbimisel võib vähese riskiga öelda, et toodangu kandidaat on sobilik kliendile üleandmiseks, tuleb koostada automaattestide komplekt, mis kontrollib väga suurt osa rakendusest. Samuti on oluline automatiseerida paigaldamist testimis-, proovi- (*staging*) ja toodangukeskkonda, et vältida käsitsi paigaldamisel tehtavaid vigu. Paljude süsteemide puhul on vajalik kasutada ka mitmeid teisi testimise vorme lisaks automaattestimisele, seega on vaja ka ka mitmeid teisi samme tarneprotsessis. Järgnev alamhulk peaks olema ühine kõigis projektides:

- **Kehtestamisfaasis** (*commit stage*) kinnitatakse, et süsteem töötab tehilisel tasandil. Selles etapis kompileeritakse kood, läbitakse komplekt automaatteste, mis koosnevad põhiliselt ühiktestidest ja käitatakse koodi analüsaatorit.
- **Automatiseeritud vastuvõtutestide etapis** kinnitatakse, et süsteem töötab funktsionaalsel ja mittefunktsionaalsel tasandil, vastavalt kasutuses olevale testidekomplektile. Rakendus peab vastama kasutajate vajadustele ja kliendi poolt kirjeldatud spetsifikatsioonile.
- **Käsitsi testimise etappides** kinnitatakse, et süsteem on kasutatav ja vastab nõuetele. Selles etapis avastatakse vead, mida automaattestidega ei leitud ja kinnitatakse, et tehtud muudatused loovad kasutajatele lisaväärtust. Need etapid sisaldavad tavaliselt uurivat testimist, integratsioontestimist ja kasutaja vastuvõtu testimist.
- **Väljalaske etapis** antakse versioon üle kliendile kas pakendatud tarkvarana või paigaldusena toodangu- või proovikeskkonda.

Eelnevalt väljatoodud etapid ja vajadusel neile lisatud sammud, mis on vajalikud edukaks tarkvara tarneprotsessiks, moodustavadki paigalduskonveieri. Kirjandusest võib leida

erinevaid nimesid - *continuous integration pipeline, a build pipeline, a deployment production line* või ka *a living build*. Pole vahet, kuidas seda nimetada, oma sisult on see automatiseeritud tarkvara üleandmise (*delivery*) protsess. Antud juhul pole inimese sekkumine sellesse protsessi välistatud, vaid veaohtlikud ja keerukad sammud protsessis on automatiseeritud.

6.1 Paigalduskonveieri praktikad

Et saada täielikku kasu paigalduskonveierist, tuleks järgida järgnevaid praktikaid [2]:

- **Binaarkood ehitada kokku ainult üks kord.** Iga kord, kui rakenduse kood kompileeritakse, võib tulemus olla erinev. Põhjuseid on mitmeid, näiteks kompilaatori versioon on vahepeal muutunud. Võib juhtuda, et valitakse mõnest teegist teine versioon või on muutunud kompilaatori konfiguratsioon. Kõik need võivad olla põhjuseks, et rakendus käitub algsest erinevalt. Lisaks on iga keskkonna jaoks koodi uuesti kompileerimine aeganõudev tegevus, mis takistab meeskonnale kiire tagasiside andmist.

- **Paigaldada igasse keskkonda samamoodi.** Rakenduse paigaldamiseks ükskõik millisesse keskkonda tuleks alati kasutada sama skripti, et ehitus ja paigaldusprotsess saaksid efektiivselt testitud. Rakendust paigaldatakse iga päev arendajate keskkondadesse, testkeskkondadesse veidi harvem ning toodangukeskkonda kõige harvem. Samas on just toodangukeskkonda paigaldamine kõige olulisem ning on esmatähtis, et paigaldusprotsess oleks testitud. Kõik keskkonnad erinevad üksteisest millegi poolest, vähemalt on kõigil keskkondadel erinev IP-aadress. Ometi pole mõttekas teha iga keskkonna jaoks eraldi paigaldus skripti, vaid iga keskkonna erilised seaded tuleks hoida skriptist eraldi, näiteks omaduste (*properties*) failis. Nii on võimalik kiiresti aru saada, mis vahe on erinevatel keskkondadel. Kui paigaldamine erinevatesse keskkondadesse toimub erinevalt, ei saa kunagi kindel olla, et rakendus, mis testkeskkonnas töötab, teeb seda ka toodangukeskkonnas.

- **Paigaldusele rakendada suitsuteste (*smoke-test*).** Kui rakenduse uus versioon paigaldatakse mingisse keskkonda, tuleks käima panna skript, mis teeb uuele versioonile suitsuteste, et veenduda rakenduse töötamises. Minimaalselt peaksid

suitsutestid kontrollima, et rakenduse pealeht avaneb ning seal on oodatud sisu. Samuti võiks kontrollida, et kõik välised süsteemid, teenused ja andmebaas samuti töötavad. Kui projektis on kasutusele võetud automaatsed ühiktestid ja neid käitatakse pidevalt, on väga oluline kirjutada rakendusele ka suitsutestid. Tänu suitsutestidele on võimalik veenduda, et rakendus tegelikult ka töötab ning kui ei tööta, peaksid suitsutestid sellest märku andma.

- **Paigaldada toodangulaadsesse keskkonda.** Põhiline viga, mida paljudes projektides tehakse, on toodangukeskkonna väga suur erinevus test- ja arenduskeskkondadest. Et olla kindel toodangukeskkonda paigaldamise edukuses, peaksid testimis- ja pideva integratsiooni keskkonnad olema toodangukeskkonnaga võimalikult sarnased. Ideaalis võiks kasutada toodangukeskkonna võimalikult täpset koopiat. Et olla veendunud, et keskkondade seadistused on samad, tuleb paika seada vastav distsipliin ja rakendada konfiguratsiooni halduse häid praktikaid.
- **Iga muudatus peaks kohe minema läbi paigalduskonveieri.** Enne pideva integratsiooni laiemat levikut kasutati paljudes projektides erinevate protsesside käitamist ajakava alusel. Näiteks iga tunni aja tagant ehitati rakendusest uus versioon, vastuvõtu teste käitati öösiti ja jõudlusteste *(capacity)* käitati nädalavahetustel. Paigalduskonveieri puhul on lähenemine järgnev: iga sissekande puhul käivitatakse paigalduskonveieri esimene samm ning kui see läbitakse edukalt, liigutakse koheselt järgmise sammu juurde kuni jõutakse paigalduskonveieri lõpuni. Selline lähenemine pole aga alati võimalik, eriti suurte meeskondade puhul, kui arendajad teevad koodi sissekandeid väga tihti, teades et paigalduskonveieri sammud võivad võtta palju aega. Siis on mõttekas kasutada varianti, kus esimese kontrollitud versiooni korral käivitatakse ühiktestide samm ja kui see läbitakse edukalt, siis käivitatakse ka automaatsete vastuvõtutestide samm, mis võtab oluliselt kauem aega. Kui vahepeal kannab keegi sisse uue versiooni, mille peale käivitatakse ühiktestide samm. Isegi kui see samm läbitakse edukalt, ei saa käivitada automaatseid vastuvõtuteste, sest need juba töötavad. Vahepeal tehakse aga veel kaks sissekannet, kuid ühiktestid tuleks käivitada ainult neist uuemal versioonil. Kui see samm ebaõnnestub siis on CI süsteemil raske aru saada,

millise versiooni ühiktestid ebaõnnestusid, samas arendajad saavad selle lihtsasti välja selgitada. Igal juhul peavad arendajad vea parandama ja tegema uue sissekande, millel käitatakse uuesti ühikteste. Kui lõpuks esimesel versioonil töötanud automaatsete vastuvõtutestide samm on lõpetanud, käivitab CI süsteem uuesti automaatsete vastuvõtutestide sammu kõige uuemal ühiktestide sammu läbinud versioonil. Sellise intelligentse ajakava kasutamine on paigalduskonveieri puhul väga oluline ning selle rakendamiseks tuleb veenduda, et kasutusel olev CI süsteem sellist ajakava haldust ka toetab. Samas saab seda rakendada ainult täielikult automatiseeritud sammudel nagu ühiktestide ja automaatsete vastuvõtutestide sammud. Hilisematel sammudel, kus paigaldatakse versioon manuaalsetestimise keskkondadesse, tuleks käivitada vastavalt testijate soovile.

- **Kui ükskõik milline samm muudatuse teekonnas läbi paigalduskonveieri kukub läbi, tuleks selle teekond peatada.** Kui mingi samm paigalduskonveieris ebaõnnestub, ei tohi vastaval versioonil käivitada uut sammu, vaid tuleb ebaõnnestumisest kohe meeskonda teavitada. Kogu meeskond peab töötama selle nimel, et viga, mis läbikukkumise põhjustas, saaks koheselt parandatud.

6.2 Kehtestusfaas *(Commit stage)*

Peale iga muudatuse sissekannet versioonihaldusesse luuakse uus eksemplar rakendusest ja kui see kompileeritakse edukalt, on tulemuseks toodangu kandidaat. Kehtestusfaasi eesmärgiks on elimineerida kõik versioonid, mis ei sobi toodangusse ja anda arendusmeeskonnale võimalikult kiiresti teada, et rakendus on katki. Seetõttu on oluline, et see etapp läbitaks kiiresti, ideaalis peaks selle läbimine võtma aega vähem kui viis minutit. Arendaja, kes tegi sissekande, peaks ootama ära kehtestusfaasi testide tulemused, enne kui ta asub järgmise tööülesande juurde.

Kehtestusfaas koosneb tavaliselt järgnevatest sammudest:

- **Koodi kompileerimine (kui vajalik).** Kood kompileeritakse ning kui see ebaõnnestub, teavitatakse vastavat arendajat ebaõnnestumisest. Samuti loetakse kehtestusfaas ebaõnnestunuks ja paigalduskonveieri eksemplari ei lubata järgmistesse sammudesse edasi.

- **Testide käitamine.** Enamus teste, mida selles sammus käitatakse, on ühiktestid, kuid võiks käitada ka teist tüüpi teste (nt automaatseid vastuvõtuteste, jõudlusteste), et tõsta kindlustunnet antud versiooni korrektse töö kohta. Ühiktestidele lisaks käitatavad testid tuleks valida aja jooksul vastavalt sellele, milliste testidega rakenduses kõige rohkem vigu leitakse. Kui testid kukuvad läbi, tuleks kehtestusfaas lugeda ebaõnnestunuks.
- **Koodi analüüs.** Koodianalüsaatori kasutamine erinevate meetrikate mõõtmiseks, nt testikate, duplitseeritud kood, tsüklomaatiline keerukus, hoiatuste arv, koodistiil. Kui selles sammus ei täideta eelnevalt meetrikatele seatud lävendeid, siis tuleks kehtestusfaas lugeda ebaõnnestunuks.
- **Järgnevate sammude jaoks artefaktide loomine.** Kui eelnevad sammud on õnnestunud, luuakse koodist paigaldatav üksus, mida saab kasutada erinevatesse keskkondadesse paigaldamiseks. Kui ka see samm läbitakse, võib kehtestusfaasi lugeda õnnestunuks.

Kehtestusfaasi läbimine on oluline samm toodangu kandidaadi teekonnal läbi paigalduskonveieri. Pärast selle sammu läbimist võib arendaja ette võtta järgmise tööülesande, kuigi talle jääb kohustus jälgida antud eksemplari edasist progressi paigalduskonveieris. Katkise versiooni parandamine peaks olema arendusmeeskonnas kõige kõrgema prioriteediga ülesanne olenemata sellest, millises kontrolli etapis viga välja tuleb.

6.3 Automaatsete vastuvõtutestide etapp

Automaatsete vastuvõtutestide etapi eesmärgiks on näidata, et süsteem loob kliendile oodatud lisaväärtust ning et see vastab vastuvõtu kriteeriumitele. See etapp pakub ka regressioontestimise jaoks vajalikku testide komplekti, millega kontrollitakse, et uue arendusega pole tehtud vigu, mis lõhuksid varem valminud funktsionaalsust. Antud etapis kasutatavate testide loomise ja haldamisega on seotud kogu arendusmeeskond. Arendajad, testijad ja klient töötavad koos, et luua vastuvõtuteste paralleelselt koodi ja ühiktestide kirjutamisega.

Automaatsete vastuvõtutestide käitamisel ilmnenud vigade korral peab keegi arendusmeeskonnast koheselt reageerima. Tuleb otsustada, mis on testide ebaõnnestumise põhjuseks, nt kas leiti regressiooniviga, testi ebaõnnestumine, on tingitud funktsionaalsuse muutumisest või on tegemist vigase testiga. Pärast testi ebaõnnestumise põhjuse välja selgitamist, tuleb vastavalt probleemi allikale leida lahendus, nii et vastuvõtutestid läbitaks edaspidi korrektse tulemusega.

Vastuvõtutestide automatiseerimine on küll kasulik, aga samas on nende loomine ja hooldamine väga kulukas. Seetõttu tuleb meeles pidada, et automaatsed vastuvõtutestid töötavad ka regressioontestidena ning alguses nende loomisele kulutatud aeg võib olla väiksem kui hiljem regressioontestimisele kuluv aeg. Samas, kui on näha, et testide loomisele ja hooldamisele kulub rohkem aega, kui need kokku hoiavad, siis ehk on mõttekam neist loobuda. Sellist olukorda saab vältida, kui testidega katta eeskätt kõige olulisem funktsionaalsus. Kindlasti pole mõtet automaattestidega katta kogu funktsionaalsust, vaid hoolikalt läbi mõelda, millist funktsionaalsust oleks kõige mõttekam automaattestidega katta.

6.4 Järgnevad testimise etapid

Automaatsete vastuvõtutestide etapi läbimine on toodangu kandidaadi teekonnal läbi paigalduskonveieri väga oluline. Kuni selle hetkeni on kogu protsess olnud automaatne ning pärast iga etapi edukat läbimist on versioon automaatselt edasi suunatud järgmisse etappi. Kõige lihtsamate paigalduskonveierite puhul on võimalik, et pärast vastuvõtutestide etapi läbimist paigaldatakse versioon automaatselt toodangu keskkonda. Samas on paljude süsteemide puhul oluline, et viidaks läbi ka mingil kujul käsitsi testimine, isegi kui on kasutusel laialdane automaattestide komplekt. Projektides on tihti olemas keskkonnad integratsiooni testimiseks teiste süsteemidega, keskkonnad jõudlustestide tegemiseks, uurivatestimise keskkonnad ning proovi- ja toodangukeskkonnad.

Paigalduskonveier vastutab ka testimiskeskkondadesse paigalduse eest. Väljalasete haldus või pideva integratsiooni haldussüsteemid pakuvad võimalust jälgida, milline versioon tarkvarast on hetkel mingisse keskkonda paigaldatud. Põhimõtteliselt on nupuvajutusega võimalik sellesse keskkonda paigaldada uus versioon rakendusest. Samas käitatakse taustal eelnevalt kirjutatud paigaldusskript, mis tegelikult paigalduse teostab. Nii saavad testijad paigaldada soovitud keskkonda neile hetkel vajaliku versiooni tarkvarast.

6.4.1 Käsitsi testimine

Iteratiivse tarkvaraarenduse puhul järgneb automaatsele vastuvõtutestimisele alati käsitsitestimise etapp, mille käigus rakendatakse uurivat testimist, testitakse süsteemi kasutatavust ning vajadusel käiakse valminud funktsionaalsus koos kliendiga üle. Neid tegevusi ei tehta aga mitte ühegi sellise versiooniga, mis ei ole läbinud automaatset vastuvõtutestimist. Paigalduskonveieri kasutamise puhul pole testija roll teostada regressioontestimist, vaid kinnitada, et vastuvõtu kriteeriumid on täidetud. Pärast seda saavad testijad tegeleda tegevustega, milles nad on osavamad kui automaattestid – uurivtestimine, kasutatavuse testid, kasutajaliidese disaini ülevaatamine ja halvima juhu testid. Need on samas ka tegevused, milleks testijatel ei pruugi jääda piisavalt aega kui automaatteste ei kasutata.

6.4.2 Mittefunktsionaalne testimine

Igal süsteemil on palju mittefunktsionaalseid nõudeid. Näiteks peab süsteem tavaliselt vastama mingitele kindlatele jõudluse ja turvalisuse nõuetele. Seetõttu oleks hea käitada automaatteste, mis kontrollivad, kas süsteem vastab talle esitatud nõuetele. Mittefunktsionaalsete testide sammu võiks lisada ka paigalduskonveierisse, aga erinevalt automaatsete vastuvõtutestide etapist võiks siin lasta testijatel otsustada, kas pärast seda sammu pääseb versioon paigalduskonveieris edasi või mitte.

6.5 Väljalase

Paigalduskonveieri viimane samm on väljalase ehk siis uue rakenduse versiooni paigaldamine toodangukeskkonda. Ka siin tuleb abiks automatiseerimine. Toodangukeskkonda versiooni paigaldamine peaks olema sama lihtne nagu testimiskeskkondadesse paigaldamine. Valitakse välja sobiv versioon ja ühe nupuvajutusega käivitatakse paigaldusprotsess.

Nagu eelnevalt mainitud, peaks paigaldamine toimuma ühtemoodi kõigisse keskkondadesse. Seetõttu on väga oluline, et testimiskeskkonnad oleksid toodangu keskkonnale võimalikult sarnased, siis on nendesse paigaldamise käigus võimalik testida ka paigaldusskripti. Skript, mida paigalduseks kasutatakse, peaks olema kõigi keskkondade jaoks sama ning see peaks kasutama vastavalt keskkonnale vajalikku konfiguratsioonifaili.

Olenevalt rakendusest võib olla vajalik uue rakenduse versiooni paigaldamine toodangukeskkonda mitu korda päevas. Ka neil ettevõtetel, kes tegelevad karbitoodete arendamisega, võib olla mõne kriitilise vea või turvaaugu avastamise korral vajalik uue versiooni avalikuks tegemine nii kiiresti kui võimalik. Paigalduskonveieri kasutusele võtmine annab selleks võimaluse. Samas tuleb meeles pidada, et isegi kui on võimalik tarnida iga paigalduskonveieri läbinud tarkvara versiooni, pole alati mõistlik seda teha. Sellegipoolest on paigalduskonveieri kasutusele võtmisel positiivne mõju loodava tarkvara kiiremale ja kvaliteetsemale tarnimisele.

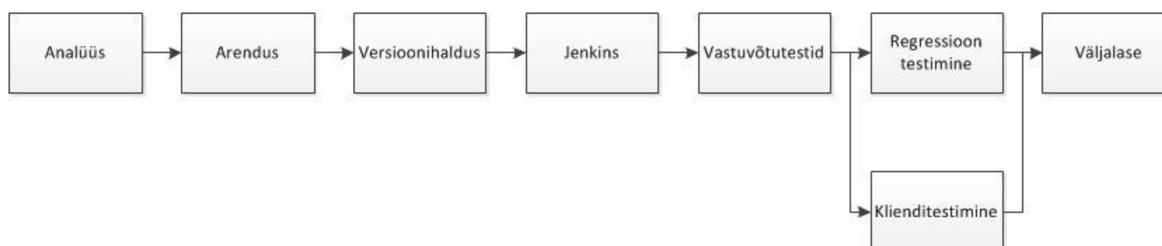
7 EMPIS arendusprotsess

EMPIS (*Employment Infosystem*) on Eesti Töötukassa infosüsteem, mida Webmedia arendab alates aastast 2009. Tegemist on veebipõhise infosüsteemiga, mille kasutajateks on töötukassa ametnikud. Antud süsteemis on võimalik isikuid töötuna ja tööotsijana arvele võtta, määrata neile tööturu toetusi, märkida üles töötukassase pöördumisi, vahendada tööpakkumisi, koostada otsuseid, sõlmida lepinguid jne. EMPISe eesmärgiks on lihtsustada ametnike tööd ning jätta rohkem aega töötu sisuliseks abistamiseks.

Nagu enamiku projektide puhul Webmedias, kasutatakse EMPISe arendamiseks agiilset arendusmetoodikat ning rakendatakse ka SCRUM raamistikku [21]. Süsteemi arendus toimub iteratsioonidena. Iga iteratsiooni tulemusena antakse kliendile üle uus funktsionaalsus ja varasemalt valminud funktsionaalsuse täiendused, mis on varem kliendiga kokku lepitud.

7.1 Arendusprotsessi kirjeldus

Järgneval joonisel on kujutatud EMPISe arendusprotsess:



Joonis 5. EMPISe arendusprotsessi sammud

Iga iteratsioon algab analüüsiga. Selles etapis lepitakse kliendiga kokku, milliseid töid antud iteratsiooni raames arendatakse. Samuti lisatakse tööülesannete haldusvahendisse vastavad tööülesanded. Seejärel kirjutab analüütik iga uue komponendi kohta tehnilise ülesande, mis kirjeldab detailselt ära selle komponendi funktsionaalsuse. Kui tegemist on varem valminud funktsionaalsuse täiendamisega, siis täiendatakse vastavalt ka varem valminud tehnilist ülesannet. Vajadusel täiendatakse ka andmemudelit ning joonistatakse prototüüp. Analüütiku töö tulem on ka sisendiks programmeerijale, kes hakkab antud komponenti arendama. Enne kui tehniline ülesanne jõuab arendajani, teostatakse analüüsi ülevaatus. Selle ülevaatuse teeb tavaliselt testija või mõni teine analüütik. Analüütik vaatab

ülevaatusel leitud vead ja võimalikud täiendusettepanekud üle ning vajadusel muudab kokkuleppel kliendiga tehnilist ülesannet. Seejärel suunatakse komponendiga seotud tööülesanne arendusse.

Ideaalis on analüüs alati arendusest ühe iteratsiooni jagu ees. See tähendab, et kui arendajad alustavad töödega iteratsioonis A, alustavad analüütikud juba iteratsiooni B tööde kokkuleppimist ja analüüsi. Kui tsükli tööde analüüs on valmis, hindab iga arendaja talle suunatud tööülesannete arendamiseks kuluvat aega. Arendajate poolt hinnatud ajale liidetakse hinnanguliselt testimisele ja vigade parandamisele kuluv aeg. Nii on võimalik paika panna tähtaeg, millal tööd võiksid valmis saada, et klient saaks planeerida aega klienditestimise ja koolituste jaoks.

Arenduse etapis tuleb jälgida, et esmalt teostatakse need ülesanded, millest võivad sõltuda teised samas tsüklis teostatavad tööülesanded. Samuti on oluline, et arendajad planeeriksid oma tööde järjekorda vastavalt tööülesannete prioriteedile ja keerukusele. Kuna keerukamate komponentide arendamine on alati riskantsem. Kui need jätta tsükli lõppu, on suur oht, et minnakse tähtajast üle.

Arendajad kirjutavad lisaks rakenduse koodile ka komponentide kohta toimivaid automaatteste. Kui arendaja on lahendanud käsil oleva tööülesande, salvestab ta tehtud muudatuse versioonihaldusesse ning suunab tööülesande testijale ülevaatamiseks. Lisaks tekitab ülesande koodiülevaatuseks, mida teostab mõni teine arendaja. Koodiülevaatuse käigus vaadatakse kood üle pilguga, et see järgiks kokkulepitud programmeerimistavasid, nagu näiteks, et see oleks piisaval hulgal ühiktestidega kaetud. Koodiülevaatusest ei sõltu tööülesande jõudmine testijate kätte. Koodiülevaatus on paralleelne tegevus ning selle järel ei oodata.

EMPISes on pideva integratsiooni keskkonnana kasutusel Jenkins [22], mis jälgib kasutusel olevat versioonihaldussüsteemi. Kui versioonihaldussüsteemi ilmub uus versioon, üritab Jenkins ehitada sellest kokku rakenduse ning seejärel üritab seda käivitada. Kui rakenduse käivitamine õnnestub, käivitatakse ka automaattestid ning nende läbimisel koostatakse raport tulemuste kohta. Ebaõnnestumise korral teavitatakse sellest automaatselt meeskonda.

Kui tööülesanne jõuab testija kätte, kontrollib ta esmalt, kas Jenkinsis on seda tööülesannet lahendav versioon olemas ning kas selle versiooni ehitamine on õnnestunud. Versiooni olemasolul paigaldab testija selle või sellest uuema versiooni testkeskkonda. Antud samm on manuaalne, et testijad saaksid vastavalt vajadusele neile sobivat versiooni testida. Eesmärgiks on omada kontrolli testkeskkondade üle. Selle sammu automatiseerimine häiriks oluliselt testijate tööd, sest arendajad salvestavad päeva jooksul korduvalt versioonihaldusesse uusi muudatusi.

Paigaldamine toimub tomcat serveris kasutades Jenkinsi poolt ehitatud paigaldusfaili (*war*-fail). Keskkonna spetsiifilist konfiguratsiooni hoitakse serveris ning see sisaldab vähemalt järgnevaid andmeid: andmebaasi URI, kasutatava skeemi nimi, kasutajanimi, parool. Kõik EMPISes kasutusel olevad keskkonnad kasutavad erinevat andmebaasiskeeme. Erinevad kasutusel olevad keskkonnad on kirjeldatud peatükis 7.4.2.

Pärast soovitud versiooni paigaldamist testkeskkonda veendub testija, et loodud funktsionaalsus töötab ja vastab tehnilisele ülesandele ning kliendi soovidele. Juhul kui kõik ei ole korrektne, saadab testija tööülesande arendajale tagasi. Korrektse lahenduse korral sulgeb testija tööülesande.

Kui kõik jooksvasse tsüklisse kuuluvad tööülesanded on suletud (või vähemalt enamik neist), paigaldatakse kõige uuem versioon rakendusest kliendi testkeskkonda. Selles keskkonnas teostavad kliendi esindajad vastuvõtutestimise neile saadetud tööülesannete nimekirja alusel. Klienditestimisega paralleelselt toimub regressioontestimine, mida teostavad arendusmeeskonda kuuluvad testijad. Regressioontestimist teostatakse rakenduse protsesside nimekirja alusel. Tegemist on nimekirjaga, kuhu on üles märgitud kõik rakenduse protsessid ning igal protsessil on olemas prioriteet. Seda nimekirja täiendatakse pidevalt. Regressioontestimise käigus testitakse üle kõik protsessid, mille prioriteet on 1 ja 2. Võimalusel testitakse ka madalama prioriteediga protsesse. Ideaalis testitakse üle kõik protsessid ning ka klient teostab regressioontestimist, seda küll mitte otseselt protsesside nimekirja alusel.

Klienditestimise ja regressioontestimise edukal läbimisel toimub tarne kliendi toodangukeskkonda. Loa tarne tegemiseks peavad andma kliendi esindaja, arendusmeeskonna poolne projektijuht ja vastutav testija. Kuna toodangukeskkonda haldab

kolmas osapool, siis ka paigalduse sellesse keskkonda teeb kolmas osapool. Webmedia ülesandeks on salvestada väljavalitud tarkvaraversiooni paigaldusfail repositooriumisse, kust kolmas osapool sellele ligi pääseb. Paigalduse tegemise ajal on Webmedia arendusmeeskonna liikmed valmis reageerima, kui paigaldamisel tekib takistusi.

7.2 Versioonihaldus

Iga rakenduse arendamisel tekib palju artefakte, mille paremaks haldamiseks oleks vaja neid versioneerida. EMPISes võib artefaktid jagada kolme rühma: dokumendid, tööülesanded ja rakenduse kood. Erinevaid dokumente hoitakse *Confluenc*'is, tööülesandeid tööülesannete haldusvahendis *JIRA* ja koodi *Mercurial*'i koodihaldus vahendis.

7.2.1 Mercurial

Mercurial'i puhul on tegemist hajusa versioonihaldussüsteemiga *(distributed source control management tool)*, mis tähendab, et igal arendajal on oma masinas lokaalne repositoorium, kuhu ta kõigepealt koodis tehtud muudatused salvestab. Muudatuste avalikustamiseks peab arendaja oma lokaalse repositooriumi sünkroniseerima tsentraalse repositooriumiga ning seejärel muudatused tsenraalsesse repositooriumisse salvestama. Tsentraalsesse repositooriumisse salvestamine toimub siis, kui arendaja on lõplikult oma tööülesande valmis saanud. Töö käigus salvestab ta oma koodi lokaalsesse repositooriumisse ja konfliktide vältimiseks sünkroniseerib seda tsentraalse repositooriumiga. Seega lahendatakse tekkinud konflikte juba koodi kirjutamise käigus, mitte ainult tsentraalsesse repositooriumisse salvestamisel.

7.2.2 Andmebaasi propagaator

Omamoodi versioonihaldusvahendina on EMPISes kasutusel ka andmebaasi propagaator. Kõik andmebaasis tehtud muudatused nii struktuuris, kui ka andmete migreerimisel teostatakse skriptidega, mis salvestatakse versioonihaldusvahendisse. Et tagada andmete korrektsus, tuleb need skriptid käivitada iga keskkonna juures olevas andmebaasis. Selleks on vahend, mis jälgib versioonihaldussüsteemi ja kui sinna on lisandunud mõni skript, mida pole rakendusega seotud andmebaasis käivitatud, siis käivitab need. Juba käivitatud skripte enam uuesti ei käivitata. EMPISes on andmebaasi propagaator üks paigalduse osa.

Igakord, kui rakendus paigaldatakse, kävitatakse ka propagaator. Nii on igas keskkonnas tagatud andmete korrektsus.

7.3 Jenkins

Oluline roll rakenduse efektiivsemal ja kvaliteetsemal arendamisel on pideva integratsiooni keskkonnal, kuna tänu selle kasutamisele saab arendusmeeskond iga muudatuse versioonihaldusesse salvestamise järel kiiresti tagasisidet selle seisu kohta. EMPISes on pideva integratsiooni keskkonnana kasutusel Jenkins, mis vastutab iga versioonihaldusesse jõudnud muudatuse ehitamise ning automaattestide ja koodianalüüsi vahendite käitamise eest. Pärast kõikide sammude läbimist koostatakse raport tulemuste kohta.

Jenkinsi seadistamiseks on väga palju erinevaid võimalusi. Samuti on saadaval väga palju erinevaid pistikprogramme, mida saab oma Jenkinsi keskkonnale lisada. EMPISe puhul on kasutusel:

- FindBugs [23] – kontrollib kompileeritud Java klasse;
- PMD analysis [24] – kontrollib Java lähtekoodi;
- Java Warnings [25] – kontrollib logisid;
- Task Scanner [26] – näitab tegemata tööülesannete (//TODO kommentaarid koodis) arvu;
- Test Results – näitab ühiktestide tulemusi, kui palju teste läbiti ja palju ebaõnnestus;
- TestNG [28] – näitab automaatsete vastuvõtutestide tulemusi, kui palju teste läbiti ja palju ebaõnnestus;
- SLOCCount [27] – tuvastab lisatud, eemaldatud ja muudetud koodiridade arvu;
- The Continuous Integration Game [29] – arvutab etteantud reeglite järgi punkte, mis hindavad versiooni edukust.

Iga Jenkinsis ehitatud versiooni puhul on võimalik vaadata, millised olid erinevate pistikprogrammide käitamise tulemused. Joonisel 6 on EMPIS rakenduse versiooni 3.1.1.456 pistikprogrammide käitamise tulemused.

Build #2478 (10.05.2012 16:33:52)

[version] 3.1.1.456

Changes

1. Merged heads of branch release-3.1 (detail / hgweb)
2. EMPIS-191138: Loendi LEPING_LIIK asendamine loendiga TEENUS (detail / hgweb)

Started by an SCM change

Revision: a69a95c2b689db774d08bf0c4ee04e08f0383ca4

FindBugs: 0 warnings from one FindBugs analysis.

- No warnings since build 2,185.
- New zero warnings highscore: no warnings for 93 days!

PMD: 130 warnings from one PMD analysis.

- 1 new warning
- 1 fixed warning

Java Warnings: 0 warnings.

- No warnings since build 519.
- New zero warnings highscore: no warnings for 794 days!

Task Scanner: 66 open tasks in 3,607 workspace files.

- Plug-in Result: - no threshold has been exceeded (Reference build: #2477)
- New highscore: only successful builds for 650 days!

Test Result (2 failures / ±0)
Show all failed tests >>>

309428 (+37) lines in 5273 (+2) files and 5 (+0) languages.

- sql : 40021 (+49) lines in 1081 (+3) files.
- java : 237855 (-12) lines in 3607 (-1) files.
- lisp : 85 (+0) lines in 1 (+0) files.
- sh : 52 (+0) lines in 4 (+0) files.
- jsp : 31415 (+0) lines in 580 (+0) files.

The build was worth 0 points

Joonis 6. Näidis Jenkinsi raportist.

Kuigi kõigi nende pistikprogrammide tulemused annavad meile versiooni kohta olulist informatsiooni, huvitavad meid enim testide tulemused. EMPISe puhul on seadistatud Jenkins nii, et kogu voog käiakse algusest lõpuni läbi, olenemata sellest, kas mõni samm vahepeal ebaõnnestub. Nii saame iga versiooni kohta võimalikult palju tagasisidet. Joonisel 7 on näidis Jenkinsis käitatud testide tulemustest. Koodi kompileerimise ebaõnnestumisel Jenkins rakendust automaattestimisekeskkonda ei paigalda ning teste ei käivita.

Test Result

2 failures (±0)

2,177 tests (±0)
Took 8 min 34 sec.
add description

All Failed Tests

Test Name	Duration	Age
>>> ee.tootukassa.empis.leping.osavott.OsavotugraafikDaoTest.findIsikOsavotugraafik	0.15 sec	9
>>> ee.tootukassa.empis.aruanded.ReportDaoTest.findYleminevadLepingud	69 ms	20

All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Total	(diff)
ee.tootukassa.empis.administreerimine.juriidilisedalused	1.8 sec	0		0		4	
ee.tootukassa.empis.administreerimine.makseliigid	1.5 sec	0		0		9	
ee.tootukassa.empis.administreerimine.riigiloiv	1.2 sec	0		0		3	
ee.tootukassa.empis.administreerimine.teenusedseisundid	0.94 sec	0		0		4	
ee.tootukassa.empis.alfresco	6.5 sec	0		0		9	
ee.tootukassa.empis.aruanded	10 sec	1		0		27	
ee.tootukassa.empis.aruanded.toopraktika	1.5 sec	0		0		5	

Joonis 7. Näidis Jenkinsis käitatud testide tulemustest.

Jenkinsit kasutavad nii arendajad kui ka testijad. Arendajad saavad sealt tagasisidet enda tehtud muudatuste kohta ja testijad näevad projekti hetkeseisu ning saavad selle põhjal teha otsuseid, milliseid versioone on võimalik erinevatesse testkeskkondadesse paigaldada. Samuti on loodud Jenkinsisse võimalus testkeskkondadesse uue versiooni paigaldamiseks. Selleks peab testija valima välja sobiva versiooni ning keskkonna, kuhu ta soovib seda paigaldada ning ühe nupuvajutusega käivitatakse paigaldusskript, mis selle operatsiooni teostab.

7.4 Testimisstrateegia

Olenevalt arendustsükli faasist viiakse läbi testimist kasutades selleks erinevaid testimise tehnikaid ja vahendeid. Tsükli algusfaasis viiakse läbi staatilist testimist, mille käigus vaadatakse üle analüütikute poolt koostatud tehnilised ülesanded ja süsteemi disain. Kirjutatud koodile tehakse teiste arendajate poolt koodiülevaatusi ja pideva integratsiooni keskkonnas kasutatakse koodianalüüsi vahendeid. Rakenduse automaatseks testimiseks kirjutatakse ühikteste ja automaatseid vastuvõtuteste. Iga tööülesanne läbib testijapoolse manuaalse valideerimise. Arendustsükli lõpufaasis teostab klient omapoolsed vastuvõtutestid ning arendusmeeskond tegeleb regressioontestimisega.

7.4.1 Automaattestid

Väga olulisel kohal rakenduse kvaliteedi tagamisel on automaattestid. EMPISes on kokku lepitud, et arenduse käigus katab arendaja enda kirjutatud koodi ühiktestidega. Testide kirjutamisega kontrollib arendaja, et kood töötab nii nagu tema arvates on õige. Need testid on kasulikud ka juhul, kui keegi muudab testidega kaetud koodi. Testid tuvastavad, kas kood läheb katki või vastupidi, annavad kinnitust, et kood töötab ka pärast muudatusi korrektselt. Seda muidugi eeldusel, et testid ise on korrektselt kirjutatud. Täielikku koodi katvust pole aga mõistlik oodata, aga katta tuleks järgnevad kohad koodist:

- *DAO* meetodid ehk suhtlus andmebaasiga. Kindlasti peaksid need testid kontrollima SQL lausete korrektsust. Päringute puhul peaks kontrollima, et tagastatakse vähemalt üks kirje. Meetodite puhul, mis midagi uuendavad, peaks kontrollima, kas sisend ID'de arv võrdub uuendatud kirjete arvuga.

- Taaskasutatavad *utility* meetodid. Tuleb kontrollida, et erinevate sisendandmete korral on kood võimalikult suures osas testidega kaetud.

- *BO* meetodid ehk ärikihi loogika meetodid. Ärikihi loogika ei pea küll olema täielikult kaetud, aga mida keerukama ja tähtsama loogikaga on tegu, seda olulisem on selle testidega katmine.

Lisaks ühiktestidele kirjutatakse ka automaatseid vastuvõtuteste. Nende testide kirjutamine on oluliselt keerukam ja aeganõudvam kui ühiktestide kirjutamine. Kõigepealt on vaja ette valmistada testjuhtum ning selle järgi saab arendaja kirjutada automaattesti. Automaatsed vastuvõtutestid suhtlevad rakendusega kasutajaliidese kaudu, imiteerides kasutaja tegevusi. Kuna nende kirjutamine võtab palju aega, ei kaeta nendega kogu rakendust. EMPISes on automaatsete vastuvõtutestidega kaetud osa rakenduse protsessidest. Automaatsed vastuvõtutestide kirjutamiseks on kasutatud Seleniumi.

7.4.2 Testimiskeskkonnad

Rakenduse testimise jaoks olenevalt teostatavast tegevusest ja selle eesmärkidest, on vaja kasutada erinevaid testimiskeskkondi. Erinevates keskkondades võivad olla erinevad rakenduse versioonid ning neid kasutatakse erinevate testimistoimingute jaoks. EMPISes on kasutusel lisaks toodangukeskkonnale neli testimiskeskkonda:

- *empis-latest*, kuhu on paigaldatud kõige uuem versioon rakendusest. Selles keskkonnas toimub käesoleva arendustsükli tööde testimine ning seetõttu on oluline, et seda keskkonda pidevalt uuendataks.
- *empis-current*, kuhu on paigaldatud kliendi toodangukeskkonnas hetkel kasutusel olev versioon. See keskkond on vajalik toodangukeskkonnas avastatud vigade reprodutseerimiseks.
- *tk-test* – kliendi testkeskkond. See keskkond on mõeldud kliendile selleks, et arendustsükli lõpufaasis saaks üle vaadata valminud uue funktsionaalsuse. Sellesse keskkonda paigaldatakse ainult niisuguseid versioone rakendusest, mis on juba läbinud arendusmeeskonna poolse testimise.
- automaattestimise keskkond, kus töötavad ainult automaattestid. Eraldi keskkonda automaattestimise jaoks on vaja selleks, et tagada testide tulemuste korrektsust. Automaattestid nõuavad enamasti mingit kindlat andmete seisu ning seetõttu on hea kui nende käitamise jaoks on eraldi keskkond, mille andmeid keegi käsitsi muuta ei saa.

8 Pideva tarnimise strateegia rakendamine EMPISes

Antud peatükis võrreldakse EMPISe arendusprotsessi pideva tarnimise strateegia põhimõtete ja praktikatega. Põhimõtted ja praktikad, mille alusel võrdlust teostatakse, on kirjeldatud käesoleva töö peatükkides 3 – 6. Võrdlust teostatakse ainult pideva tarnimise strateegia kõige olulisemate põhimõtete korral. Väiksemaid soovitusi, mis nendes peatükkides välja on toodud, võrdlustes ei kajastata. Iga põhimõtte või praktika puhul tuuakse välja, kas see on juba EMPIS projektis kasutusel ning lisatakse kommentaar, mille alusel vastus on antud. Kõik kommentaarid on kirja pandud vastavalt töö autori nägemusele EMPIS arendusprotsessist ja seetõttu võivad mõned aspektid olla vaieldavad arendusmeeskonna teiste liikmete poolt.

8.1 Tarkvara tarnimise põhimõtted

Tabelis 1 on välja toodud EMPIS arendusprotsessi vastavus tarkvara tarnimise kaheksale põhimõttele. Tarkvara tarnimise põhimõtted on kirjeldatud antud töö peatükis 3.5.

Tabel 1. Tarkvara tarnimise põhimõtete kasutamine EMPISes.

Põhimõte/praktika	Kasutusel?	Kommentaar
Luua korratav ja usaldusväärne tarkvara väljalaske protsess.	Osaliselt	EMPISe arendusprotsessis on kasutusel CI süsteem, tänu millele käitatakse iga versioonihaldusesse tehtud sissekande puhul automaatteste ja koodianalüüsi vahendeid. Erinevatesse keskkondadesse paigaldamine on tehtud võimalikult lihtsaks, kuid Webmedia testimiskesk-kondadesse ja kliendi hallatavatesse kesk-kondadesse paigaldamiseks kasutatakse erinevaid paigaldusfaile.
Automatiseerida niipalju kui võimalik.	Osaliselt	Nagu eelnevalt kirjas, on kasutusel CI süsteem, milles käitatakse kõiki versiooni-haldusesse salvestatud muudatusi. CI süsteemis jooksevad ühiktestid ja koodianalüüsi vahendid, ning automaatsed

		vastuvõtutestid. Automaatsete vastuvõtu-testid katavad rakendust ainult osaliselt ning osa olemas olevaid teste on aegunud.
Hoida kõike versiooni-halduses.	Jah	Ühel või teisel moel hoitakse EMPIS projektis kõiki artefakte versioonihalduses. Täpsemalt on EMPISe versioonihalduse meetodeid kirjeldatud peatükis 7.2.
Probleemide ennetamiseks tuleks veaohtlikke tegevusi teha tihedamini.	Osaliselt	EMPIS projektis alustatakse testimisega juba enne tööülesannete arendusse jõudmist, vaadates üle analüütikute poolt kirjutatud tehnilised ülesanded. Uued arendused vaadatakse üle esimesel võimalusel pärast arenduse valmimist. Arendusprotsessis on probleemseks kohaks aga regressioontestimise faas, mis võtab arendustsükli lõpus palju aega. Automatiseeritud vastuvõtutestide komplekti kordategemine ja suurendamine oleks selles osas suureks abiks.
Kvaliteedi tagamine.	Osaliselt	Puudujäägiks on rakenduse ainult osaline kaetus automaatsete vastuvõtutestidega ning olemas olevate testide töökorras hoidmine.
Tehtud/valmis tähendab, et on kliendile üle antud.	Jah	Igapäevaselt ei pruugi kõik meeskonnaliikmed mõelda sellele, et funktsionaalsus on valmis alles siis, kui klient seda kasutab. Enamasti loetakse funktsionaalsus valminuks siis, kui vastav tööülesanne on suletud. Sellisele mõtlemisele aitab kindlasti kaasa ka kasutusel olev tööülesannete haldusvahend, kus pärast järelkontrolli õnnestumist ja koodiüle-

		vaatuse lõpetamist saab tööülesanne staatuseks „suletud“.
Kogu meeskond vastutab eduka tarneprotsessi eest.	Jah	Kogu arendusmeeskond töötab koos ühes ruumis ja omavaheline suhtlus on väga tihe. Analüütikud, arendajad ja testijad töötavad kõik koos selle nimel, et luua kliendile õiget ja kvaliteetset tarkvara. Loodavaid lahendusi arutatakse omavahel ning vajadusel küsitakse teiste abi. Teiste rollide tööülesandeid küll üle ei võeta, aga toetatakse igaühe töö tegemist nii palju kui võimalik.
Tarnimisprotsessi pidev täiendamine.	Osaliselt	Pärast iga arendustsükli lõppu toimuvad retrospektiivi koosolekud, kuhu tuleb kokku kogu arendusmeeskond. Koosoleku käigus saavad kõik osalised sõna ning peavad jagama oma ideid teemadel: mis läks möödunud tsüklis hästi, mis läks halvasti ja mida saaks teha paremini. See on koht, kus kiita kaastöötajaid ja samas juhtida tähelepanu probleemidele. Lisaks toimuvad iganädalased koosolekud, kus põhiliselt keskendutakse jooksvatele probleemidele. Vajadusel saab ka seal oma mõtteid jagada. Need koosolekud pole küll otseselt viidud läbi sooviga tarneprotsessi parandada, aga aitavad sellele kindlasti kaasa. Antud põhimõtte ideede kasutusele võtmine näiteks retrospektiivi koosoleku raames on täiesti mõeldav.

8.2 Versioonihaldus

Tabelis 2 on välja toodud EMPISes kasutatavate versioonihalduse praktikate vastavus antud töö peatükis 4.1 välja toodud praktikatele.

Tabel 2. Versioonihalduse praktikate kasutamine EMPISes.

Põhimõte/praktika	Kasutusel?	Kommentaar
Hoida kõike versiooni-halduses.	Jah	Kordab tarkvara tarnimise kolmandat põhimõtet. Kommentaar kirjas eelmise alampeatüki all.
Teha sissekandeid regu-laarselt.	Osaliselt	EMPISe arendajad paigaldavad oma muudatused Mercuriali tsentraalsesse repositooriumisse tavaliselt alles siis, kui neil tööülesanne valmis saab. Mõne suurema ülesande puhul võib sellisel juhul kahe sissekande vahe olla isegi mitu päeva. Lokaalsesse repositooriumisse salvesta-takse töid tihedamini ning lokaalset repositooriumi sünkroniseeritakse tsent-raalsega. Nii on võimalik vältida suuremaid konflikte tsentraalsesse repositooriumisse salvestamisel. Sissekannete regulaarsust tsentraalsesse repositooriumisse aitaks suurendada ka tööülesannete jagamine sobiva suurusega tükkideks.
Kasutada sisukaid *commit* sõnumeid.	Jah	EMPISes kasutatakse *commit* sõnumina alati arendusega seotud tööülesande nime ja ID-d, et hiljem oleks üheselt tuvastatav, mis muudatusega on tegemist. Täpsemat infot muudatuse kohta saab tavaliselt tööülesande kommentaaridest.

8.3 Testimisstrateegia

Tabelis 3 on toodud välja antud töö peatükis 5 kirjeldatud praktikad projekti testimisstrateegia väljatöötamiseks. Neid praktikaid on võrreldud EMPISes kasutusel oleva testimisstrateegiaga.

Tabel 3. Testimisstrateegia praktikate kasutamine EMPISes.

Põhimõte/praktika	**Kasutusel?**	**Kommentaar**
Testimisstrateegia olemasolu.	Jah	EMPISes on kasutusel testimisstrateegia, mis on kooskõlas ettevõtte kvaliteedi tagamise poliitikaga. Strateegia dokumendis kirjeldatakse kasutatavate meetodite, vahendite ja protsesside suhe.
Automaattestide olemasolu.	Jah	EMPIS rakenduse kood on kaetud olulisel määral ühiktestidega. Automaatseid vastuvõtuteste on kirjutatud kõige olulisemale funktsionaalsusele, mis on juba stabiilne ja mida pole ammu muudetud.
Testjuhtumid kirjutatakse valmis enne arendust.	Ei	Testjuhtumeid kirjutatakse funktsionaalsusele, mis on rakenduses väga oluline ja mida pole ammu muudetud. Selle praktika kasutusele võtmise üle on arutatud, kuid kasutusele võtmiseni pole jõutud.
Testimiskvadrantide kasutus.	Jah	Kõik testimise kvadrandid on EMPISes erinevates arendustsükli faasides kaetud, lähenemine testide jagamiseks nelja sektorisse on aga uus.

8.4 Paigalduskonveier

EMPIS arendusprotsessis pole kasutusel paigalduskonveierit, kuid kasutatakse pideva integratsiooni süsteemi. CI süsteemi võib samuti nimetada algseks paigalduskonveieriks ning paigalduskonveieri kasutusele võtmisel peaks nii või teisiti tegema seda olemasoleva CI süsteemi vahenditega. Tabelis 4 on võrreldud EMPISes kasutusel olevat CI süsteemi antud töö peatükis 6 kirjeldatud paigalduskonveieri praktikatega.

Tabel 4. Paigalduskonveieri praktikate kasutamine EMPISes

Põhimõte/praktika	Kasutusel?	Kommentaar
Binaarkood ehitada kokku ainult üks kord.	Osaliselt	EMPIS rakenduse puhul ehitatakse *war*-faile kaks korda. Ühte kasutatakse Webmedia hallatavates keskkondades ning teist kliendi hallatavates keskkondades (tk-test ja toodangukeskkond). Seega ei vasta praegune protsess otseselt antud praktikale, aga samas ei kasutata toodangukeskkonna paigaldamiseks faili, mida pole kasutatud kliendi testkeskkonda paigaldamiseks.
Paigaldada igasse keskkonda sama moodi.	Osaliselt	Paigaldamine testkeskkondadesse toimub põhimõtteliselt sama protsessi alusel. Toodangukeskkonnale Webmedial ligipääsu pole, seetõttu on ka sinna paigaldamine võrreldes teiste keskkondadega erinev. Samas on paigaldusprotsess ka praegu kõigi keskkondade puhul sarnane ning paigaldus toodangukeskkonda pole probleeme valmistanud.
Paigaldusele rakendada suitsuteste.	Ei	Hetkel suitsuteste ei kasutata. Kasutusel olevad automaatsed vastuvõtutestid suhtlevad rakendusega kasutajaliidese kaudu ja seetõttu ei ole ka otsest vajadust suitsuteste kasutusele võtta.

Paigaldada toodangu-laadsesse keskkonda.	Jah	Kõik kasutatavad testkeskkonnad on tehtud toodangukeskkonnale võimalikult sarnaseks. Kliendi testkeskkonna puhul on tegemist põhimõtteliselt toodangu-keskkonna koopiaga.
Iga muudatus peaks kohe minema läbi paigaldus-konveieri.	Ei/Jah	EMPISes pole otseselt kasutusel paigalduskonveierit ja seetõttu ei saa väita, et iga muudatus läheb läbi paigaldus-konveieri. Küll aga võib väita, et iga versioonihaldusesse salvestatud muudatuse peale käivitatakse CI keskkonnas antud versioonil automaattestid ja koodianalüüsi vahendid.
Kui ükskõik milline samm muudatuse tee-konnas läbi paigaldus-konveieri kukub läbi, tuleks selle teekond peatada.	Osaliselt	Antud põhimõte on väga range ja hetkel pole CI keskkonnas nii rangeid reegleid paika panud. Praegu lähtutakse põhi-mõttest, et soovime iga versiooni kohta saada võimalikult palju tagasisidet korraga ja seetõttu lastakse kõikidel testidel lõpuni käia. Kui rakenduse ehitamine ebaõnnes-tub, siis teste ja koodianalüsaatoreid ei käivitata.

8.5 Mida võiks EMPISes kasutusele võtta?

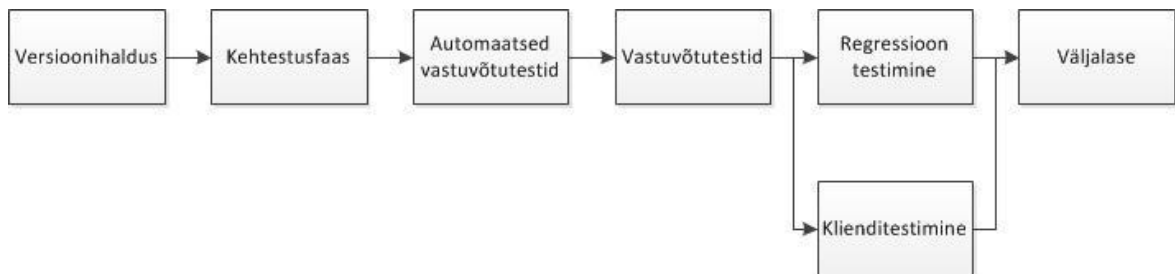
Pideva tarnimise strateegia on kogum põhimõtteid ja praktikaid tarkvara kiiremaks ja kvaliteetsemaks tarnimiseks kliendile. Antud töö raames on kirjeldatud pigem üldiseid arendusprotsessiga seotud põhimõtteid ja vähem on keskendutud väga spetsiifilistele andmete-, keskkondade- ja konfiguratsioonihalduse küsimustele. Sellegipoolest on välja toodud suur hulk soovitusi, mis võiks tarkvara arendusprotsessi efektiivsemaks muuta. EMPIS projektis on osad neist ka juba kasutusel. Kuid arendusprotsessi muutmiseks võimalikult pideva tarnimise strateegiale vastavaks tuleks esimese sammuna võtta

kasutusele paigalduskonveier. Et paigalduskonveieri kasutusele võtmine omaks mõtet, tuleks olulisel määral suurendada automaatsete vastuvõtutestide hulka ning muuta nende haldamine igapäevaseks rutiiniks. Kui need sammud on tehtud, siis jälgides arendusprotsessis olevaid pudelikaelu, on võimalik jooksvalt arendusprotsessi täiustada.

8.5.1 EMPIS paigalduskonveier

Paigalduskonveieri kasutuselevõtmisel tuleb kõigepealt valida tööriist, millel on olemas vastav funktsionaalsus. Hetkel on EMPISes kasutusel Jenkinsi CI keskkond, millele on saadaval pistikprogramm paigalduskonveieri jaoks [30]. Selle pistikprogrammi kasutusele võtmine oleks kõige lihtsam lahendus, kuna Jenkins CI keskkond on juba EMPISe jaoks seadistatud. Samuti on selle pistikprogrammi seadistamise kohta olemas palju juhendmaterjale.

Pärast vahendi valimist on oluline välja mõelda, milliseid samme soovitakse paigalduskonveieris näha. Põhimõtteliselt võiks see välja näha nagu näidatud Joonisel 8.



Joonis 8. EMPISe võimalik paigalduskonveier

Praeguse arendusprotsessiga võrreldes muutuks ainult niipalju, et lisandub automaatsete vastuvõtutestide etapp. Selle etapi suurim eesmärk on vabaneda regressioontestimise etapist. Nii suure projekti puhul, nagu EMPIS, pole see koheselt võimalik. Arvestades, et siiani on automaatseid vastuvõtuteste kirjutatud vähesel määral ja kui hakata neid tagantjärele kirjutama, võtab see väga palju aega ja ressursse. Selletõttu on ka regressioontestimise etapp paigalduskonveierisse sisse jäetud, küll aga peaks selle maht oluliselt vähenema. Kuna tegemist on manuaalse testimisega, nagu ka klienditestimise puhul, on need kaks etappi jäetud paralleelseteks.

EMPISes pole veel astutud reaalseid samme paigalduskonveieri kasutusele võtmiseks ja seetõttu on Joonisel 8 esitatud paigalduskonveier ainult üks võimalik variant, milline see võiks välja näha.

8.5.2 Automaatsed vastuvõtutestid

EMPISes paigalduskonveieri kasutuselevõtmiseks sellisel kujul nagu Joonisel 8 välja pakutud, tuleb kõigepealt üle vaadata olemasolevad automaatsed vastuvõtutestid ja vajadusel neid parandad. Seejärel tuleks olemasolev funktsionaalsus katta suuremas mahus automaatsete vastuvõtutestidega ning kasutusele võtta põhimõte, et uue funktsionaalsuse puhul kirjutatakse testjuhtumeid enne arenduse teostamist ning automaattestid kirjutatakse koos arendusega.

Soovides kasutatada automaatseid vastuvõtuteste paigalduskonveieri eraldi sammuna, mida teostatakse automaatselt, tuleb hoida testikomplekti kogu aeg töökorras. Selleks peab arendusmeeskonnas juurutama rutiini, et katkise testi ilmnemisel, tuleb see kohe korda teha. Üheks põhjuseks, miks hetkel on probleeme olemasolevate testide stabiilsusega, ongi sellise rutiini puudumine. Ennetavalt tuleb kindlasti luua hea ülevaade olemasolevast testide komplektist. Samuti tuleb hakata jälgima, et kui tehnilistesse ülesannetesse viiakse sisse muudatus, siis tuleb vaadata üle vastav(ad) testjuhtum(id). Kui on vajadus ka testjuhtumit muuta, siis tuleb sellest teavitada arendajat, et ka automaattesti vastav muudatus sisse viidaks.

9 Kokkuvõte

Antud töös uuriti pideva tarnimise strateegiat ning selle rakendamise võimalusi EMPIS projektis. Esmalt tutvustati pideva tarnimise strateegia üldiseid põhimõtteid ja praktikaid, konfiguratsioonihalduse tavasid, testimisstrateegia väljatöötamise põhimõtteid ja paigalduskonveierit. Seejärel kirjeldati EMPISes kasutusel olevat arendusprotsessi ning võrreldi seda pideva tarnimise strateegiaga. Lõpuks toodi välja soovitused pideva tarnimise strateegia kasutusele võtmiseks.

Üheks põhiliseks eesmärgiks oli leida võimalus, kuidas arendustsükleid lühemaks muuta ilma kvaliteedis järgi andmata. Selle eelduseks on vältida pikka regressioontestimise faasi. Pideva tarnimise strateegia pakub välja lahenduse probleemile automaattestide laialdase kasutuselevõtmisega. Kuid antud soovituse rakendamine projektis EMPIS on väga aeganõudev ja kulukas protsess.

Töö tulemusena toodi välja esmased soovitused, mida EMPIS projektis pideva tarnimise strateegia kasutuselevõtmiseks rakendada. Töö tulemusi tutvustatakse EMPISe arendusmeeskonnale. Heakskiidu saamisel on võimalik, et neid hakatakse projektis rakendama. Kuna teema on huvitav ja võib tuua olulist kasu projekti efektiivsemaks muutmisel, siis jätkatakse pideva tarnimise strateegia uurimist arendusprotsessi täiustamiseks.

Pideva tarnimise strateegia põhimõtete täielik rakendamine EMPISes on raskendatud, kuna projekt on kestnud juba rohkem kui kolm aastat ning tagantjärele muudatuste sisse viimine on keerukas. Seetõttu tuleks töös kirjeldatud põhimõtteid tutvustada alles alustavates ja algusfaasis olevatele projektidele, et nende arendusprotsess muuta algusest peale efektiivsemaks.

Continuous Delivery Implementation Analysis in Webmedia Group Project EMPIS

Master thesis (30 EAP)

Mirjam Rauba

Summary

This thesis is about investigating continuous delivery strategy and its implementation options in project EMPIS. General principles and practises of continuous delivery are introduced. The outcomes of investigation are compared to EMPIS development processes and some recommendations for implementing continuous delivery in EMPIS are laid out.

The main goal of this thesis is to find a way to avoid software development iterations that take several months to finish, without any change in quality. Prerequisite for this is to avoid long regression testing phases. Continuous delivery gives a solution to the problem by wide use of automation. But it would be a time-consuming and expensive process to adapt all the practices of continuous delivery in project EMPIS.

As the result of this thesis recommendations are made to adapt some of the practices of continuous delivery in project EMPIS to improve the development process. These results will be demonstrated to development team. In case of approval they might be implemented. As the topic is interesting, some further research will be made to learn more about continuous delivery while implementing it on EMPIS.

The full implementation of continuous delivery principles in EMPIS is difficult as the project has been going on for more than three years and making changes in the system is complex. Therefore the introduction of continuous delivery should be made for projects that are starting development as they could contribute more from this strategy.

Sõnastik

Acceptance tests – vastuvõtutestid

Build – ehitus. Arendaja ehitab oma muudatusest versiooni.

Capacity tests – jõudlustestid

Check in – sissekanne

Commit – kehtestama, muudatusi püsivateks salvestama [31]

Continuous Delivery – pideva tarnimise strateegia

Continuous Integration – pidev integratsioon

Cycle time – tsükliaeg

Deliver – üle andma, tarnima

Deploy – paigaldus. Versiooni paigaldamine mingisse keskkonda.

Deployment pipeline – paigalduskonveier. Tarkvarakonveierid (*software pipelines*) koosnevad mitmest protsessist, mis on organiseeritud nõnda, et ühe protsessi väljundvoog muudetakse automaatselt järgmise protsessi sisendvooks [31].

Exploratory testing – uuriv testimine

Integaration - integratsioon

Lean development – paindlik arendus (*lean management* – paindlik juhtimine)

Manual testing – manuaalne/käsitsi testimine ehk testimine

Properties file – omaduste fail

Release – väljalase

Release Candidate – toodangu kandidaat

Sequence diagram – järgnevusskeem

Smoke-test – suitsutest

Staging environment – proovikeskkond

Test-driven development – testidest juhtitud arendus

Value-stream – kasuvoog

Allikad

1. ThoughtWorks Continuous Delivery, http://www.thoughtworks.com/consulting/continuous-delivery, [Viimati vaadatud 11.05.2012].
2. Jez Humble, David Farley, „Continuous Delivery“, Addison Wesley, 2010.
3. Jez Humble, Martin Fowler, „Continuous Delivery“, http://agile2010.agilealliance.org/schedule.html, [Viimati vaadatud 11.05.2012].
4. Jez Humble, „Continuous Delivery“, http://program2011.agilealliance.org/event/492bf007bcfaf60f02a6799717c84ff8, [Viimati vaadatud 11.05.2012].
5. David Farley, „Continuous Delivery“, http://www.devoxx.com/display/DV11/Continuous+Delivery, [Viimati vaadatud 11.05.2012].
6. Agile Alliance, http://www.agilealliance.org/, [Viimati vaadatud 10.05.2012].
7. Mary and Tom Poppendieck, „Implementing Lean Software Development“, Addison Wesley, 2006.
8. Patrick Debois, „What Is This Devops Thing, Anyway?“, http://www.jedi.be/blog/2010/02/12/what-is-this-devops-thing-anyway/, [Viimati vaadatud 13.03.2012].
9. Timothy Fitz, „Continuous Deployment“, http://timothyfitz.wordpress.com/2009/02/08/continuous-deployment/, [Viimati vaadatud 13.05.2012].
10. Jez Humble, „Continuous Delivery: The Value Proposition“, http://www.informit.com/articles/article.aspx?p=1641923, [Viimati vaadatud: 13.05.2012].

11. Rolf Russell, „Introducing Continuous Delivery“, Webinar, http://continuous-delivery.thoughtworks.com/events/introduction-continuous-delivery, [Viimati vaadatud 13.05.2012].

12. „Software release life cycle“ , http://en.wikipedia.org/wiki/Software_release_life_cycle, [Viimati vaadatud 03.04.2012].

13. Nancy R. Taque, „The Quality Toolbox“, Second Edition, ASQ Quality Press, 2004.

14. Paul M. Duvall, Steve Matyas, Andrew Glover, „Continuous Integration“, Addison Wesley, 2007.

15. Brad Stratton, „Gone But Never Forgotten“, Qualiti Progress, märts 1994, http://deming.org/index.cfm?content=654, [Viimati vaadatud 03.04.2012].

16. Brian Marick, „Exploration Through Example“, 2003 http://www.exampler.com/old-blog/2003/08/21/, [Viimati vaadatud 03.04.2012].

17. Lisa Crispin, Janet Gregory, „Agile Testing“, Addison Wesley, 2009.

18. Sam Newman, „A brief and incomplete history of build pipelines“, http://www.magpiebrain.com/2009/12/13/a-brief-and-incomplete-history-of-build-pipelines/, [Viimati vaadatud 08.04.2012].

19. Christopher Read, „CI, Pipelines and Deployment“, http://www.slideshare.net/ChristopherRead/continuous-integration-build-pipelines-and-continuous-deployment, [Viimati vaadatud 08.04.2012].

20. Jez Humble, „Continuous Delivery“ http://continuousdelivery.com/2010/02/continuous-delivery/, [Viimati vaadatud 05.04.2012].

21. Scrum Alliance, http://www.scrumalliance.org/pages/scrum_101, [Viimati vaadatud 05.05.2012].

22. Jenkins, http://jenkins-ci.org/, [Viimati vaadatud 13.05.2012].

23. FindBugs pistikprogramm Jenkinsile, https://wiki.jenkins-ci.org/display/JENKINS/FindBugs+Plugin, [Viimati vaadatud 10.05.2012].

24. PMD pistikprogramm Jenekinsile, https://wiki.jenkins-ci.org/display/JENKINS/PMD+Plugin, [Viimati vaadatud 10.05.2012].

25. Java Compiler Warnings pistikprogramm Jenkinsile, https://wiki.jenkins-ci.org/display/JENKINS/Warnings+Plugin, [Viimati vaadatud 10.05.2012].

26. Task Scanner pistikprogramm Jenkinsile, https://wiki.jenkins-ci.org/display/JENKINS/Task+Scanner+Plugin, [Viimati vaadatud 10.05.2012].

27. SLOCCount pistikprogramm Jenkinsile, https://wiki.jenkins-ci.org/display/JENKINS/SLOCCount+Plugin, [Viimati vaadatud 10.05.2012].

28. TestNG pistikprogramm Jenkinsile, https://wiki.jenkins-ci.org/display/JENKINS/testng-plugin, [Viimati vaadatud 10.05.2012].

29. The Continuous Integration Game pistikprogramm Jenkinsile, https://wiki.jenkins-ci.org/display/JENKINS/The+Continuous+Integration+Game+plugin, [Viimati vaadatud 10.05.2012].

30. Build Pipeline pistikprogramm Jenkinsile, https://wiki.jenkins-ci.org/display/JENKINS/Build+Pipeline+Plugin, [Viimati vaadatud 13.05.2012].

31. H. Vallaste, „E-teatmik – inglisekeelsete info- ja sidetehnoloogia terminite seletav sõnaraamat“, http://www.vallaste.ee/, [Viimati vaadatud 12.05.2012].