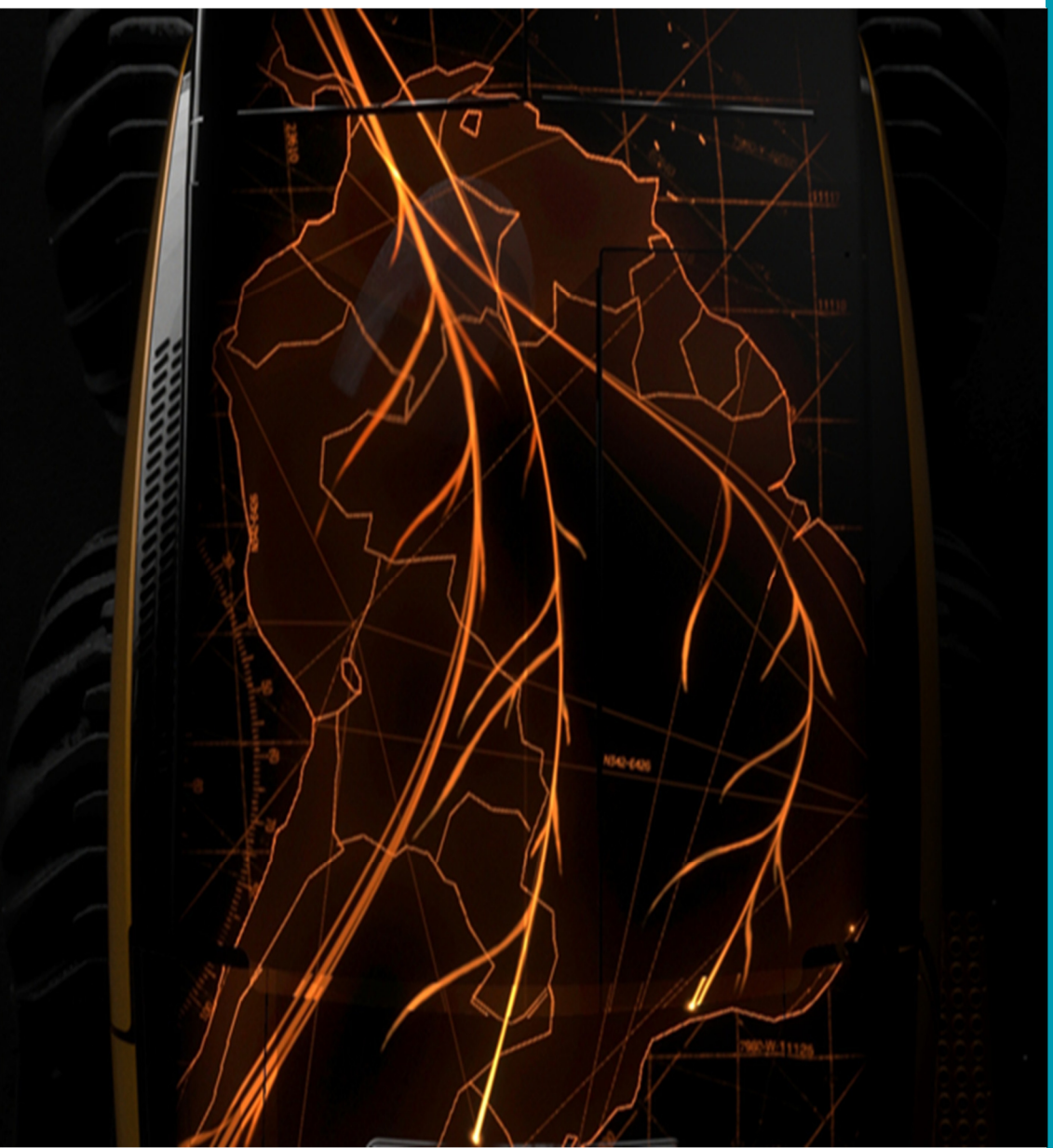
Perttu Moilanen

Testauksen integroituminen ohjelmistokehitys-prosessiin



Teknologiaosaamisen johta-minen

YAMK

Kevät 2020

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Moilanen Perttu

Työn nimi: Testauksen integroituminen ohjelmistokehitysprosessiin

Tutkintonimike: Teknologiaosaamisen johtaminen

Asiasanat: Jatkuva toimitus, Testiautomaatio

Tämän työn tarkoituksena oli löytää raskaita työkoneita valmistavan yrityksen ohjelmistokehitysprosessia tehostavia keinoja, sekä käytäntöjä, joiden avulla testaus voitaisiin paremmin integroida osaksi tätä prosessia. Vaikka yritystä ei itsessään mielletä ohjelmistotaloksi, painottuu merkittävä osa tuotekehitystä kuitenkin erinäisiin ohjelmisto ratkaisuihin. On erittäin todennäköistä, että ohjelmistojen määrä tulevaisuudessa on noususuhdanteinen, joten kannattavuuden ja laadun kannalta ajateltuna testauksen, sekä toimintaa tehostavien käytäntöjen merkitys tulee ainoastaan kasvamaan.

Usein ohjelmistokehityksen tehottomuus johtuu prosessin tuottaman palautteen hitaudesta. Hidas palaute puolestaan johtuu suuresta määrästä manuaalista työtä, jota ei jostain syystä ole saatu automatisoitua. Mitä laajempi järjestelmä, sitä tärkeämpi on automatisoida toistettavissa olevat asiat, jotta ihmiset voivat keskittyä motivoivaan luovaan työhön. Tämän työn puitteissa pyrittiin löytämään syitä prosessin tehottomuuteen, sekä lisäksi tarkoituksena oli löytää niitä keinoja, joiden avulla näitä prosessin tuottoa rajoittavia ”hukka” -tekijöitä voitaisiin minimoida.

Työn teoreettisena viitekehyksenä toimi jatkuvan integroinnin, sekä toimituksen prosessimalli painotettuna testauksen osallisuudella prosessiin. Teoriaosuudella pyrittiin avaamaan prosessimallia yleensä, sekä olennaisia käytäntöjä prosessin toimivuudelle. Vaikka testausprosessi voidaan tietyissä tilanteissa käsittää omana prosessinaan, niin tämän työn puitteissa käsitys siitä pyrittiin liittämään osaksi kokonaisuutta. Prosessimallin puitteissa testaus on osa ketterää kehitystiimiä ja vastuu testatun kokonaisuuden toimittamisesta on yksistään kehitystiimillä.

Tutkimusosa toteutettiin haastatteluiden sekä havainnoinnin avulla. Haastattelut toivat näkemystä prosessin nykytilasta niin työntekijöiden, kuin johdon näkökulmastakin. Merkittävänä seikkana tulosten analysoinnin kannalta olivat yllättävän yhteneväiset näkemykset riippumatta haastateltavan asemasta. Myös havainnointi täydensi haastatteluista saatuja näkemyksiä.

Nykyisellään yrityksen ohjelmistokehitysprosessi noudattaa jatkuvan integroinnin prosessimallia, joten tämän työn puitteissa laadittiin ainoastaan ehdotuksia kehitystoimenpiteistä prosessin tehostamiseksi. Lisäksi luotiin alustava testausstrategia tulevalle ohjelmistosukupolvelle tulevaksi kolmeksi vuodeksi.

Abstract

Author: Perttu Moilanen

Title of the Publication: Integration of testing into the software development process

Degree Title: Master of Engineering, Technology Competence Management

Keywords: Continuous delivery, test automation

The purpose of this work was to get acquainted to the software development process of a company manufacturing heavy work machines, as well as to find good practices and how testing could be more integrated into this process. Although the company itself is not considered as a software house, a significant part of its product development is focused on various software solutions. It is very likely that the amount of software in the future will be further increased, so in terms of quality, the importance of testing will only be pronounced.

Often software development inefficiency is the result of slow development process feedback. Slow feedback is largely result of number of manual works that for some reason have not been automated. The wider the system, the more important it is to automate repetitive things so people can focus on motivating and creative work. The goal of this work was to find the reasons for the inefficiency of the process, as well as to find ways to minimize the “waste” in process limiting its productivity.

The theoretical framework of the work was the process model of continuous integration and continuous delivery, weighted by the involvement of testing in the process. The theoretical part was aimed at opening the process model in general, as well as the essential practices required for the functionality of the process. Although the testing process can be understood as a separate process, the goal of this work was to see testing as a responsibility of whole team. Within the process model, testing is part of the development team and the responsibility for delivering the tested software rests solely with the team.

The research part of the thesis was carried out through interviews and observation. The interviews provided an insight into the current state of the process from the perspective of both employees and management. A significant point for the analysis of the results was that the views were surprisingly consistent regardless of status. The material obtained from the interviews was supplemented with observations, which also confirmed the results obtained through the interviews.

At present, the company's software development process follows a continuous integration process model, therefore, only suggestions for measures to improve the process were made in the work. In addition, a preliminary testing strategy was created for the next software generation for the next three years.

Sisällys

Liitteet

Symboliluettelo

Validointi:	Tarkoittaa prosessia, jonka puitteissa tarkistetaan, että prosessin kohde täyttää tietyt kriteerit [10].
Lean:	Johtamisfilosofia, joka keskittyy poistamaan prosessista ylimääräiset tuottamattomat, prosessia hidastavat toiminnot, jotka tuovat tarpeettomia kustannuksia.
Regressio	Ohjelman toiminta muuttuu tahattomasti esimerkiksi koodimuutosten seurauksena.
Paradigma:	Oikeana pidetty, yleisesti hyväksytty, auktoriteetin asemassa oleva teoria tai viitekehys [12].
Heurestiikka:	Vapaamuotoinen metodi ongelmanratkaisuun, jota käyttämällä päästään nopeasti riittävän lähelle parasta mahdollista lopputulosta [13].
Binääri:	Lähdekoodeista käännetty ohjelma.
Transaktio:	Esimerkiksi tietokantojen yhteydessä tarkoitetaan joukkoa perättäisiä komentoja, jotka tietokantajärjestelmä suorittaa yhtenä kokonaisuutena.
TOC:	Teoria, jonka mukaan jokaisen prosessin tuottavuus voi olla ääretön, mikäli siinä ei olisi rajoitteita.
Tietokantamigraatio:	Tietokannan siirtäminen ohjelman eri versioiden välillä.
Ohjelmametodi:	Olio-ohjelmoinnissa käytettävä nimitys ohjelman aliohjelmalle.

1 Johdanto

1.1 Taustat

Opinnäytetyö tehtiin työkoneiden hallintaohjelmistoja sekä ohjausjärjestelmiä valmistavalle yritykselle. Kehitettävät ohjelmistot koostuvat sulautetuista järjestelmistä, sekä niiden kanssa integroituvasta PC-pohjaisesta työkoneen hallintaohjelmistosta. Ohjelmistojen laajetessa, sekä kehitystiimien henkilöstömäärien kasvaessa, eivät ohjelmistotestausta ohjaavat prosessit ja menetelmät enää täysin pysty vastaamaan yrityksen laatuvaatimusten asettamiin tarpeisiin. Yrityksen visio on olla alan halutuin yhteistyökumppani, joten järjestelmien tehokkaampaan ja kattavampaan testaamiseen on sitoumus yritysjohtoa myöten.

Nopeassa syklissä tapahtuva uusien tuotteiden kehitys sekä nykyisten tuotteiden ylläpito aiheuttavat huomattavaa painetta testaukseen nykyisillä resursseilla. On myös tunnettu tosiasia, että resurssien holtiton lisääminen ei johda haluttuun lopputulokseen, mikäli prosessit eivät ole suunniteltu tai niitä ei noudateta. Tällöin käsissä on ainoastaan suurempi joukko ihmisiä tekemässä asioita ilman selkeää päämäärää. Sitoutumisen tärkeys tekemistä ohjaavaan prosessiin nousikin esille useissa eri vaiheissa opinnäytetyötä.

Työ piti sisällään myös nykyisen ohjelmistotestausstrategian päivittämistä tulevan ohjelmistosukupolven osalta. Käytännön tasolla testausstrategian suunnittelussa oli huomioitava, että tulevaa ohjelmistosukupolvea on tehty jo jonkin aikaa, jolloin tilanne oli lähtökohtaisesti eri verrattuna täysin uudelle projektille laadittavaan testausstrategiaan. Tuote ei myöskään ole vielä ylläpidossa, joten sellaiset seikat, jotka ovat tyypillisiä ylläpidossa oleville projekteille (legacy projects), eivät toisaalta ole vielä ajankohtaisia. Käytännössä työn puitteissa laadittu ehdotus tuotteen testausstrategiaksi on yhdistelmä tyypillisesti uuteen sekä ylläpidossa olevaan projektiin liittyviä asioita.

1.2 Tavoitteet, rajaus ja työn tulokset

Tämän opinnäytetyön tavoitteena oli tutustua ohjelmistokehityksessä vallitseviin uusimpiin käytäntöihin sekä laatia niiden pohjalta prosessimalli, jota noudattamalla kehitystiimi voi luottavaisin

mielin julkaista toteutetun työn ja määritellä sen valmiiksi. Lisäksi tavoitteena oli auttaa kehitystiimejä ymmärtämään testauksen sekä testattavuuden suunnitteluun tärkeys jo ennen varsinaisen kehitystyön aloittamista. Paremman suunnittelun avulla voidaan selkeyttää varsinaista testausta, sekä testiautomaation kehitystä, koska tällöin testauksen eri tasot voidaan huomioida huomattavasti paremmin ja asiat testataan niille luonnollisissa paikoissa.

Työssä oli tarkoitus havainnoida, kuinka vaiheet vaatimusten hahmottelusta ohjelmiston julkaisuun, voidaan solmia sujuvasti yhteen, jotta kehitysprosessista saadaan mahdollisimman tehokas ja luotettava. Hyvin suunniteltujen prosessien avulla ohjelmiston julkaisu voi olla matalan riskitason omaava, ennustettava ja toistettavissa oleva tapahtuma. Mikä tahansa laadittu prosessimalli on turha, mikäli sitä ei noudateta. Eräs tämän työn tärkeimmistä tavoitteista olikin saada kehitystiimit ymmärtämään testaus osana kehitysprosessia sen sijaan, että ”valmiiksi” koodatut työt heitetään testaukseen ja siirrytään seuraavaan tehtävään.

Laajassa moniportaisessa järjestelmässä on myös erittäin tärkeää, että jokainen osa-alue tulee testattua kunnollisesti. Tämä on erityisen tärkeää toimittaessa työkonemaailmassa, jossa ohjelmistovirheestä johtuva koneen arvaamaton liike voi aiheuttaa vakavia vaaratilanteita koneita käyttäville, sekä niiden läheisyydessä oleville henkilöille. Edellä mainituitten syiden vuoksi myös testaukseen liittyviä vastuualueita oli tarkoitus selventää.

Työn teoreettinen viitekehys perustuu jatkuvan integroinnin, sekä toimituksen prosessimalliin ohjelmistotuotannossa, jolloin suunnittelijan tekemän muutoksen tuotantokelpoisuus pyritään varmistamaan ohjelman julkaisuputkessa täysin automaattisesti. Lisäksi teoreettisen viitekehyksen puitteissa tutustutaan pintapuolisesti testauksen eri tasoihin sekä vaiheisiin julkaisuputkessa, sekä testidatan hallintaan. Mukaan teoriaosuuteen on otettu myös kokemuksen kautta hyviksi havaittuja käytäntöjä.

Työn tutkimusosuus tehtiin haastattelemalla organisaation eri tasoilla vaikuttavia henkilöitä sekä luomalla kuva prosessin nykytilasta perustuen näihin haastatteluihin. Haastatteluiden sekä aiheeseen liittyvän teorian pohjalta määriteltiin esitys prosessia tehostavista toimenpiteistä sekä aliprosesseista. Lisäksi työn puitteissa pyrittiin selkeyttämään prosessin keskeiset roolit, sekä näiden roolien vastuut.

Tämän työn puitteissa ei syvennytä erinäisiin testaustekniikoihin, vaan pääpaino on prosessimallin kokonaisvaltaisessa kehittämisessä painottuen testauksen integrointiin osaksi kehitystyötä. Käytettävän prosessimallin tarkoituksena on pystyä toimittamaan toimivaksi testattua ohjelmistoa mahdollisimman ketterästi siten, että toimitusvastuu on kehitystiimeillä itsellään.

2 Ohjelmistoprosessit sekä toimituksen keskeiset periaatteet

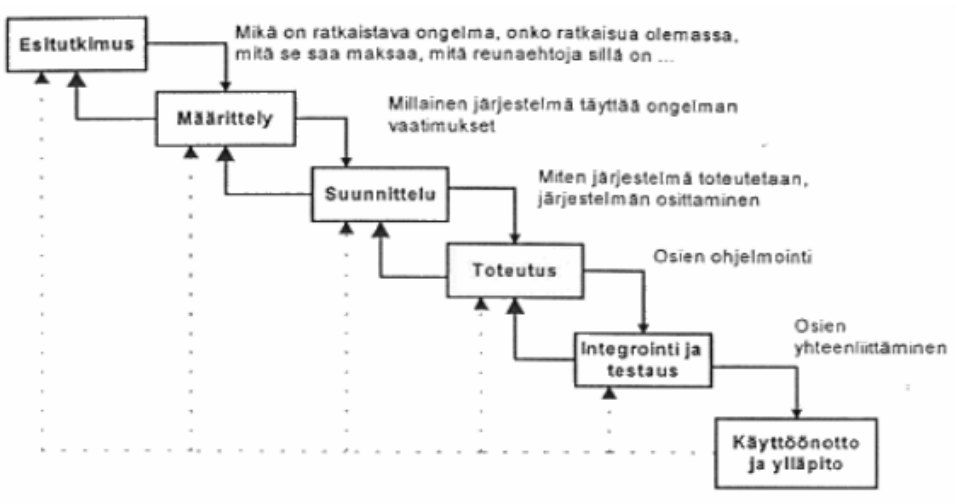
Ohjelmistojen toimitukseen liittyvä tärkein kysymys kuuluu, että kuinka ohjelmistokehitysprosessin täytyy toimia, jotta uusi idea on jalostunut ohjelmistossa käytettäväksi ominaisuudeksi mahdollisimman nopeassa ajassa. Mitkä ovat kehitysprosessin aikaiset vaiheet, kun vaatimukset on tunnistettu, ja vaatimukset toteuttava koodi on tehty ja testattu. Kuinka nämä vaiheet nivoutuvat toisiinsa ja kuinka prosessi täytyy toteuttaa, jotta kaikki siihen osallistuvat voivat tehdä työnsä tehokkaasti vuorovaikutuksessa keskenään? [1.s.3]

Eräs ohjelmistojen julkaisuun liittyvä vahva perinne on tietynlainen julkaisupäivän lähestymisestä johtuva kireys, jännitys sekä pelko tulevaa tapahtumaa koskien. Yleensä tämä johtuu ohjelmistokehityksen prosessimallista, joka mahdollistaa julkaisuun liittyvän korkean epäonnistumisen riskin. Tunnusomaisia merkkejä huonolle prosessimallille ovat esimerkiksi suuri määrä manuaalisia työvaiheita, siiloutuminen sekä tehtävien pallottelu osastojen välillä. Tehottoman prosessimallin vuoksi voidaan helposti ajautua tilanteeseen, jossa julkaisupäivän koittaessa kukaan ei tiedä miksi ohjelma ei toimi, koska on niin paljon muistettavaa ja jotain on voinut jäädä tekemättä. [1.s.3]

2.1 Perinteiset ohjelmistokehityksen prosessimallit

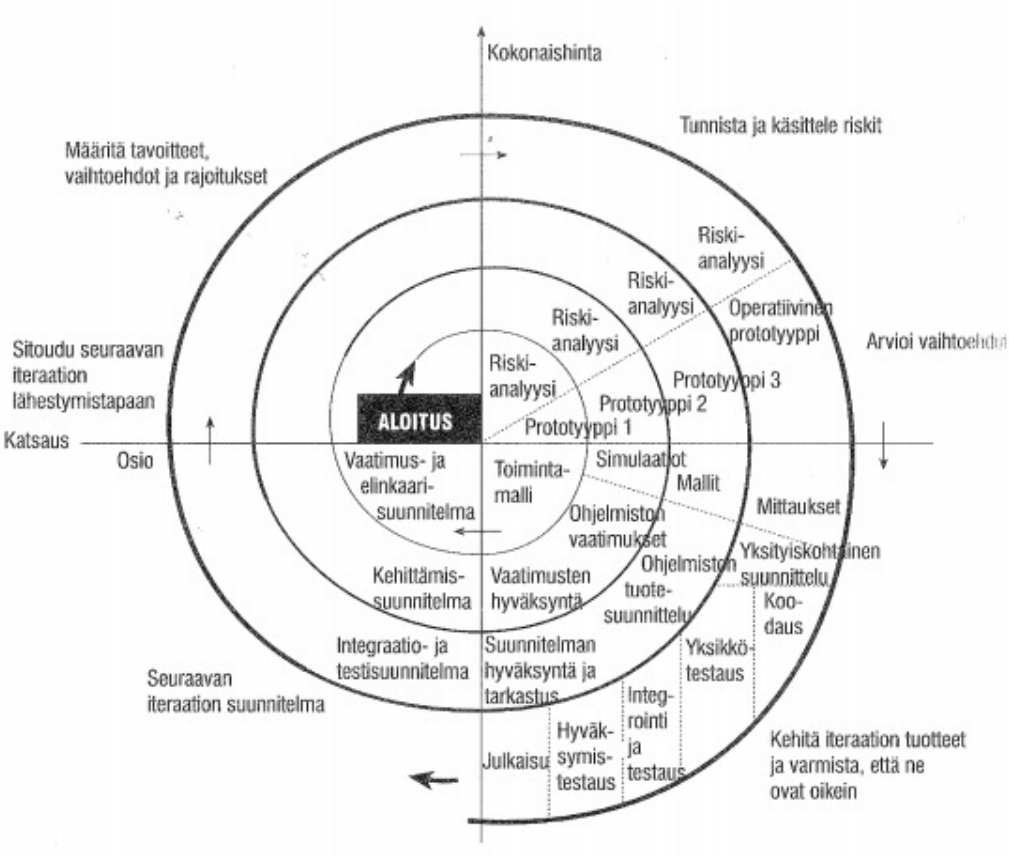
Perinteisinä ohjelmistokehitysmenetelminä voidaan pitää esimerkiksi vesiputousmallia, joka perustuu lineaariseen vaiheittaiseen etenemiseen, kunnes viimeinenkin vaihe on suoritettu. Siirtyminen seuraavaan vaiheeseen voidaan tehdä vasta, kun edellinen vaihe on todettu täysin toimivaksi sekä siitä on laadittu vaadittava dokumentaatio. [4.s.45]

Ohjelmistokehityksen suhteen vesiputousmalli on saanut osakseen paljon kritiikkiä sen soveltumattomuudesta projekteihin, joiden vaatimukset eivät ole tiedossa projektin alkuvaiheessa. Mallin orjallisen noudattamisen seurauksena testausvaiheeseen voidaan siirtyä vasta suhteellisen myöhäisessä vaiheessa, sekä dokumentointi vaatii suhteellisen paljon työtä verrattuna muihin toimintoihin. Perinteinen vesiputousmalli on esitetty kuvassa 1. [2.s.37, 5.s.46]



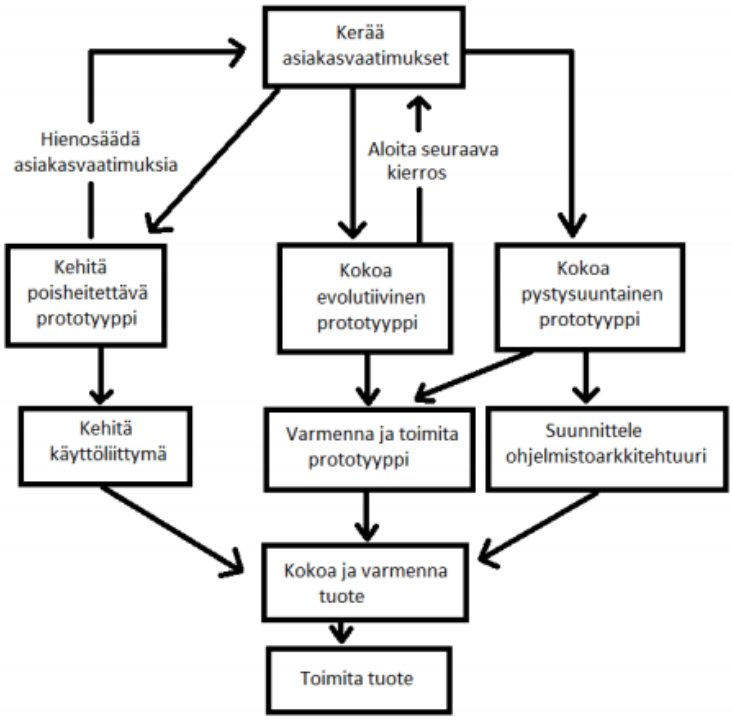
Kuva 1. Vesiputousmalli [3.s.36]

Vesiputousmallin lisäksi on olemassa myös muita perinteisiksi kehitysmenetelmiksi luettavia prosessimalleja, kuten esimerkiksi spiraalimalli tai prototyyppimalli. Kuvassa 2 esitetty spiraalimalli perustuu iteraatiokierroksiin, jossa seuraava kierros on aina edellistä suurempi. [5. s.53]



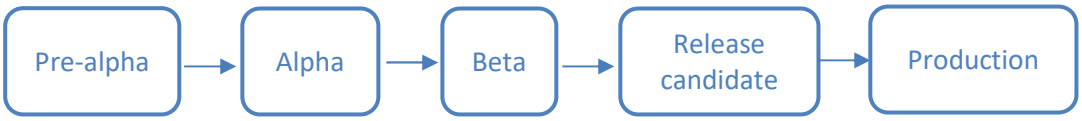
Kuva 2. Ohjelmistokehityksen spiraalimalli [6.s.142]

Kehitysprosessina prototyyppimalli perustuu kolmeen eri päämalliin. Ensimmäinen malli on selvittää ja täydentää asiakasvaatimuksia, koska prototyyppien käyttämisellä sekä esittelyllä voidaan paljastaa puutteita asiakasvaatimuksissa. Prototyypeillä voidaan lisäksi selvittää eri toteutusvaihtoehtojen välisiä eroja vertailemalla niitä keskenään. Prototyypistä voidaan myös kasvattaa asteittain valmis tuote asiakaspalautteen perusteella. Kuvassa 3 on esitettynä prototyyppimallinen kehitys useita eri polkuja hyödyntäen. [7. s.234, 236]



Kuva 3. Ohjelmistokehityksen prototyyppimalli [7. s.239]

Wikipediassa ohjelmiston julkaisun elinkaarta kuvataan perinteiselle ohjelmistoprojektille tyypilliseen malliin. Tässä mallissa käännöstä pidetään julkaisukandidaattina vasta, kun se on läpäissyt kaikki prosessin eri vaiheet kuvan 4 mukaisesti. Perinteisessä ohjelmistoprosessissa myös projektin onnistumisen kannalta tärkeät, palautetta tuottavat vaiheet ovat sijoitettuna prosessin loppupuolelle.



Kuva 4. Julkaisukandidaatti prosessin omana vaiheena [11]

2.2 Automaattinen julkaisuputki osana ketterää kehitystä

Mikäli ollaan tilanteessa, jossa sovellettu prosessimalli tuottaa tehottomasti korkean riskitason omaavia julkaisuja, tulee meidän miettiä vaihtoehtoisia prosessimalleja ohjelmiston julkaisua silmällä pitäen. Meidän tulee miettiä keinoja, joiden avulla edellisestä poiketen ohjelmiston kehityksestä ja julkaisusta saadaan matalan riskitason, nopeaa, tehokasta ja toistettavissa olevaa toimintaa. Eräs sovellettava, korkean automaatiotason omaava prosessimalli on automaattinen julkaisuputki, joka on automatisoitu toteutus ohjelmiston käännös-, toimitus-, testaus sekä julkaisuprosesseista [1. s 3]. Esimerkkikuva julkaisuputkesta on esitetty kuvassa 5.

Kuva 5. Esimerkki julkaisuputkesta

Automaattisella julkaisuputkella tähdätään ketterän ohjelmistokehityksen tärkeimpään periaatteeseen, jonka mukaan asiakkaalle halutaan toimittaa tämän tarpeet täyttäviä ohjelmistoversioita mahdollisimman nopeasti ja säännöllisesti [29].

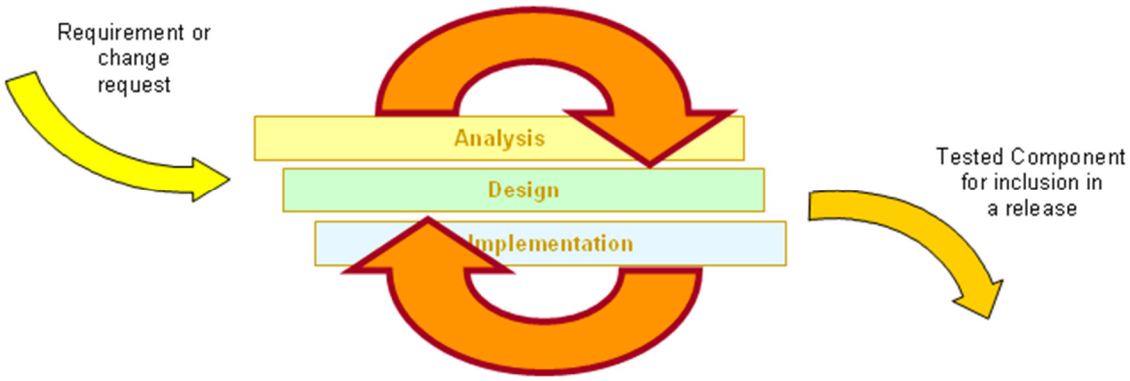
Automatisoidun julkaisuputken prosessimallilla on kolme päätavoitetta. Ensiksi jokainen vaihe käännösputkessa (kääntäminen, toimitus, testaus sekä julkaisu) tulee olla läpinäkyvä kaikille kehitysprosessien parissa toimiville henkilöille paremman yhteistyöhön aikaansaamiseksi. Toiseksi halutaan parantaa organisaation prosessista saamaa palautetta, jotta ongelmat voidaan paikantaa ja korjata mahdollisimman aikaisessa vaiheessa. Kolmantena tavoitteena on antaa kehitystiimeille mahdollisuus ottaa käyttöön ja julkaista ohjelmaan tehdyt muutokset täysin automatisoidusti [1. s 4].

Tärkeä osa ohjelmiston toimivuutta kokonaisuudessaan on laatu, sekä ohjelman kyky toteuttaa sille määritellyt tehtävät. Laatu ei kuitenkaan ole sama asia, kuin täydellisyys, joka usein onkin ”vihollinen” järkevän hyötysuhteen omaavalle toimitukselle. Tärkeää onkin asettaa ohjelman laatukriteerit siten, että niiden puitteissa ohjelma on käyttökelpoinen käyttäjille, mutta niistä ei aiheudu lisäarvoa tuottamatonta työtä. Ohjelmiston laatuun vaikuttaa merkittävästi mahdollisuus julkaista sitä automaattisesti riittävän usein. Mikäli julkaisuprosessi ei ole automaattinen, se ei ole luotettavasti toistettavissa ja lisäksi tällöin myös altis virheille. Tiheällä syklillä tehtävien jul-

kaisuiden ansiosta versioiden väliset muutokset pysyvät pieninä, ja mahdollisuus palata edeltävään versioon on huomattavasti helpompaa. Eräs ketterän kehityksen periaatteista on julkaista ohjelmaa tiheällä syklillä [29]. Jotta tämä olisi mahdollista, prosessin tulee tuottaa palautetta mahdollisimman nopeasti jokaisesta tuotteeseen kohdistuvasta muutoksesta. [1. s 12]

2.2.1 Iteratiivinen prosessi sekä siihen liittyvät roolit ja vastuut

Ketterän kehityksen periaatteisiin kuuluu myös, että liiketoiminnan edustajien ja ohjelmistokehittäjien tulee työskennellä yhdessä koko projektin ajan [29]. Iteratiivisesta ohjelmistokehityksestä (kuva 6) onkin paljon hyviä kokemuksia, koska se auttavaa hahmottamaan liiketoiminnan edustajien, kehittäjien sekä testaajien roolit. Liiketoiminnan edustajien ja testaajien välinen yhteistyö tehostaa prosessia, koska testaajilla on usein näkemys esimerkiksi mittauksista, joiden avulla tehty työ voidaan määritellä valmiiksi. Testaajat puolestaan hyötyvät siitä, että saavat vaatimuksesta kokonaisvaltaisen ymmärryksen ennen kuin se varsinaisesti tulee työn alle. Vaikka prosessin toiminnan kannalta oleelliset roolit (Liite 8) eivät aina henkilöidy 100% tiettyyn henkilöön, on tärkeää, että ne kuitenkin ovat mukana kehitystyössä. [1.s.194]



Kuva 6. Iteratiivinen kehitysmalli [27]

Merkittävä osa prosessia ovat laadukkaat hyväksymiskriteerit. *Hyväksyntätestit* periytyvät suoraan *hyväksymiskriteereistä*, joten hyväksymiskriteerit on hyvä kirjoittaa suoraan testiautomaatiota tukeviksi. (katso liite 6, kohdat INVEST ja Gherkin) Hyväksyntätestaus itsessään toimii tärkeänä ajurina tämän periaatteen noudattamiselle, painostaen hienovaraisesti luomaan parempia vaatimuksia testauksen tueksi. Huonosti kirjoitetut hyväksymiskriteerit ovatkin eräs syy huonoille ja hankalasti ylläpidettäville automaattitesteille. Kun vaatimus kertoo selkeästi sen käyttäjälle tuoman lisäarvon, on se tällöin helppo todentaa automaattitestillä. Jokaista hyväksymiskriteeriä

vastaan on hyvä kirjoittaa vähintään yksi automaattitesti, joka varmistaa, että määritelty toiminnallisuus voidaan toimittaa asiakkaalle. [1.s.93; 190-191]

Usein luotetaan, että testaajat tutkivat uudet ominaisuudet kaikkine mahdollisine skenaarioineen ja suunnittelevat niille monimutkaiset ja kattavat testit. Edellä mainitun sijaan toimiva malli on kutsua koolle oikeat henkilöt tarvittavista sidosryhmistä työn tullessa suunnitteluun, jolloin tarvittavan asiantuntemuksen turvin voidaan tunnistaa tärkeimmät testattavat skenaariot sekä luoda hyväksymiskriteerit. [1.s.99]

Hyväksyntätestit ja lyhyet kuvaukset niiden kohteista toimivat kehitystyön lähtökohtana. Kehittäjien ja testaajien onkin hyvä keskustella mahdollisimman aikaisin hyväksyntätesteistä ennen varsinaisen kehityksen aloittamista. Tämä auttaa kehittäjiä ymmärtämään ominaisuutta, sekä sen tärkeimpiä skenaarioita kokonaisuudessaan. Lisäksi malli vähentää ohjelmistovikoja ja testaajien palautteesta aiheutuvaa kehityssyklien määrää. Mallin avulla voidaan lisäksi minimoida riskiä jossa, kaikkea tarvittavaa toiminnallisuutta ei ole implementoitu mukaan toteutukseen. Vastaavasti töiden siirtäminen kehityksestä testaukseen vasta varsinaisen ohjelmointityön valmistuttua voi muodostaa pullonkaulan, mikäli testaus löytää puutteita toiminnallisuudesta kehittäjien jo työstäessä seuraavaa tehtävää. [1.s.99]

Kun hyväksymiskriteerit on määritelty, mutta kuitenkin ennen varsinaisen kehitystyön aloitusta, on liiketoiminnan edustajien, testaajien sekä kehittäjäien hyvä vielä kerran katselmoida vaatimukset, niiden käyttötapaukset, sekä vaatimuksiin liittyvät hyväksymiskriteerit. Lisäksi testaajien tulee kehittäjien kanssa katselmoida testit, joiden avulla hyväksyntäkriteerit voidaan katsoa täytetyiksi. Näiden nopeiden tapaamisten avulla voidaan varmistaa, että jokaisella kehitykseen osallistuvalla taholla on riittävä ymmärrys vaatimuksesta sekä omasta osuudesta sen toteuttamiselle. Tämä toimintamalli estää liiketoiminnan edustajia tekemästä vaatimuksia, jotka ovat kalliita toteuttaa tai testata. Se estää testaajia kirjoittamasta ymmärryksen puutteesta johtuvia vikaraportteja. Kehittäjiä malli puolestaan estää kirjoittamasta toiminnallisuutta, joita kukaan ei ole pyytänyt. [1.s.194]

Kehitysprosessin aikana kehittäjät konsultoivat liiketoiminnan edustajia epäselvissä tapauksissa ja kun toiminto mielletään valmiiksi, esitellään se liiketoiminnan edustajille, testaajille ja asiakkaille. Tässä vaiheessa asiakas voi vahvistaa, että ominaisuus täyttää sille annetut vaatimukset. Kun ominaisuus on hyväksytty asiakkaan toimesta, voi se seuraavaksi siirtyä esimerkiksi kenttätestaukseen. [1.s.194-195]

Helpoin vaihe iteratiivisen prosessin käyttöönotolle on heti projektin alussa. Muutaman iteraatio kierroksen jälkeen esimerkiksi testaukseen liittyvien työkalujen käyttöönotto on huomattavasti vaikeampaa, kuten myös suunnittelijoiden vakuuttaminen prosessin noudattamisen tärkeydestä. On myös erittäin tärkeää, että sekä tiimi, että asiakas on sitoutunut noudattamaan prosessia ja sen hyödyt on ymmärretty. [1.s.93]

2.2.2 Prosessin mittaaminen

Prosessin mittaamisen tulee olla oikein kohdennettua sekä jatkuva-aikaista. Koska ketterien ohjelmistokehitysmenetelmien käyttö tukee ohjelmiston hyvän rakenteen ja teknisen laadun jatkuvaa kehittämistä, soveltuvat ne erityisen hyvin tämän päivän ohjelmistokehitykseen. [29]. Yksittäisten asioiden sijaan tulee keskittyä lean -ajattelun mukaisesti globaaliin optimointiin, jolloin mittarit tulee laatia siten, että niiden avulla voidaan paikantaa ongelmapaikkoja koko toimitusprosessissa. [1.s.137-138]

Ohjelmistokehitysprosessissa tärkein mittari on sykliaika, joka kiteytyykin ajatukseen; Kuinka kauan kestää yhden rivin muutoksen toimittaminen asiakkaalle ja tehdäänkö se luotettavasti toistettavissa olevien rutiinien avulla. Sykliajan mittaus käynnistyy, kun on päätetty ominaisuuden lisäämisestä järjestelmään. Vastaavasti kello pysähtyy, kun ominaisuus on julkaistu asiakkaalle. Vaikka sykliaikaa onkin vaikea mitata, kertoo se paljon prosessista ja keskittyminen sitä lyhentäviin toimenpiteisiin auttavat parantamaan prosessia kokonaisuudessaan. Mittaamalla ohjelmistovikojen määrää, voidaan se kyllä saada tietoon, muttei se välttämättä auta kehittämään prosessia. Mikäli esimerkiksi löydetyn ohjelmistovian korjauksen julkaisemiseen menee puoli vuotta, ei vian näkyminen mittarissa välttämättä hyödytä ketään. [1.s.138]

Sykliajan ja ohjelmistovikojen määrän lisäksi muita prosessin laadusta kertovia mittareita ovat esimerkiksi: [1.s.139]

- Automaattitestien testikattavuus
- Koodistatistiikat
- Nopeus, jolla tiimi tuottaa käyttöönotettavaa koodia
- Versionhallintaan tehtävien muutosten lukumäärä päivässä
- Käännösten määrä päivässä

- Käännöksen valmistumiseen kuluva aika sisältäen automaattitestit

Nykyaikaisten CI-työkalujen avulla voidaan vaiheistettu läpimenoaikojen mittaaminen toteuttaa kohtuullisen helposti, jolloin pullonkaulat ovat helpommin paikannettavissa. Kun läpimenoaika tunnetaan, voidaan seuraavassa vaiheessa tehdä toimenpiteitä sen pienentämiseksi. Läpimenoaikojen minimointiin voidaan soveltaa esimerkiksi TOC-prosessia (theory of constraints), jonka mukaan jokaisen prosessin tuottavuus voi olla ääretön, mikäli siihen ei kohdistu rajoitteita. [1.s.138]

TOC-prosessin soveltaminen

Ensimmäinen vaihe prosessissa on tunnistaa prosessin sujuvuutta rajoittavat tekijät. Julkaisuputkessa tämä voisi helposti olla manuaalitestausvaihe. Seuraavaksi tulee pyrkiä hallitsemaan ja nopeuttamaan tunnistettuja prosessia hidastavia tekijöitä. Manuaalitestausta voidaan esimerkiksi nopeuttaa organisoimalla töitä ja varaamalla tarvittavat resurssit tilanteeseen, kun niitä tarvitaan. [1.s.138]

Kokonaisprosessia voidaan myös skaalata hitaimpien prosessinosien mukaan. Tämä voi käytännössä tarkoittaa, että kehittäjät eivät tuota jatkuvalla syötöllä työtä testauksen paisuvalle työlistalle, vaan sen sijaan uuden kehittämistä tehdään sen verran, että testaus pysyy perässä ja loppuaika tuetaan testausta kirjoittamalla automaattitestejä, jotka osaltaan vielä pienentävät testauksen kuormaa. [1.s.139]

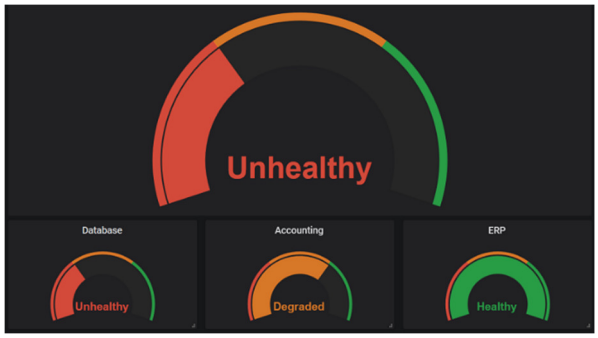
Mikäli edelliset vaiheet eivät tuo helpotusta tilanteeseen ja läpimenoaika on edelleen liian pitkä, voidaan pullonkauloja yrittää poistaa lisäresurssein tai sijoittamalla esimerkiksi enemmän testiautomaation kehittämiseen. Kun edellä mainitut kohdat on toteutettu, voidaan ottaa työnalle seuraavaksi pahin pullonkaula ja siirtyä kohtaan 1. [1.s.139]

2.2.3 Prosessin tuottama palaute

Syynä hitaasti saatavaan palautteeseen on yleensä manuaalisesti tehtävä työ, joka on henkilöriippuvaista sekä hankalasti auditoitavaa. Tehokas palaute prosessista perustuukin näin ollen puhtaasti automaatioon. Nopein palaute saadaan julkaisuputken ensimmäisestä vaiheesta (commit stage), jonka puitteissa ohjelma käännetään, sekä sille suoritetaan nopeat ensivaiheen testit sekä

koodianalyysit. Tämän vaiheen testit tuleekin suunnitella siten, että niiden testikattavuus on vähintään 75% ja mikäli jokin testeistä ei mene läpi, pysäyttää se julkaisukandidaatin (release candidate) etenemisen seuraaviin vaiheisiin. Tässä vaiheessa tuleekin testata ainoastaan toiminnan kannalta kriittisiä asioita. Testit eivät myöskään saa vaatia ohjelmiston lopullista toimintaympäristöä, vaan niiden tulee olla mahdollisimman pitkälle ympäristöriippumattomia. Perus palvelinympäristössä ajettujen ensimmäisen vaiheen testien jälkeen, voidaan ohjelmiston toiminnasta saada nopeasti tietynlainen peruskäsitys. [1. s15]

Prosessin tehtävä on myös varmistaa, että palaute on toimitusprosessiin osallistuvien henkilöiden saatavilla. Palautteen välittämistä tiimeille voidaan tehostaa esimerkiksi joko manuaalisilla, tai sähköisillä infotauluilla. Kuvassa 7 on esitetty kuvitteellinen esimerkki tiimihuoneen infotaulunäkymästä, jonka tarkoituksena on indikoida ohjelmiston sen hetkistä tilaa. Palautteen asianmukaista käsittelyä edesauttaa vahva vuorovaikutus tiimien kesken. [1.s.16]



Kuva 7. Kuvitteellinen esimerkki ohjelmiston tilaa kuvastavasta infotaulusta

Ideaalisessa tilanteessa ohjelmistovirheet eivät koskaan päädy testaajille, saati loppukäyttäjille, mutta käytännössä näin käy väistämättä jossain vaiheessa ohjelman elinkaarta. Tällöin löydettyjen ohjelmistovikojen tulee päätyä tiimin työlistalle, jossa niiden etenemistä voidaan seurata.

Käytännöllinen lähestymiskulma on pitää tehtävät työt sekä korjattavat viat samalla työlistalla. Tällöin voidaan helposti arvioida vikojen korjaamiseen sekä uusien töiden tekemiseen tarvittava työmäärä kokonaisuudessaan. Esimerkiksi toimittaessa Scrum –viitekehyksen puitteissa, töiden priorisointi on tällöin huomattavasti helpompaa, koska sprinttiin voidaan nostaa tietty määrä töitä yhdeltä työlistalta. Listalla olevien ohjelmistovikojen priorisointi esimerkiksi kriittisiksi, blokkaaviksi tai näitä pienemmillä prioriteeteilla auttaa osaltaan hahmottamaan niiden tärkeyttä verrattuna muihin listalla oleviin töihin. [1.s.100-101]

2.2.4 Prosessin tuottamat hyödyt

Yksi julkaisuputken perusperiaatteista on sen perustuminen lean -mallista tuttuun ”pull system” –periaatteeseen, jossa minimoidaan turha työ ja tuotetaan asiakkaalle pelkästään tämän tilaama tuote. Esimerkiksi testaaja voi tällöin oma-aloitteisesti valita listalta haluamansa ohjelmaversion ja toimittaa sen haluamaansa ympäristöön napin painalluksella. Tämän kaltaisesta järjestelmästä on paljon hyötyä, koska tiettyyn ohjelmaversioon kohdentuvat toimenpiteet on mahdollista toteuttaa toistettavasti ilman turhaa odottelua. [1. s.17]

Julkaisuputken laadunvarmistukseen liittyy olennaisesti myös konfiguraatioiden hallinta koska varsinaisen ohjelmiston lisäksi sen toimintaan voi liittyä huomattava määrä erilaisia konfiguraatiotiedostoja. Näiden automaattinen versiointi, hallinta sekä palauttaminen on välttämätöntä, mikäli halutaan ennaltaehkäistä ja tutkia konfiguraatioista johtuvia, hankalasti toistettavia virheitä. Tuotantoympäristössä käytettävien konfiguraatioiden lisäksi myös testiympäristöjen konfiguraatiot tulee pitää versionhallinnassa, koska ainoastaan tällöin voidaan luottaa testien toistettavuuteen. [1. s.18-20]

Eräänä automatisoidun julkaisuprosessin merkittävimpänä etuna voidaan pitää sen mukanaan tuomaa varmuutta ohjelman julkaisuun. Manuaalisesti aika-ajoin julkaistavien ohjelmien julkaisuhetkeen liittyy usein paljon stressiä ja mahdollisesti myös viime hetkellä tapahtuvia ”yhden rivin” –muutoksia, joiden ei pitäisi vaikuttaa mihinkään. Sen lisäksi julkaisu edellyttää usein tiettyä listaa toimenpiteitä, joiden oikeaoppinen suorittaminen tietyssä järjestyksessä on edellytyksenä julkaisun onnistumiselle. Pitkien julkaisuvälien takia muutosten määrä kasvaa usein myös huomattavan suureksi. Automatisoidussa mallissa julkaisuiden välit voidaan pitää pieninä ja näin ollen myös palauttaminen edelliseen versioon käy huomattavasti helpommin. [1. s.20-21]

2.2.5 Jokainen muutos projektissa luo potentiaalisen uuden version

Mikäli ohjelmaan tehdään muutos, tulee tieto sen julkaisukelpoisuudesta saada mahdollisimman pian. Ilman toistettavissa olevaa käännös, toimitus sekä testausprosessia vastaus julkaisukelpoisuudesta on hyvin epävarma ja laajempien ohjelmistokokonaisuuksien kohdalla käytännössä täysin arpapeliä. [1. s.22-23]

Luvussa 2.1 esiteltyjen perinteisten mallien sijaan kehitysprosessi on hyvä ajatella jatkumona, jossa automaattisesti käännetyt koodit toimitetaan automaattisesti haluttuun ympäristöön, ja

jossa käännöksen laadusta vastaa kattava testiautomaatio. Tällöin projektin loppumetreillä manuaalinen testaus voi painottua tutkivaan testaukseen sen sijaan, että sen päätarkoitus on etsiä mahdollisia regressioita tai puhtaita ohjelmistovikoja, mikä itsessään on kallista, epävarmaa ja aikaa vievää. Testausta ei kannata jättää kehitysprosessin loppuun, koska vikojen korjaus on tehokkainta siellä, missä ne syntyvät. Pahimmillaan julkaisun kynnyksellä löytyneen vian kunnolliseen korjaamiseen ei ole aikaa ja toteutettu korjaus lisää ainoastaan järjestelmän teknistä velkaa. [1. s.23]

Vaihetta, jossa muutokset lisätään järjestelmään, kutsutaan yleisesti integraatioksi ja se on usein ennalta-arvaamattomin ja hallitsemattomin vaihe kehitysprosessia. Lähtökohtaisesti voidaan olettaa, että integraatio on aiheuttanut tahattomia muutoksia ohjelmaan, mikäli ohjelmistokokonaisuutta ei ole testattu asianmukaisesti. Pelko ohjelman särkymisestä voi vähentää säännöllistä integraatiota, mikä yleensä ainoastaan pahentaa tilannetta. Seikka voi korostua entisestään toimittaessa, kun ohjelmistoa kehitetään useiden tiimien voimin. Parhaillaan integrointi tehdään jokaisesta muutoksesta (Continuous integration), mikä nykyisellään onkin ohjelmistoalalla vallitseva paradigma. Ideaalisessa tilassa jatkuvan integroinnin avulla voidaan huomata jokainen systeemin rikkova, tai vaatimusten vastainen muutos. Kehitystiimien vastuulla on tällöin korjata ongelma heti sen ilmennyttyä, mikä on jatkuvan integroinnin ensimmäinen sääntö. Noudattamalla jatkuvan integroinnin prosessimallia, voidaan ohjelmisto pitää aina toimintakuntoisena. [1. s.23-24]

2.3 Ohjelman toimituksen keskeisimmät periaatteet

Ohjelman laadukas julkaiseminen tarvitsee taustalleen automatisoidun, luotettavan ja toistettavissa olevan prosessin. Toistettavuus ja luotettavuus periytyvät korkeasta automatisoinnin tasosta sekä tarvittavien asioiden versioinnista. Ohjelmiston toimittaminen sen lopulliseen käyttöympäristöön kulminoituu seuraaviin päätekijöihin. [1. s .24 - 25]

1. On tiedostettava ympäristö, jossa ohjelmaa suoritetaan
2. On tiedostettava versio, joka halutaan toimittaa
3. Ohjelman konfigurointi on tehtävä hallitusti

Automatisoinnin suhteen voidaan yleisesti ajatella, että asioita voidaan automatisoida, kunnes vastaan tulee tarve inhimilliseen päätöksentekoon. Usein automatisoinnin puutteelle on syynä siirtyminen pois omalta mukavuusalueelta. Automatisoinnin kynnystä ei usein myöskään laske se, että ensimmäisellä ja toisella kerralla prosessin manuaalinen suorittaminen voi tuntua täysin tehtävissä olevalta. Prosessin näkökulmasta katsottuna voidaan kuitenkin sanoa, ettei prosessia voi toteuttaa napin painalluksella, mikäli tehtäviä ei ole automatisoitu. [1. s.25]

Kaikki ohjelman julkaisuputkessa tarvittavat asiat tulee olla versionhallinnassa. Kunnollinen versionhallinta luo lisäksi puitteet tehokkaaseen työskentelyyn. Tällöin esimerkiksi versionhallintaan voidaan luoda tietty kehityshaara (branch), josta ohjelmakoodit voidaan ottaa omalle työasemalle editoitavaksi (check-out). Muutosten jälkeen ohjelmaversio voidaan toimittaa haluttuun ympäristöön esimerkiksi käynnistämällä automatisoitu toimitusprosessi (deployment) napin painalluksella. [1. s.26]

Toimitettavan ominaisuuden valmiiksi määrittelemisen täytyy aina tarkoittaa, että työn puitteissa on valmistunut jokin käyttäjälle lisäarvoa tuova julkaisukelpoinen ominaisuus. Käytännössä ohjelmisto ja sen ominaisuudet kuitenkin elävät sen elinkaaren aikana ja toimitettavan ominaisuuden ”valmiiksi” määritteleminen voikin tarkoittaa käyttäjien edustajan hyväksyntää ominaisuudelle tuotantoympäristöä vastaavassa ympäristössä. Ominaisuuden valmiiksi määrittäminen ei saa tarkoittaa, että tietty henkilö on sen toteuttanut ja kuitannut tehdyksi. Todellisuudessa kehitystiimi on vastuussa ominaisuuden valmistumisesta ja sen julkaisemisesta tai julkaisun viivästymisestä. [1. s.27-28]

Jatkuvan parantamisen periaatteen mukaan on tärkeää muistaa, että ensimmäinen julkaisu on ensimmäinen askel ohjelmiston elinkaaressa. Ohjelmasta tulee uusia versioita, joiden mukana myös toimitusprosessin tulee kehittyä ja sitä täytyykin arvioida säännöllisesti esimerkiksi palautetilaisuuksissa, joihin osallistuvat kaikki toimitusprosessiin osallistuvat henkilöt. Siiloutumisen sijaan palautteen tulee olla koko toimitusorganisaation saatavilla, jolloin pienille osa-aleille kohdistuvat, siilojen sisäiset optimoinnit voidaan muuttaa koko organisaatiota ja prosessia koskeviksi. [1. s.28-29]

3 Jatkuva integrointi (CI)

Kummallinen, mutta yleinen tunnusmerkki useille ohjelmistoprojekteille on, että kehityksen alla oleva ohjelma on toimintakuntoinen vain pienen osan kehitysprosessin kestosta. Tilanteeseen voidaan helposti päätyä, mikäli esimerkiksi suunnitteluvaiheessa ohjelma testataan tuotantoympäristöä vastaavan ympäristön sijaan pelkästään yksikkötasolla. Tilannetta voi entisestään pahentaa pitkään elossa olevat kehityshaarat, joiden hyväksyntätestaus tehdään vasta kun kehityshaara yhdistetään versionhallinnan päähaaraan. Jatkuvan integroinnin ajatuksena onkin välttää tämänkaltaisia tilanteita, jolloin versionhallintaan tehtävät lisäykset ovat pieniä, ja niistä käynnistyy koko ohjelmistoa testaavat hyväksyntätestit. [1. s.55 - 56]

3.1 Prosessin käyttöönotto ja toiminnan edellytykset

Mallina jatkuva integrointi voidaan ottaa käyttöön hyvin vähäisin vaatimuksin. Ensimmäinen ja tärkein asia on versionhallinta, jonne tallennetaan kaikki kehitykseen liittyvä. Lisäksi ohjelma on voitava kääntää automaattisesti esimerkiksi komentorivityökaluilla. Käännösskriptit ovat automaation kannalta käteviä sekä suositeltuja, koska niiden avulla halutut asiat voidaan toteuttaa riippumatta käännösympäristöstä. Skriptit ovat helposti auditoitavissa ja ne voidaan pilkkoa tekemään halutut tehtävät. [1. 56-57]

Lisäksi tarvitaan palvelinohjelmisto, jonka tehtävänä on huomata versionhallintaan tehdyt muutokset, kääntää niistä uusi ohjelmaversio, suorittaa testit ja lopuksi raportoida tulokset prosessista. Palvelinohjelmisto voi esimerkiksi sisältää kaksi pääkomponenttia, joista ensimmäinen hoitaa kääntämisen sekä testaamisen ja toinen tulosten raportoinnin niitä tarvitseville tahoille. [1.s.63]

Parhaillaan CI-putki voi olla alusta tiedon sekä statistiikan julkaisemiseksi esimerkiksi kehitystiimien info näytöillä. Käännösprosessiin voidaan sisällyttää lisäksi erilaiset koodi- sekä testikattavuusanalyysit tuottamaan dataa käännöksen laatua kuvaaviin mittareihin. Tämänkaltaista näkyvyyttä voidaankin pitää yhtenä CI-serverin suurimmista hyödyistä. [1.s.64-65]

Toimiakseen malli edellyttää sitoumusta yhteisiin periaatteisiin, sekä toimintamalleihin (liite 2). Tämä korostuu eteenkin laajoissa projekteissa, joita toteuttavat useat eri tiimit ilman paikkasidonnaisuutta.

4 Julkaisuputki

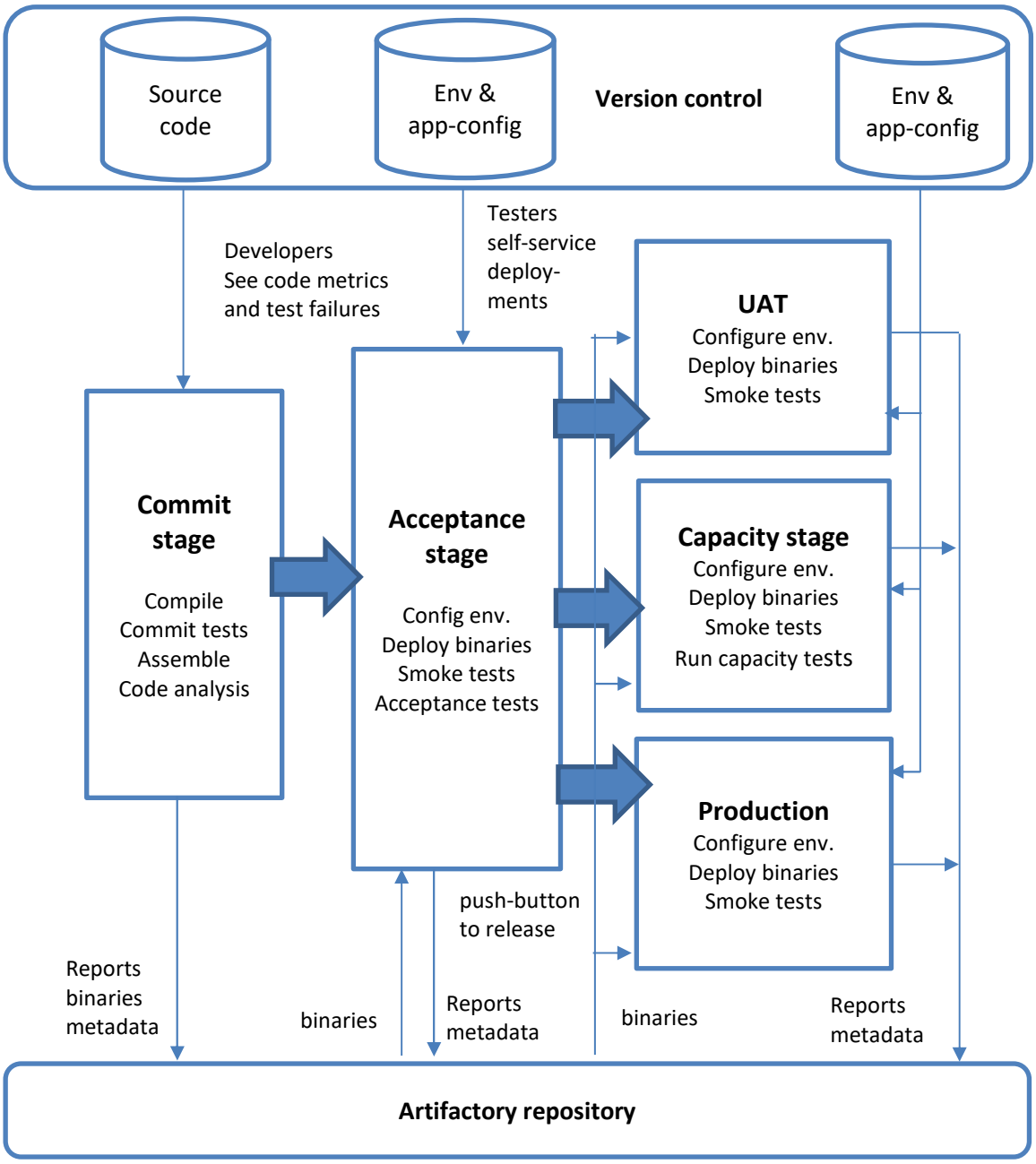
Jatkuva integrointi on valtava harppaus tuottavuuden sekä laadun parantamiseksi projekteissa, joissa se on otettu käyttöön. Jatkuvan integroinnin mukanaan tuoma nopea palaute auttaa varmistamaan, että ohjelmakoodi toimii odotusten mukaisesti ja että versionhallintaan lisätyt koodimuutokset kääntyvät oikein ja läpäisevät niille määritellyt testit hyväksytysti. Ohjelman julkaisun kannalta pelkkä *jatkuva integrointi* ei kuitenkaan riitä, koska se keskittyy pääasiassa kehitystiimeihin ja sen tuottama materiaali on ainoastaan syöte julkaisuprosessin seuraaville vaiheille, kuten manuaalitestaukselle. [1.s.105]

Valtaosa hukasta julkaistaessa ohjelmaa tuleekin prosessin myöhemmistä vaiheista. Valitettavan yleinen ja tunnistettavissa oleva tilanne on, että testaajat odottavat toimivaa käännöstä tai kehitystiimit saavat palautetta löytyneistä ohjelmistovioista huomattavan myöhäisessä vaiheessa. Kehitysprosessin päätteeksi voidaan myös huomata, ettei ohjelmisto täytä sille määriteltyjä ei-toiminnallisia vaatimuksia. Edellä mainitut seikat voivat todennäköisesti myös johtaa siihen, että ohjelmisto ei ole julkaisukelpoinen. [1.s.105]

Hukan vähentämiseksi jokainen prosessin osa täytyy automatisoida mahdollisimman pitkälle. Testaajilla tulee olla mahdollisuus toimittaa ohjelma helposti haluttuun ympäristöön ja kehittäjillä nähdä helposti käännöksen tila ja etenemisen julkaisuprosessin eri vaiheessa. Projektista vastaavilla henkilöillä tulee olla mahdollisuus seurata prosessin kokonaisuuden kannalta tärkeitä mittareita (Key Performance Indicator). Tuloksena jokaisella prosessiin osallistuvalla henkilöllä on näkyvyys ja pääsy toiminnan kannalta tärkeisiin asioihin. Lisäksi näkyvyys auttaa kehittämään julkaisuprosessin tuottamaa palautetta, jotta pullonkaulat voidaan tunnistaa, optimoida tai poistaa kokonaan. [1.s.106]

4.1 Perusmalli julkaisuputkesta

Lyhyesti sanottuna julkaisuputki on kuvan 8 mukainen automatisoitu prosessi, jonka puitteissa versionhallinnassa oleva materiaali muutetaan suoritettavaksi ohjelmaksi, joka toimitetaan ohjelmaa tarvitsevalle käyttäjälle. Julkaisuputken ruumiillistumana voidaan pitää CI/CD-työkalujen tuomaa mahdollisuutta hallita ja nähdä muutosten etenemistä versionhallinnasta aina loppukäyttäjälle saakka. [1.s.106-107]



Kuva 8. Julkaisuputken periaatekuva [1.s.111]

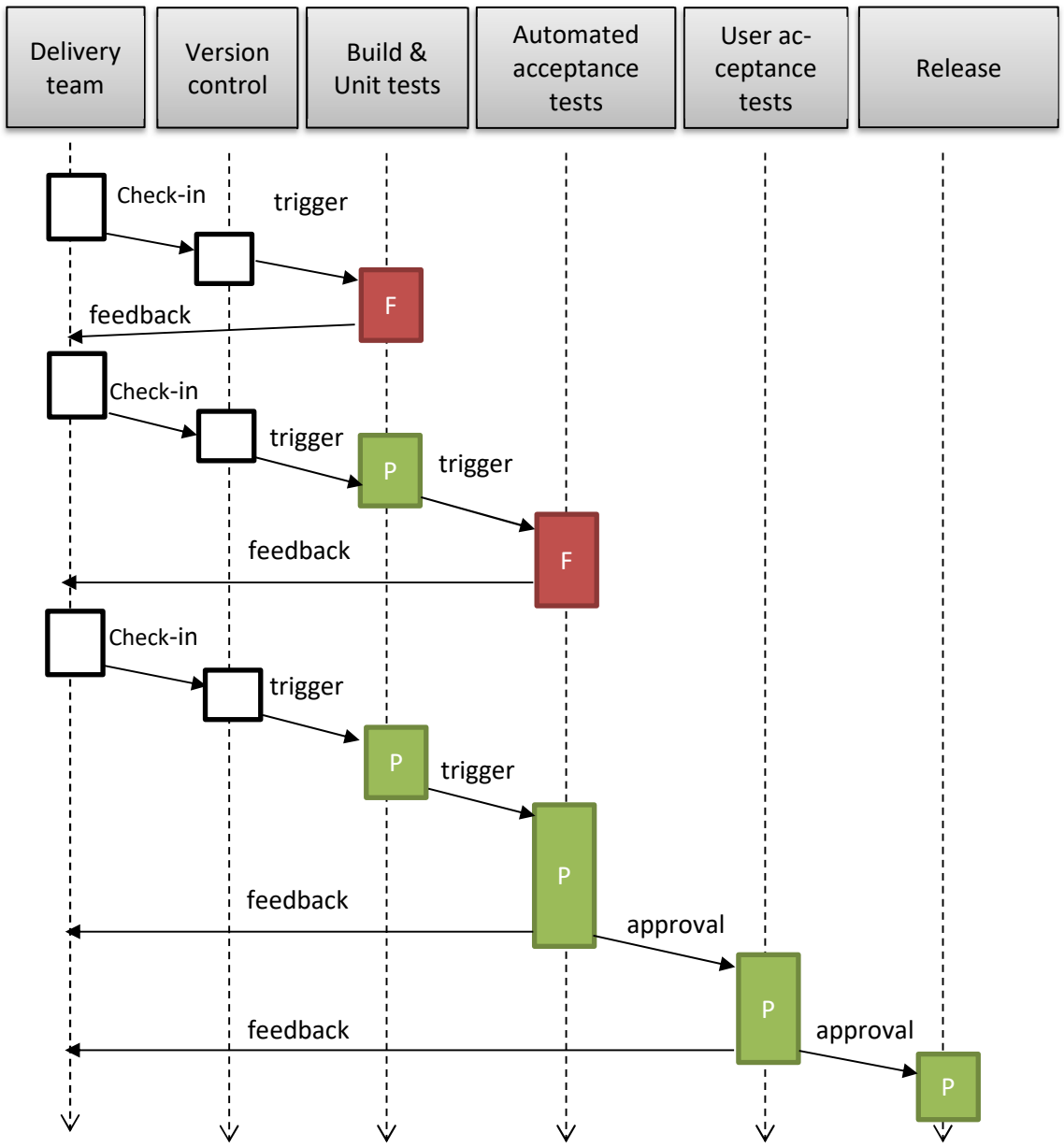
Prosessi käynnistyy versionhallintaan tehdystä muutoksesta, jolloin CI-ympäristö käynnistää julkaisuputkesta uuden instanssin. Ensimmäisessä vaiheessa käännetään koodit, ajetaan yksikkötestit, suoritetaan koodianalyysit. Mikäli ensimmäinen vaihe menee läpi, käännetyt komponentit versioidaan ja tallennetaan niille määriteltyyn versionhallintaan. Tässä vaiheessa voidaan tehdä myös muita toimenpiteitä, kuten luoda asennuspaketti tai tehdä testitietokantojen valmistelut myöhemmin tehtäville hyväksyntätesteille. [1.s.111]

Toisessa vaiheessa suoritetaan tyypillisesti pidempään kestäviä automatisoituja hyväksyntätestejä. Tässä vaiheessa pitkäkestoiset testisetit voidaan jakaa rinnakkaisiin ajoihin, jotta palaute kehittäjille saataisiin mahdollisimman nopeasti. Toinen vaihe kannattaa käynnistää ainoastaan, mikäli ensimmäinen vaihe menee hyväksytysti läpi. Toisen vaiheen jälkeen putki haarautuu siten, että se sallii itsenäiset toimitukset erinäisiin ympäristöihin. Kuvan 8 esimerkissä näitä ympäristöjä ovat UAT, suorituskykytestaus sekä tuotanto. Tässä tapauksessa käännös toimitetaan niihin manuaalisesti. [1.s.112]

Toimitukset eri ympäristöihin kannattaa tehdä käyttäen samaa käännösskriptiä ja poikkeavuudet ympäristöjen kesken hallita esimerkiksi konfiguraatiotiedostojen avulla. Toimitus on hyvä tehdä aina tuotantoympäristöä vastaavaan ympäristöön. Tällöin voidaan luottaa, että hyväksytysti läpi menneen testausvaiheen jälkeen ohjelma toimii myös tuotantoympäristössä. [1.s.117-118]

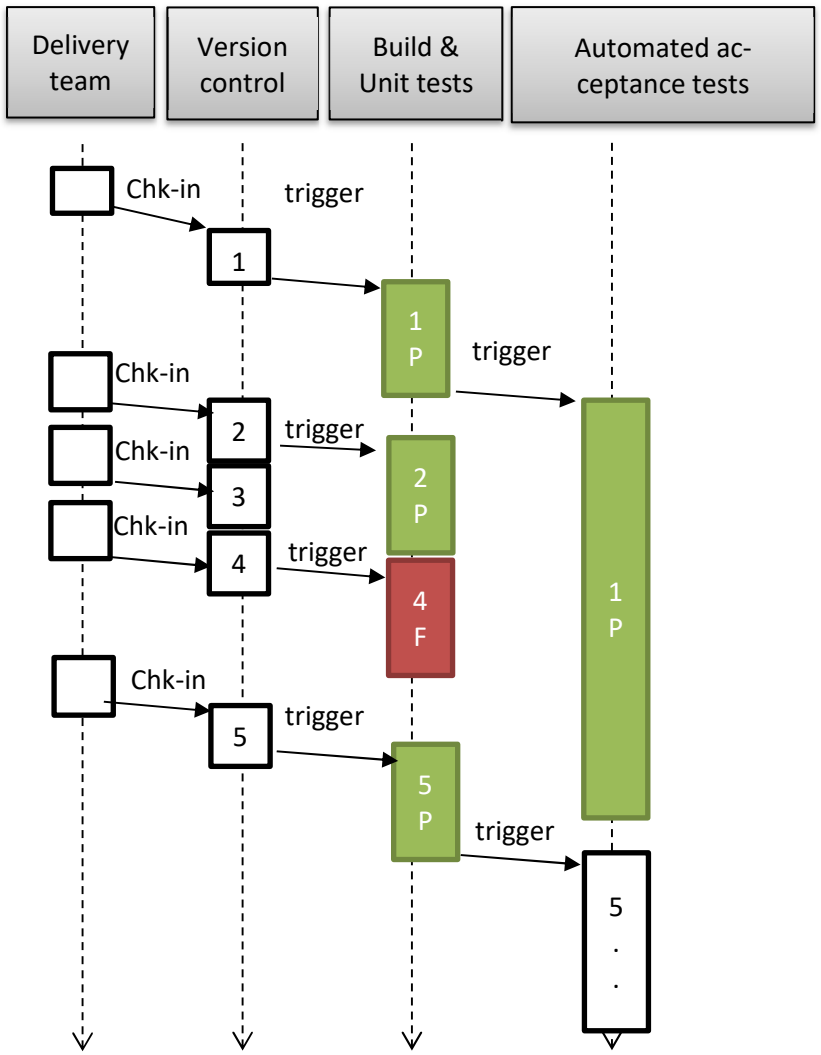
Kuvassa 9 esitetyn prosessimallin päätarkoitus on eliminoida epäkelpo julkaisukandidaatti mahdollisimman aikaisessa vaiheessa. Jokainen ohjelmaan tehtävä muutos käynnistää putken ja käy läpi mittavan prosessin ennen päätymistä asiakkaalle sisältäen koodin kääntämisen sekä sen jälkeiset useat eri testauksen ja toimituksen vaiheet. [1.s.106-107]

Prosessin toimivuus perustuu kuvassa 9 esitettyyn nopeaan palautteeseen prosessista, sekä epäonnistumiseen johtaneen juurisyyn nopeaan paikantamiseen. Lisäksi mallin soveltamisen mukana tulee myös muita hyötyjä. Esimerkiksi testaamattomia käännöksiä ei prosessin myötä pääse tuotantoon tai pelko regressioista on huomattavasti pienempi kiireellisten päivitysten kohdalla. Automatisoidut toimitus sekä julkaisu prosessit ovat nopeita, toistettavia sekä luotettavia, jolloin ei ole merkitystä, palaanko edelliseen vai siirrytäänkö seuraavaan versioon. [1.s. 108-109]



Kuva 9. Nopea palaute prosessista [1.s.109]

Kuvassa 10 on esitettynä periaatekuva siitä, kuinka jokainen muutos versionhallintaan käsitellään omana instanssina. Mikäli esimerkiksi edellisen muutoksen yksikkötestit ovat vielä käynnissä, odotellaan testien valmistumista ennen testausvaiheen käynnistämistä uudelleen. Malli toimii hyvin tilanteessa, jossa yksikkötestausvaihe ei kestä liian kauaa, jolloin samaan testisettiin ei oteta mukaan useita eri muutossettejä.



Kuva 10. Vaiheiden ajastus putkessa [1.s.119]

Kun edellä mainittu prosessi käydään läpi ja julkaisukandidaatti on valmiina odottamassa toimitusta tuotantoon, tiedämme seuraavat asiat: [1.s.132]

- Koodi on kääntynyt onnistuneesti
- Koodi tekee sen mitä suunnittelija on suunnitellut sen tekevän, koska yksikkötestit ovat menneet läpi
- Järjestelmä tekee sen, mitä tuotemäärittely ja asiakkaat ovat määritelleet, koska kaikki hyväksyntätestit ovat menneet läpi
- Konfigurointi on kunnossa, koska ohjelma on testattu mahdollisimman lähellä tuotantoympäristöä tai sen kopiossa

- Koodissa on mukana kaikki sen tarvitsemat komponentit, koska sen toimittaminen erinäisiin ympäristöihin on onnistunut
- Toimitus järjestelmä toimii koska minimissään sitä on käytetty kyseiselle julkaisukandidaatille ainakin kerran kehitysympäristössä, kerran hyväksyntätestausvaiheessa ja kerran testausympäristössä ennen kuin sen etenemistä tähän vaiheeseen.
- Versionhallinnassa on kaikki tarvittava tuotteen julkaisemiseen ilman manuaalista väliintuloa, koska toimitus on tässä vaiheessa tehty jo useaan kertaan

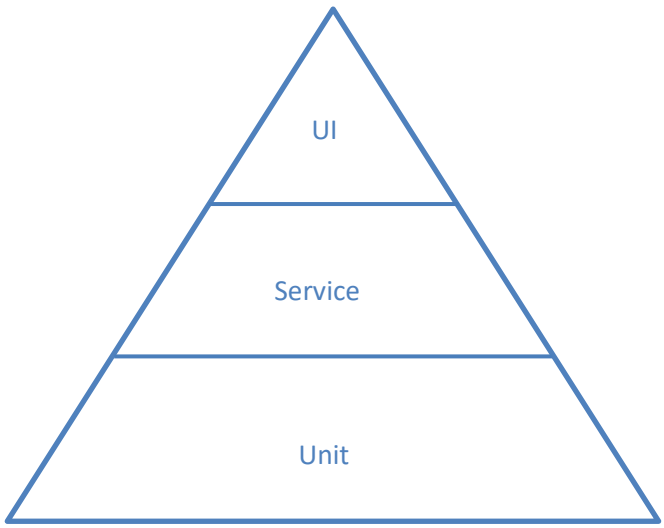
4.2 Julkaisuputken ensimmäinen vaihe (commit stage)

Käännösputken ensimmäisen vaiheen tarkoituksena on eliminoida mahdolliset tuotantoon sopimattomat käännökset mahdollisimman nopeasti. Lisäksi sen tehtävänä on tuottaa nopeaa palautetta kehitystiimeille ohjelman tilasta mahdollisimman nopeasti. Vaiheeseen on hyvä sisällyttää myös erinäiset koodianalyysit. Vaiheen avulla voidaan minimoida kooditason integrointiin käytettävä aika sekä parantaa tuotetun koodin laatua, koska se toimii samalla ajurina hyvien suunnittelumallien käyttämiselle. Liitteessä 3 on esitetty joitakin hyödyllisiä käytäntöjä vaiheeseen liittyen. [1.s. 169-170]

Käännettävien ohjelmointikielien osalta vaihe käynnistyy ohjelmakoodin kääntämisellä, minkä jälkeen läpi mennyt käännös testataan nopeaksi optimoidulla setillä vaiheeseen valittuja testejä. Näihin testeihin on hyvä sisällyttää yksikkötestien lisäksi pieni määrä muun tyypin testejä. Testauksen laajentaminen yksikkötestauksen ulkopuolelle tässä vaiheessa parantaa luottamusta ohjelman toimintaan. Vaikka yksikkötestit kattavatkin valtaosan vaiheen testauksesta, sen erillinen nimeäminen (commit test suite) on perusteltua vaiheen sisältäessä myös muun tyypin testejä. [1.s.120]

Tarve yksikkötestauksen laajentaminen muihin testityyppeihin perustuu opittujen ja tyypillisten kipupaikkojen, sekä myöhemmissä testausvaiheissa esiintyviä virhetyyppien nopeaan testaamiseen. Tämän tyyppinen jatkuva kehitystyö on tarpeen, jotta edellä mainituissa tilanteissa esiintyvät virheet saataisiin kiinni mahdollisimman ajoissa. Lisättävät testit voivat olla esimerkiksi integraatio tai hyväksyntätason testejä. Mikäli testimassan kasvaminen venyttää vaiheen kestoa yli 5 minuutin, kannattaa testien ajamista jakaa rinnakkaisiin prosesseihin. [1.s.135]

Suurin osa vaiheen testeistä tulee olla yksikkötestejä, joiden suoritus on nopeaa. Toinen tärkeä perusperiaate on, että yksikkötestauksen puitteissa katetaan suurin osa koodikannasta. 80% kattavuutta voidaan pitää hyvänä, koska tällöin saadaan perusluottamus koodin tekniseen toimintaan. Yksikkötestaus ei missään tilanteessa anna täyttä varmuutta, että ohjelma toimii, mutta sitä varten ovat putken seuraavat vaiheet. Kuvassa 11 esitetty testauspyramidi visualisoi havainnollisesti, kuinka koko ohjelman testiautomaatio kannattaa rakentaa sisältäen yksikkö, palvelu sekä UI-testejä. Pyramidissa yksikkötestit kattavat suurimman osan, koska ne ovat nopeita ajaa. Yksikkötestien lisäksi kuuluvat testiautomaatiokokonaisuuteen myös lukumäärältään vähäisemmät ja pidempikestoiset hyväksyntätestit. Tässä yhteydessä ne on jaettu palvelu sekä UI testeihin. Jotta ohjelma voidaan todeta toimivaksi, tulee sen testauksen sisältää kaikkia näitä tasoja. Testauksen neliöstä testauspyramidi kattaa vasemmanpuoleiset laatikot, eli pyramidin testit ovat ohjelmistokehitystä tukevia. Vaiheen testien suunnitteluun on olemassa muutamia liitteessä 4 esitettyjä periaatteita ja käytäntöjä.

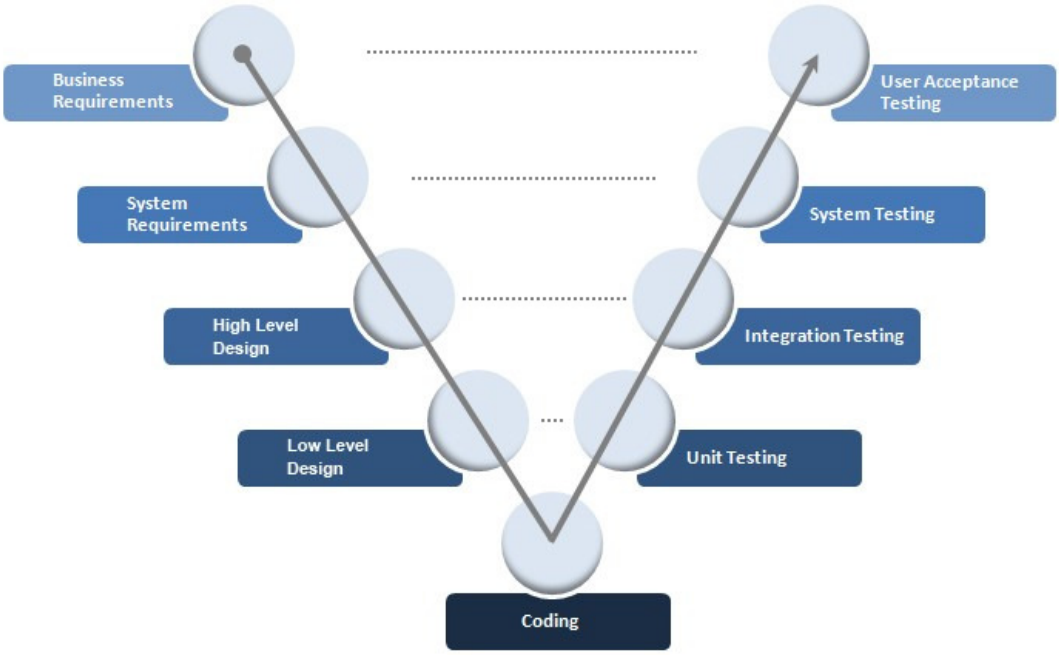


Kuva 11. Testauspyramidi [1. s.178]

4.3 Hyväksyntätestausvaihe

Kuvassa 12 on esitettynä testauksen V-malli, joka havainnollistaa eri testauksen tasojen suhdetta vaatimuksiin. Mallin pohjalle sijoittuvien tasojen kattava testaus ensimmäisessä vaiheessa on hyvä tapa saada kiinni monenlaisia virheitä, mutta on paljon asioita, joita sen tuoman testipeiton

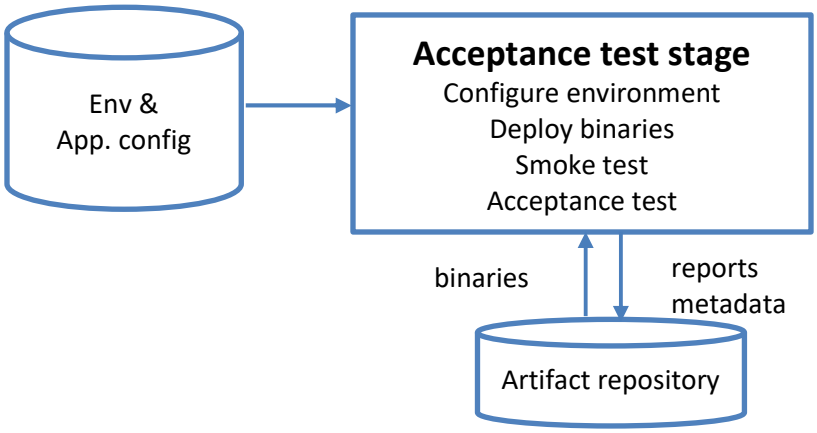
avulla ei voida vielä varmentaa. Luonnollisena jatkumona ensimmäisen vaiheen testaukselle tulee tuotantoympäristöä vastaavassa ympäristössä toteutettava automaattinen hyväksyntätestaus, joka on esitettynä V-mallin ylimmällä tasolla. Ilman tätä testauksen vaihetta ei voida olla varmoja, täyttääkö ohjelma sille asetetut hyväksymiskriteerit. Hyväksyntätestausvaiheessa suoritettavia toiminnallisia testejä voidaan verrata ensimmäisessä vaiheessa suoritettaviin yksikkötesteihin. Näin myös siitä syystä, että vaikka suurin osa hyväksyntätason testauksesta on toiminnallisia hyväksyntätestejä, niin ei kuitenkaan kaikki. [1.s.124]



Kuva 12. Testauksen V-malli [28]

Hyväksyntätestit kannattaa suorittaa jokaiselle ensimmäisen vaiheen läpäisseelle käännökselle, jotta kelpoisuus seuraaviin vaiheisiin voidaan varmistaa. Esimerkiksi suorituskykytestauksessa voi olla tietynlaista tulkinnanvaraisuutta, mutta hyväksyntätestaus menee joko läpi tai ei. Kehitysprosessin sujuvuuden kannalta vaiheeseen täytyy suhtautua sen vaatimalla vakavuudella. Vaikka automaattitestit aiheuttavatkin aluksi työtä ja kustannuksia, tulevat ne jatkossa vähentämään ylläpitokustannuksia merkittävästi. Kokemuksen pohjalta tiedetään, että ilman kattavaa hyväksyntätestausta, aikaa kuluu paljon bugien etsintään sekä korjaukseen. Näin käy yleensä juuri tilanteessa, jolloin ohjelmaa pidettiin jo julkaisukelpoisena [1.s.213]

Hyväksyntätestit ovat olennainen osa julkaisuputkea ja ne testaavat sovelluksen toiminnallisten ja ei-toiminnallisten vaatimusten hyväksyntäkriteereiden täyttymistä. Kuvassa 13 on esitetty hyväksyntätestausvaihe lohkokaaviotasolla. [1.s.187]



Kuva 13. Hyväksyntätestausvaihe [1.s.187]

Hyväksyntätestit on hyvä ajaa tuotantoympäristöä vastaavassa ympäristössä. Mikäli tuotantoympäristöä täysin vastaavaa ympäristöä ei ole mahdollista toistaa, täytyy se simuloida mahdollisimman hyvin. Hyväksyntätestien kehitys- ja ylläpitovastuu on hyvä olla tiimeillä, jolloin niihin saadaan tietynlainen *omistajuussuhde*. Omistajuussuhteen puuttuminen onkin eräs merkittävimmistä syistä sille, että hyväksyntätestien korjaamiseen ja ylläpitoon ei löydy riittävää mielenkiintoa. Pelkästään testaajien voimin kirjoitetut testit, ovat yleensä hyvin käyttöliittymä painotteisia, koska testaajilla ei välttämättä ole osaamista tehdä testejä konepellin alle tai luoda erillisiä API-rajapintoja, joiden avulla ohjelman testaus onnistuisi ohjelman oman rajapinnan avulla. [1.s.125]

4.3.1 Miksi on hyvä automatisoida

Joskus voi kuulla epäiltävän automatisoitujen hyväksyntätestien tuomaa hyötyä suhteutettuna niistä aiheutuviin kustannuksiin. On myös esitetty, että kattava yksikkö-, ja komponenttitestaus (tuettuna ”Extreme Programming” -käytännöillä) on riittävä tapa varmistaa ohjelman toiminta ilman hyväksyntätestauksesta aiheutuvia kuluja. Käytäntö on kuitenkin osoittanut, että automaattiset hyväksyntätestit voivat löytää ohjelmasta puutteita, joita yksikkö tai komponenttitasolla on mahdoton löytää. Lisäksi asiallisesti laadittu automaattinen hyväksyntätestaus ovat huomattavasti kustannustehokkaampi tapa julkaista ketterästi laadukkaita ohjelmia, verrattuna satunnaisesti suoritettuun manuaali-, ja regressiotestaukseen. Automaattisista hyväksyntätesteistä

luopuminen lisää myös testaajien taakkaa, koska tällöin heidän vastuulleen jää toistuva samojen asioiden läpikäynti. Huonosti toteutetut automaattiset hyväksyntätestit voivat olla hankalasti ylläpidettäviä ja syödä kohtuuttomasti tiimin resursseja. Lisäksi tulee muistaa, että kaikkea ei kannata, tai ei ole järkevää automatisoida. [1.s.188-189, 87]

Automaattisen hyväksyntätestauksen puolesta puhuu myös se, että minkään muun tyypin testit eivät verifioi tuotantoympäristön kaltaisessa ympäristössä ohjelman kykyä toimia käyttäjän odottamalla tavalla. Muut testit eivät myöskään testaa erilaisia käyttäjäskenaarioita ja löydä ohjelmistovikoja, jotka esiintyvät käyttäjän toteuttaessa tiettyjä ohjelman suorituspolkuja. Ne ovat hyviä löytämään esimerkiksi säikeistykseen tai tapahtumaohjattuun (event programming) ohjelmointiin liittyviä ongelmia. Lisäksi niiden avulla voidaan saada kiinni erityyppisiä, ohjelmistoarkkitehtuurista tai ympäristö- ja konfiguraatio-ongelmista johtuvia ohjelmavikoja. [1.s.189]

Kattava hyväksyntätestien automatisointi madaltaa myös kehittäjien kynnystä koodin uudelleenjärjestelyyn sen tuoman suojan vuoksi. Tällöin usein yksikkö ja komponenttitason testausta joudutaan muuttamaan vastaamaan uutta kooditason toteutusta, mutta automaattiset hyväksyntätestit pysyvät ennallaan ja varmistavat että ohjelma toimii edelleen vaatimusten määrittämällä tavalla. On myös yleisesti tunnettu tosiasia, että kehittäjien itse omalle työlleen kirjoittamat testit eivät yleensä löydä ohjelmasta bugeja samalla tehokkuudella verrattuna testaajien tekemiin testeihin. Tästä syystä testaajien osallistuminen automaattisten hyväksyntätestien tekemiseen on olennaisen tärkeää. [1.s.189-190]

Yksittäisen hyväksyntätestin tarkoituksena on verifioida, että yksi tai useampi vaatimukselle määritelty hyväksyntäkriteeri täyttyy. Yksittäinen vaatimus voi olla joko toiminnallinen, tai ei-toiminnallinen. Hyväksyntävaiheen testien tuleekin verifioida, että ohjelma täyttää asiakkaan sille asettamat vaatimukset, sekä löytää mahdolliset regressiot ja muut ohjelmistoviat. Hyväksyntätestien tarkoituksena on todistaa, että ohjelma tekee sen, mitä asiakas on pyytänyt. Yksikkötestien tarkoituksena on todistaa, että koodi tekee sen mitä koodari on koodannut. [1.s.188]

Hyväksyntätestausvaihe toimii myös osana regressiotestausta, jonka tarkoituksena on verifioida, etteivät tehdyt muutokset ole rikkoneet olemassa olevaa toiminnallisuutta. Automaattiset hyväksyntätason testit eivät ole pelkästään erillisten tiimien vastuulla, vaan se on osa kehitysprosessia ja niitä suunnitellaan, tehdään ja ylläpidetään yhteisvastuullisesti kehittäjien, testaajien sekä asiakkaiden kanssa. [1.s.124]

4.3.2 Hyväksyntätestien automatisointi

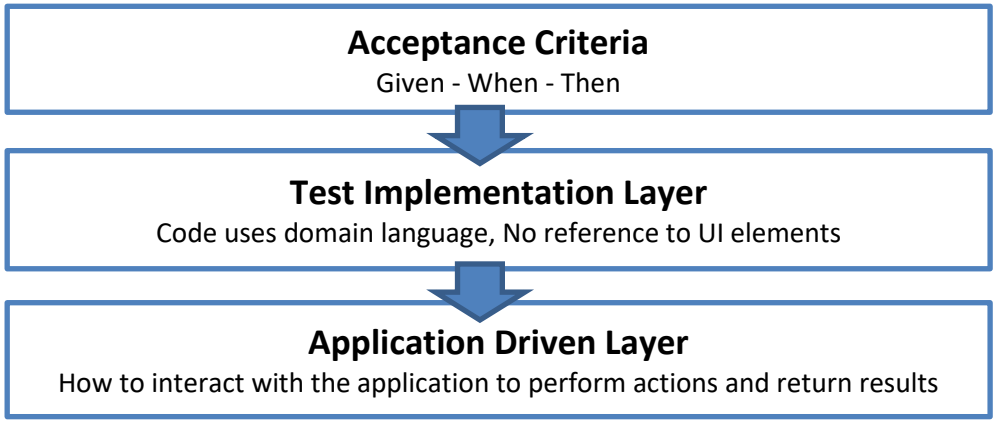
Manuaalitestaukseen käytettävää aikaa on hyvä seurata, jotta toimintojen automatisoinnin kannattavuudelle saadaan selkeä mittari. Toiminnon testaus kannattaa automatisoida, mikäli sen manuaalinen testaus on toistuvaa, ja toimintoa testaavan automatisoidun testin ylläpito ei vaadi liikaa työtä. [1.s.88]

Vaikka testiautomaation kannalta tärkeintä on testien hyvä laatu, voidaan automaation tasoa mitata myös seuraamalla testiautomaation kattamaa ohjelmakoodin määrää. Automaattisen hyväksyntätestauksen tasoa voidaan kokeilla esimerkiksi kommentoimalla tietty osa ohjelmakoodista. Mikäli testit menevät edelleen läpi, voidaan tästä tehdä omat johtopäätökset. [1.s.87]

Usein voi olla hyvä jakaa testit tiettyihin kategorioihin, jotta esimerkiksi hyväksymistestien ajo voidaan kohdentaa täsmällisesti riippuen siitä, mihin kohtaan koodia on koskettu. Tietyissä tilanteissa on ajan hukkaa suorittaa kattava testaus, mikäli ohjelmaa on muutettu pelkästään tietyiltä osin. [1.s.61]

Natiivilla kooditoteutuksella tehdyt hyväksyntätestit on hyvä tehdä noudattamaan kuvassa 14 esitettyä, kolmen kerroksen periaatetta. Ylin kerros kuvaa hyväksymiskriteeriä, mistä kaikki lähtee liikkeelle. Toisessa kerroksessa kuvatut testien toteutukset tulee kirjoittaa siten, ettei hyväksymiskriteerin täyttymiseen vaikuttamaton muutos riko testiä. Testien toteutuskerros käyttää alimmaiseksi sijoitettua ohjelman ajurikerrosta, jonka varsinainen tehtävä on olla vuorovaikutuksessa testattavan koodin kanssa. [1.s.191]

Liitteissä 5, 6 ja 7 on esitetty hyväksyntätestaukseen liittyviä teknisiä menetelmiä, sekä testien kirjoittamiseen ja ylläpitoon liittyviä hyviä käytäntöjä.



Kuva 14. Hyväksyntätestin kerrokset [1.s.191]

4.3.3 Hyväksyntätestien suorituskyky

Hyväksyntätestien suorituskyvyn parantamiseen ei useinkaan välttämättä käytetä paljoa työtä, koska ne ovat muutenkin hitaita ja niiden tärkein tehtävä on osoittaa, että ohjelma täyttää sille annetut kriteerit. Suorituskyvystä voidaan myös ajatella, että se on suoraan verrannollinen testisetin laatuun ja pitkät testiajot kertovat heikkolaatuisista testiseteistä. Hyväksyntätestisetit on hyvä pitää hyvin suunniteltuina ja yhdenmukaisina, mutta nopeuden sijaan on parempi painottaa parempaa kattavuutta. [1.s.218]

Hyväksyntätason testaus ottaa aikaa, eikä se tosiasia muutu mihinkään. Yksinkertaisenkin systeemin hyväksyntätestaus vaatii paljon valmistelevia toimenpiteitä. Ennen yhdenkään testin suorittamista tulee esimerkiksi konfiguroida järjestelmä, saattaa testattava ohjelma (SUT) toimintakuntoon ja lopuksi käynnistää se. Siitä huolimatta, että ohjelmistoviat pyritään paikantamaan mahdollisimman nopeasti palautteen saamiseksi kehitystiimeille, on aivan tyypillistä ja hyväksyttävää, että hyväksyntätestisetin ajo kestää useita tunteja. On kuitenkin olemassa tiettyjä mahdollisuuksia tämän ajan lyhentämiseksi. [1.s.218]

Helppoja menetelmiä hyväksyntätestaukseen käytetyn ajan lyhentämiseksi on esimerkiksi listata hitaimmat testit, ja säännöllisin väliajoin käyttää aikaa niiden tehokkuuden parantamiseksi. Koska hyväksyntätestit ovat tilallisia ja useat testit käyttävät tiettyjä valmistelu rutiineja, kannattaa näiden rutiinien tehokkuuteen myös kiinnittää huomiota. Mikäli ohjelmassa on julkinen rajapinta tiettyjen asioiden hoitamiseksi ilman, että niitä suoritetaan käyttöliittymän kautta, kannattaa niitä käyttää. [1.s.219]

Erittäin tehokas tapa testiajojen nopeuttamiseksi on ajaa niitä rinnakkain, jolloin esimerkiksi erinäiset testisetit suoritetaan erillisissä prosesseissa. Tällöin on kuitenkin huomioitava, että prosessit eivät ole mitenkään vuorovaikutuksessa keskenään. Tämä voikin olla erittäin kustannustehokas tapa, mikäli tuotantoympäristön kaltaiset ympäristöt eivät vaadi kalliita investointeja. [1.s.220]

4.4 Myöhäisemmät testausvaiheet

Julkaisukandidaatille hyväksyntätestaus on merkittävä virstanpylväs, koska tämän vaiheen jälkeen se siirtyy pois kehitystiimien käsistä laajempaan levitykseen. Joissain tapauksissa julkaisu

voidaan toimittaa tuotantoon jo hyväksyntätestausvaiheen jälkeen automaattisesti, mutta yleensä se käy vielä läpi sarjan erinäisiä manuaalisia tai puoliautomaattisia testejä.

4.4.1 Manuaalitestaus

Manuaalitestausvaihe varmistaa, että järjestelmää voidaan käyttää vaatimustenmukaisesti. Manuaalitestauksella voidaan löytää ohjelmistovikoja, joita testiautomaatiolla on hankala löytää. Iteratiivisessa prosessissa hyväksyntätestausta seuraa aina jonkin muotoista manuaalitestausta, kuten esimerkiksi tutkivaa testausta, käytettävyystestausta tai erinäisiä näyttötapauksia (showcases). Testaajan rooli tässä tapauksessa ei ole tehdä regressiotestiä, vaan ensisijaisesti validoida hyväksyntätestien toiminta käymällä toiminta kertaalleen läpi manuaalisesti. Seuraavassa vaiheessa voidaan keskittyä testaukseen, jossa ihminen on hyvä, mutta testiautomaatio ei. Ihmisen vahvuusalueita ovat tutkiva testaus, käytettävyystestaus, sekä ohjelman käyttökokemuksen analysointi (miltä näyttää ja tuntuu). Lisäksi voidaan käydä läpi myös perinteiset ongelmapaikat (worst case scenario). Testiautomaation tarkoituksena on vapauttaa testaajat tekemään tämän vaiheen luovaa testaustyötä. [1.s. 110, 128]

4.4.2 Ei-toiminnallinen testaus

Jokaisessa järjestelmässä on paljon ei-toiminnallisia vaatimuksia. Näitä voivat vastata esimerkiksi kapasiteettitestit, suorituskykytestit tai turvallisuustestit. Usein tämänkaltaiset asiat on hyvä testata / mitata testiautomaation avulla. Tähän katekoriaan liittyviä testejä ei perinteisesti ajeta jokaiselle käännökselle, johtuen niiden kestosta tai muista seikoista. Tarvittaessa testien automaattinen käynnistys putkesta on kuitenkin perusteltua ja se tuo lisäarvoa testaukseen. Esimerkiksi tilanteissa, joissa suorituskyky on kriittinen tekijä ohjelman käyttämisen kannalta, kannattaa ainakin osa kriittisistä testeistä ottaa mukaan hyväksyntätason testeihin. [1.s.128]

Ei-toiminnallisten vaatimusten testaus on hyvä ottaa mukaan heti projektin alkumetreillä ainakin jollakin tasolla, jotta ohjelman ei-toiminnallisten vaatimusten tilasta voidaan saada asianmukainen käsitys. Automaation kannalta katsottuna tämä vaihe voi toimia vastaavasti hyväksyntätestausvaiheen kanssa. Tästä syystä ei-toiminnallisille testeille onkin hyvä luoda oma testisetti sisältäen joitakin tämän kategorian testejä. [1.s.136]

4.5 Integraatiot ulkoisiin järjestelmiin

Integraatiotestit ulkoisten järjestelmien kanssa voidaan myös ottaa osaksi kehitysputkea. Mikäli komponenttien integraatio analysoidaan kehitysprosessin aikaiseksi riskiksi, on automatisoitujen integraatiotestien kehitys hyvä ottaa mukaan jo projektin alkuvaiheissa. Integraatiot ulkoisiin järjestelmiin on hyvä huomioida mahdollisimman aikaisin, koska jokainen ulkoiseen järjestelmään integroituminen on riski projektille. [1.s.98]

Mikäli ohjelmisto on vuorovaikutuksessa useiden eri systeemien kanssa, tulee integraatiotestauksesta erittäin tärkeä osa ohjelman testiautomaatiota. Käsitteenä integraatiotestaus on ylikuormitettu, mutta käytännössä sillä tarkoitetaan kahden itsenäisen komponentin väliseen vuorovaikutukseen liittyviä testejä. Integraatiotestejä voidaan ajaa esimerkiksi kytkemällä testattava järjestelmä (SUT) simuloituun ulkoiseen järjestelmään. Vaihtoehtoinen malli on testata järjestelmän rajapintaa erillisessä testipenkissä, jonka koodi on osa järjestelmän testikoodia. Integraatiotesteissä ei ole hyvä kytkeytyä oikean elämän käytössä oleviin ulkoisiin järjestelmiin, kuten esimerkiksi käytössä oleviin tietokantoihin. Mikäli näin kuitenkin on, ulkoiselle järjestelmälle on jotenkin kerrottava, että siihen on kytkeydytty testaustarkoituksessa. [1.s.96]

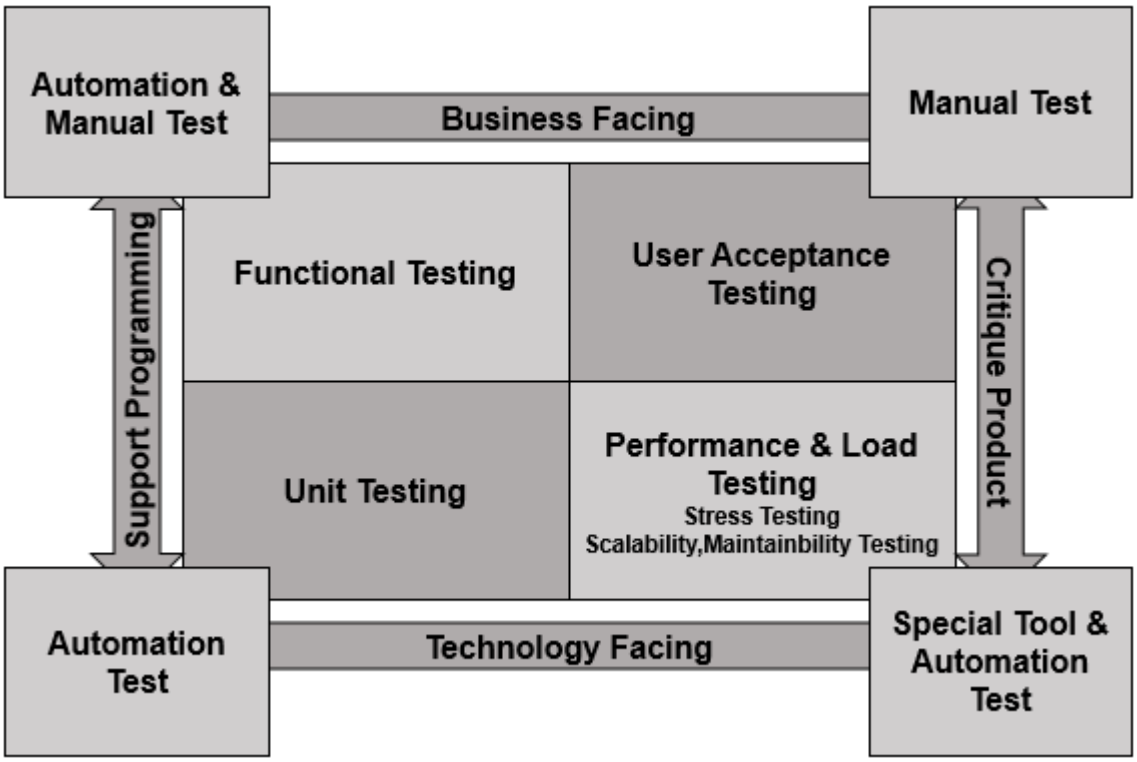
Ideaalitilanteessa integraatiotestausta voidaan tehdä tuotantojärjestelmän kanssa identtisesti käyttäytyvässä systeemissä. Usein tähän tilanteeseen ei kuitenkaan päästä ja tällöin integraatiotestejä varten voidaan joutua kehittämään erillinen testipeti (test harness). Testipeti voi olla hyvinkin tarkka ja yksityiskohtainen, todellisuutta yksityiskohtaisesti simuloiva. On tärkeää, että testipeti generoi vasteet tuotantoympäristöä vastaavalla tavalla odotettujen tilanteiden lisäksi myös odottamattomiin tilanteisiin. [1.s.98]

5 Testauksen neliödiagrammi ketterässä ohjelmistokehityksessä

Ohjelmistolle tehtäviä testejä voidaan jakaa useisiin eri tyyppeihin, mutta eräs malli on jakaa ne kuvassa 15 esitettyyn neliödiagrammiin (testing quadrant). Diagrammissa testit jaetaan osa-alueittain seuraavasti:

- Testaavatko ne ohjelmiston alempia kirjastokerroksia, vai vaatimusten määrittelemää toimintalogiikkaa
- Tukevatko ne kehitysprosessia, vai käytetäänkö niitä arvioimaan ohjelman vaatimuksenmukaisuutta

Jotta toimitettavan ohjelmiston korkea laatu voidaan varmistaa, tulee sen testaamiseen sisällyttää testejä kaikilta testauksen neliödiagrammin osa-alueilta. [1.s.84]



Kuva 15. Testauksen neliödiagrammi [14]

Vasempaan alakulmaan sijoittuvat kehitysprosessia tukevat kirjastotason testit, jotka toimivat ”turvaverkkona” ohjelmakoodille. Ohjelman toiminta rakentuu tämän tason päälle ja testit ovat

yleensä suunnittelijoiden itsensä kehittämiä sekä ylläpitämiä. Testit jakautuvat kolmeen pääkategoriaan, joita ovat yksikkötestit, komponenttitestit sekä ohjelman toimitusta varten tarvittavat, suorituskelpoisuutta testaavat testit. Yksikkötestien tulee kattaa valtaosa ohjelma koodipoluista, minkä vuoksi ne ovat regressiotestisetin selkäranka. Yksikkötestausta täydentämään tarvitaan komponenttitestausta, jolla voidaan varmistaa ohjelman eri komponenttien välinen toimivuus niiden elinkaaren eri vaiheissa. Tällä tasolla testaus on yleensä yksikkötestausta hitaampaa, koska testeihin liittyy usein tietokantojen, tiedostojärjestelmien sekä muiden rajapintojen alustuksia. [1.s.89] [14]

Vasempaan yläkulmaan sijoittuvat kehitysprosessia tukevat, toiminnallisia vaatimuksia testaavat automatisoidut testit tunnetaan usein nimellä toiminnalliset hyväksyntätestit. Hyväksyntätestauksen piiriin voi kuulua testejä useista eri kategorioista kuten esimerkiksi toiminnallisesta, kuormituskyky, käytettävyys ja turvallisuustestauksesta. Raja toiminnallisten ja ei-toiminnallisten hyväksyntätestien (oikea alakulma) välillä voi joskus olla häilyvä. Hyväksyntätestit ovat tärkeä osa ketteriä kehitysmenetelmiä, koska kehittäjälle ne vastaavat kysymykseen ”Milloin tiedän ominaisuuden olevan valmis?” ja käyttäjälle kysymykseen ”Sainko mitä tilasin?”. [1.s.85] [14]

Oikeaan yläkulmaan sijoittuvat hyväksymiskriteerien vaatimuksenmukaisuutta testaavat testit. Näiden manuaalisten testien on tarkoitus varmistaa, ettei ohjelma toimi pelkästään määritysten mukaisesti, vaan että määritykset ovat olleet asiakkaan vaatimusten mukaiset. Ohjelman toiminnot on hyvä esittää tällä tasolla asiakkaalle mahdollisimman usein, jotta mahdolliset väärinymmärrykset tai puutteelliset määrittelyt saataisiin mahdollisimman pian kiinni. Tällä tasolla tehtävät esittelyt poikivat usein kehitysideoita sekä keskustelua siitä, mihin suuntaa projektia halutaan jatkossa viedä. Asiakkaalta saatu palaute mahdollisimman aikaisessa vaiheessa on tärkeää, jolloin asioiden muuttaminen on vielä suhteellisen helppoa. Samaan kategoriaan kuuluu myös tutkiva testaus, jonka tarkoitus ei pelkästään ole löytää ohjelmistovikoja, vaan sen sijaan inspiroida luomaan uusia testiautomaatiosettejä. Tutkiva manuaalinen testaus voi osaltaan myös johtaa luomaan uusia vaatimuksia ohjelmistolle. [1.s.89-90]

Käytettävyystestaus sijoittuu myös tähän lohkoon ja sen avulla on tarkoitus saada palautetta ohjelman käytettävyydestä, eli kuinka helposti käyttäjä voi ohjelman avulla toteuttaa tarvitsemansa toimenpiteet. Käytettävyystestauksen toteutukseen voi olla useita eri menetelmiä. Eräs muoto on A/B-testaus ja Lahtisen (2015) mukaan sillä tarkoitetaan mallia, jonka puitteissa tietylle asiakasryhmälle annetaan käyttöön kaksi erilaista versiota testattavasta tuotteesta ja testin tulokset saadaan vertailemalla asiakkaiden tuotteista antamaa palautetta. Käytettävyystestausta voidaan

tehdä myös pyytämällä eri käyttäjiä toteuttamaan tietyt toimenpiteet ohjelman avulla samalla kuin testisessiot taltioidaan erinäisin menetelmin myöhempää analysointia varten. [1.s.90]

Viimeisessä vaiheessa ohjelma annetaan testiin sen todellisille loppukäyttäjille. Menetelmiä voi olla useita ja eräs malli on julkaista tietyille asiakkaille uusia ominaisuuksia jatkuvalla syötöllä ilman erinäistä ilmoitusta. Eräs vaihtoehto on julkaista useita eri aliversioita eri asiakkaille (Canary releasing). Aliversioissa julkaistuista uusista ominaisuuksista sekä niiden käyttöasteesta kerätään statistiikkaa, joista tehtyjen laskelmien perusteella voidaan sitten päättää mitä kannattaa kehittää eteenpäin. [1.s.90]

Oikeaan alakulmaan sijoittuvat vaatimuksenmukaisuutta testaavat kirjastotason testit. Näillä ei-toiminnallisilla testeillä tarkoitetaan järjestelmän laadullisien ominaisuuksien testausta ja niiden erottelu toiminnallisista testeistä on jossain mielin kaksijakoista, kuten myös etteivät ne testaisi ohjelman toimintalogiikkakerrosta. Usein ei-toiminnallisia vaatimuksia ei kuitenkaan kohdella tasavertaisesti verrattuna muihin vaatimuksiin, tai pahimmassa tapauksessa niitä ei validoida ollenkaan. Vaikkakaan puutteet näiden ei-toiminnallisten vaatimusten toiminnassa tulevat harvemmin esille, todennäköisesti ne kuitenkin vaikuttavat epäsuorasti toiminnallisten vaatimusten käyttäytymiseen ääritiloissa, kuten esimerkiksi raskaan kuormituksen alaisena. Usein termin ”non-functional requirement” käyttöä onkin kritisoitu ja sen korvaajaksi on ehdotettu termiä ”cross-functional requirement”. Ei-toiminnalliset vaatimukset on hyvä määritellä tasavertaisina vaatimuksina toiminnallisten kanssa. [1.s.91]

Ei-toiminnallisten vaatimusten testaaminen vaatii yleensä kattavampia laitteistoja verrattuna toiminnallisten vaatimusten testaukseen, olivat testit sitten automatisoituja tai eivät. Lisäksi niiden suoritusajat ovat yleensä huomattavasti pidempiä. Tämänkaltaisia testejä ajetaan yleensä harvakseltaan ja niiden implementointi osaksi testiautomaatiota ei yleensä tapahdu ensimmäisessä vaiheessa. Ei-toiminnallisen testauksen suunnittelulle onkin hyvä varata oma aikansa eteenkin vaativimmissa projekteissa. [1.s.91]

6 Ei-toiminnalliset vaatimukset

Ei-toiminnalliset vaatimukset ovat tärkeässä roolissa ohjelmistokehityksessä, koska pahimmillaan ne voivat estää ohjelman julkaisun ja siten tuottavat merkittävän riskin. Vaikka ei-toiminnalliset vaatimukset voitaisiinkin selvästi tunnistaa ja määritellä, voi olla hankala määritellä miten ja millä työmäärällä niiden toteutuminen voidaan varmistaa. Useat järjestelmät ovat kaatuneet tai olleet muuten käyttökelvottomia, koska ne eivät ole pystyneet toimimaan kuormituksen alaisena, tietoturva ei ole ollut kunnossa, ovat olleet liian hitaita tai viimeistään hylätty todettaessa, että edellä mainittuja asioita ei voi saattaa toiminnalliselle tasolle koodiarkkitehtuurin vuoksi. [1.s.225]

Jako toiminnallisten ja ei-toiminnallisten vaatimusten välillä on keinotekoinen, koska järjestelmän toiminnan kannalta ei-toiminnalliset vaatimukset ovat vähintäänkin yhtä tärkeitä toiminnallisten kanssa. On tärkeää, että projektin alkuvaiheessa nämä ei-toiminnalliset vaatimukset tunnistetaan heti ja niiden mittaamiseen ja testaamiseen löydetään kunnolliset menetelmät. Lisäksi on hyvä määritellä toimintatavat, joiden avulla näiden ei-toiminnallisten vaatimusten täyttymisestä ollaan tietoisia läpi kehitysprosessin, kuten esimerkiksi niiden ottaminen mukaan julkaisuputkeen jossakin muodossa. [1.s.226]

6.1 Ei-toiminnallisten vaatimusten hallinnointi ja seuranta

Eräs ei-toiminnallisiin vaatimuksiin liittyvä ongelma on yleisesti se, että ne on helppo pudottaa tai unohtaa pois projektisuunnittelusta tai niiden analysointiin ei kiinnitetä riittävästi huomiota. Tämä voi olla katastrofaalista, koska juuri nämä voivat olla merkittävä riski projektin onnistumiselle. Mikäli julkaisun kynnyksellä huomataan, ettei ohjelma ole käyttökelpoinen esimerkiksi suorituskykyongelmien vuoksi, voi se viivästyttää projektin valmistumista merkittävästi tai pahimmillaan päättää koko projektin. Ei-toiminnallisten vaatimusten implementaatio integroituu usein syvälle ohjelman arkkitehtuuriin ja tästä syystä ne voivat olla erittäin hankalia ja kalliita korjattavia projektin loppuvaiheessa, kun kaikki toiminnallisuus on rakentunut toimimattoman arkkitehtuurin päälle. [1.s.226]

Eräs tapa ei-toiminnallisten vaatimusten seurantaan on ottaa ne mukaan hyväksymiskriteereihin niiltä osin, kuin on järkevää ja vaatimuksen täyttymisen todentaminen ei vaadi liikaa työtä. Edellä

mainitun lisäksi voi projektin alkuvaiheissa olla hyvä määritellä kriittinen setti seurattavia ei-toiminnallisia vaatimuksia ja yhdistelemällä näitä kahta tapaa voidaan paremmin seurata ei-toiminnallisten vaatimusten täyttymisestä. Eräs lähestymistapa ei-toiminnallisten vaatimusten hallinnointiin voi olla myös ei-toiminnallisten vaatimusten kääntäminen vaatimuksiksi käyttäen hyväksi käyttäjän näkökulmaa. Käyttäjän vaatimus voisi olla esimerkiksi vasteaika painikkeen painamiselle. [1.s.227]

6.2 Ohjelman suorituskyvyn mittaaminen

Ohjelman kapasiteetin mittaus liittyy ohjelman eri ominaisuuksien tutkimiseen ja mittaamiseen. Perinteisiä kapasiteettitestauksen alaisia asioita ovat esimerkiksi:

- Skaalautuvuuden testaus, jonka puitteissa tutkitaan kuinka esimerkiksi ohjelman viestinkäsittelylogiikan toteuttaminen useammassa säikeessä voi parantaa ohjelman kykyä käsitellä sille lähetettyjä viestejä ja sitä kautta muuttaa vasteaikoja
- Kestotestaus, jota voidaan tehdä esimerkiksi kuormittamalla systeemiä tasaisesti ja mittaamalla esimerkiksi muistinkäyttöä ja vasteaikoja ja niiden muutoksia pitkällä aikavälillä
- Käsittelykykytestaus, jonka avulla voidaan mitata esimerkiksi ohjelman kykyä käsitellä sille annettuja tehtäviä verrattuna aikaan
- Kuormitustestaus, jonka avulla voidaan testata järjestelmän toimintakykyä suuren kuormituksen alaisena

Kaikki edellä mainitut testaustyypit ovat kokonaisuuden kannalta tärkeitä ja paras anti niiden suhteen on suorittaa niitä siten, että testit vastaavat mahdollisimman lähelle todellisia skenaarioita ohjelman käytöstä. [1.s.231]

Usein kapasiteettitestaus on enemmän mittaamista, kuin testaamista ja usein päätös testin läpimenosta tehdäänkin analysoimalla tilannetta kokonaisvaltaisesti kerätyn mittausdatan perusteella. Pelkän testaamisen sijaan voikin olla erittäin hyödyllistä kerätä kapasiteettitestien tuottamaa mittausdataa pitkällä aikavälillä ja tarkastella kokonaisuutta datan pohjalta luoduista graafeista, jotka sitten ovat helposti kehitystiimien nähtävillä. Joissain tilanteissa voi olla vaikea määritellä raja, jonka perusteella testin läpimenosta päätetään. Mikäli esimerkiksi käsittelykykytestin rajaksi asetetaan 100 käsittelyä sekunnissa ja todellisuudessa ohjelma käsittelee 200, voi ohjelman käsittelykyky pudota puoleen, ennen kuin asiaan kiinnitetään huomiota. Eräs malli tämän kaltaisten tilanteiden ehkäisemiseksi on määritellä suorituskyvylle ehdoton alaraja ja tämän

lisäksi jokaiselle testille muuteltavissa oleva, sen hetkistä suorituskykyä vastaava raja. Tämä auttaa nopean palautteen saantiin huomattavasti nopeammin, mikäli ohjelman suorituskyky laskee johtuen tietystä muutoksesta. Tällöin voidaan tapauskohtaisesti päättää, lasketaanko hetkellisestä suorituskyvystä kertovaa rajaa, vai optimoidaanko ohjelmaa. [1.s.232-233]

6.3 Kapasiteettitestauksen automatisointi

Kapasiteettitestauksen lisääminen julkaisuputkeen on eräs vakavasti harkittava seikka, koska sen satunnainen muistinvaraisesti tehtävä testaaminen ei ole systemaattista ja kapasiteettitestauksen jättäminen projektin loppupuolelle on testauksen tuoman hyödyn kannalta katsottuna myöhässä ja mahdolliset viat hankalia tai kannattamattomia korjata. Mikäli kapasiteettitestit otetaan mukaan julkaisuputkeen, tulee ne ajaa jokaiselle ensimmäisen vaiheen hyväksytysti läpäisevälle käännökselle. Kapasiteettitestien ajaminen julkaisuputkessa voi kuitenkin olla huomattavasti haastavampaa verrattuna hyväksyntätesteihin, koska ne voivat edellyttää testausympäristöltä esimerkiksi tiettyä mittalaitearsenaalia tai riippuvuuksien takia ne voivat olla helposti särkyviä. Alla on listattuna hyvälle kapasiteettitestille tunnusomaisia piirteitä. [1.s.238]

- Testaa todellisen elämän skenaarioita, todellisessa elämässä esiintyvien asioiden verifioimiseksi.
- Sillä on etukäteen määritellä kynnys, jonka perusteella tulos voidaan arvioida
- Kesto pysyy järkevissä puitteissa
- Se on sietokykyinen ohjelmassa tapahtuvia muutoksia vastaan, jotta sen jatkuva päivittäminen ei ole tarpeen.
- Se voidaan muuntaa helposti vastaamaan muuttuvia todellisen elämän tilanteita.
- Se on toistettavissa siten, että peräkkäiset ajot samaa ohjelmaversiota vasten antaa samat tulokset

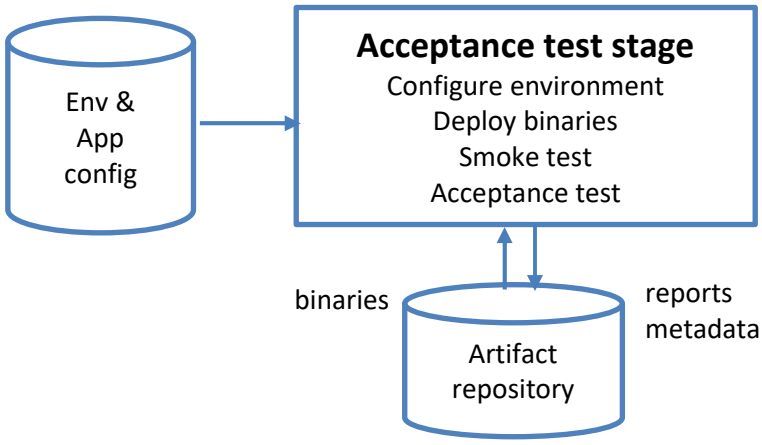
Hyvän kapasiteettitestin luominen ei ole helppoa, mutta on joitakin asioita, joiden avulla voidaan lähteä liikkeelle. Eräs malli on muokata ajossa olevista hyväksyntätesteistä testejä kapasiteettitestausta varten. Testiä voidaan esimerkiksi muokata siten, että suoritetaan *toimintoa x* tietyssä luupissa ja samanaikaisesti kuormitetaan järjestelmää keinotekoisesti. Tämän lisäksi järjestelmään voidaan kytkeä testin ulkopuolisia mittareita, jotka mittaavat erinäisiä asioita. [1.s.238]

6.4 Kapasiteettitestien lisääminen julkaisuputkeen

Mikäli kapasiteettitestit voitaisiin ajaa muutamassa sekunnissa, pitäisi ne ilman muuta suorittaa jo julkaisuputken ensimmäisessä vaiheessa nopeimman palautteen saamiseksi. Tähän voidaan osittain pyrkiä suojaamalla tunnistetut suorituskykyyn vaikuttavat kipupisteet ja luomalla niiden toimintaa mittaavat, julkaisuputken ensimmäisessä vaiheessa suoritettavat ”vartija testit”. Nämä ovat eräänlaisia nopean palautteen ensitestejä suorituskyvyn testaukseen antaen palautetta mahdollisista tulevista suorituskykyongelmista. [1.s.244-245]

Useimmat kapasiteettitestit eivät kuitenkaan ole soveltuvia ajettavaksi julkaisuputken ensimmäisessä vaiheessa johtuen niiden kestosta, sekä vaadituista resursseista. Hyväksyntätestausvaiheessa niitä voidaan suorittaa, mikäli ne pysyvät yksinkertaisina ja kestoltaan kohtuullisina. Kapasiteettitestauksen luotettavuuden parantamiseksi, tulee kaikki ulkopuoliset vaikutukset testaukseen minimoida. Epäluotettavuutta tuovat esimerkiksi samalla koneella käynnissä olevat useat rinnakkaisia testausprosessit. [1.s.245-246]

Erillinen testausvaihe voi useissa tilanteissa olla järkevä vaihtoehto automatisoidulle kapasiteettitestaukselle. Vaiheen käsittely putkessa riippuu kuitenkin täysin projektista. Esimerkiksi suorituskykykriittisissä ohjelmissa voi olla perusteltua rikkoa käännös, mikäli kapasiteettitestit eivät mene läpi. Usein sen tarkoituksena on kuitenkin enemmän monitoroida ohjelman suorituskyvyllisiä ominaisuuksia, joiden perusteella käännöksen kohtalo voidaan päättää tapauskohtaisesti. Periaatekuva kapasiteettivaiheesta on esitetty kuvassa 16. [1.s.246]



Kuva 16. Kapasiteettitestausvaihe [1.s.246]

7 Testidata

Testidata näyttelee tärkeää roolia niin automaattisessa-, kuin myös manuaalitestauksessakin. Usein joudutaankin miettimään vaatimuksia datalle, jolla testataan varsinainen toiminta, toiminnan rajat sekä ohjelmalle annettavat laittomat syötteet. Testit tarvitsevat myös dataa, jolla järjestelmä ajetaan erinäisiin vikatiloihin tarkempia tutkimuksia varten. Nämä ovat pohdinnan arvoisia kysymyksiä, koska hyvin suunniteltu testidata mahdollistaa järjestelmän kattavan testauksen, mutta toisaalta huonosti suunniteltuna se voi aiheuttaa esimerkiksi hankalaselkoiseen tietokantakytköksen vuoksi ongelmia testien luotettavuudelle. [1.s.334-335]

Testauksen tehokkuutta silmällä pitäen kannattaa testidatan suunnitteluvaiheessa kiinnittää huomiota erityisesti datan suorituskykyyn. Lisäksi suunnittelussa tulee huomioida, että testidata ei estä testien eristämistä. Ideaalinen testi suoritetaan tarkasti määritellyissä olosuhteissa, joiden syötteet ovat hallittuja ja vasteet voidaan helposti todentaa. Tietokanta on pitkäaikainen tapa säilöä testidataa, mutta sen datasisältö voi olla erilainen testien eri suorituskerroilla. Muuttuva datasisältö voi aiheuttaa ongelmia esimerkiksi muutettaessa testien suoritusjärjestystä, tai muuttuneen testidatan vuoksi testien lähtötilanne ei ole aina tunnettu. Testidatan helppo hallittavuus on myös olennainen osa suunnittelua. Tällöin esimerkiksi tietokantojen varmuuskopioita tulee välttää viimeiseen saakka. [1.s.334-335]

7.1 Tietokantojen simulointi yksikkötestauksessa

On erittäin tärkeää, ettei yksikkötestejä ajeta todellista tietokantaa vasten. Tietokanta on mahdollista eristää testistä käyttämällä esimerkiksi testituplia sen koodin sijasta, jonka avulla normaalisti liityttävän tietokantaan. Yleisesti ottaen on hyvän ohjelmointitavan mukaista kapseloida tietokantaliitoksen toteuttava ohjelmakoodi (repository pattern). Käytännössä tämä tarkoittaa erillisen kerroksen luomista, joka erottaa ohjelman käytettävästä tietokannasta. Testejä varten tämä kerros voidaan korvata normaalia toimintaa simuloivalla komponentilla ilman kytkeytymistä todelliseen tietokantaan. [1.s.335]

Edellä mainittu malli ei ainoastaan tue testausta, vaan lisäksi sen avulla voidaan eristää tietokantaan liittyvä toiminnallisuus ohjelman muusta koodista. Sen tarkoituksena on myös varmistaa, että tietokantaan liittyvä koodi on yhdessä paikassa, jolloin koodikannan ylläpito on myös hel-

pompaa. Vaikka tätä lähestymistapaa ei käytettäisikään, on silti mahdollista simuloida tietokantaa esimerkiksi käyttämällä kolmannen osapuolen ohjelmia, jotka luovat muistissa olevan tietokantaemulaation. Tämänkin mallin käyttö on hyödyllinen koodiarkkitehtuurisessa mielessä, koska myös tällöin koodi täytyy kirjoittaa yhteensopivaksi eri tietokantatoteutusten kanssa. [1.s.336]

7.2 Testien ja datan välinen yhteys

Myös testidatan suhteen on tärkeää, että jokaisella testillä on tunnettu lähtötilanne. Ainoastaan tällöin voidaan varmuudella todentaa testin aikana tapahtunut muutos alkutilanteesta lopputilanteeseen ja varmistaa testin tulos. Tämä on yksinkertaista yhdelle testille, mutta vaatii suunnittelua monimutkaisille testikokonaisuuksille eteenkin silloin, kun testidata tulee tietokannasta. Yleisesti ottaen on olemassa kolmea eri perustyyppiä sille, kuinka testien tilat voidaan hallita. [1.s.336]

Ensimmäinen on testien eristäminen, jolloin testin tarvitsema data on näkyvissä pelkästään kyseiselle testille. Toisessa mallissa testi adaptoituu testidataan, jolloin testi toimii sen hetkisen datasisällön mukaisesti. Kolmannessa mallissa testit ajetaan aina tietyssä sekvenssissä, jolloin ne ovat riippuvaisia edeltävien testien tuloksista. Näistä kolmesta mallista suositelluin on ensimmäinen malli, koska tällöin testit ovat joustavampia ja niitä voidaan ajaa yksittäin ilman riippuvuuksia sekä myös rinnakkaisissa prosesseissa. Jälkimmäisten mallien käyttöä ei suositella, koska testikokonaisuuksista tulee huomattavasti monimutkaisempia ja testien rikkoontumisia huomattavasti vaikeampi selvittää. Lisäksi testien ylläpidosta tulee työläämpää ja ylläpitokustannukset kasvavat. [1.s.336]

7.3 Testien eristäminen

Testien eristämisellä jokaisesta testistä saadaan atominen, eli testi ei ole riippuvainen edellisten testien tuloksista ollakseen tietyssä lähtötilassa. Vastaavasti myöskään seuraavat testit eivät riipu edeltävän testin tuloksesta. Esimerkiksi yksikkötesteille eristäminen tällä tasolla on suhteellisen helppo saavuttaa, vaikka käytettäisiinkin tietokantoja. Yksinkertaisin ratkaisu on palauttaa testin jälkeen data aina sitä edeltävään tilanteeseen. [1.s.337]

Tämä voidaan tehdä helposti hyödyntämällä esimerkiksi relaatiotietokantojen (relational database) hallintajärjestelmien ominaisuuksia. Tietokantaliitännäisissä testeissä voidaan esimerkiksi testin alussa luoda transaktio, jonka puitteissa tehdään tarvittavat operaatiot ja lopuksi, riippumatta testin tuloksesta, suoritetaan transaktion peruutus (rollback), jolloin mitään muutoksia ei päädy kantaan saakka pysyvästi. Toinen tehokas lähestymismalli on luoda testiä varten tietokantaan oma nimetty datasetti ohjelman omaa toiminnallisuutta käyttäen. Tällöin ei ole vaaraa, että testi sotkisi muita testejä. Tämä toimii erityisen hyvin esimerkiksi tilanteissa, joissa testataan tiettyyn käyttäjätiliin liitettyä toiminnallisuutta ja tili voidaan poistaa testin päätyttyä. [1.s.337]

Joissakin tilanteissa voidaan ajatella, että luodaan yhtenäinen testidatasetti, jota mahdollisimman suuri määrä testejä voi käyttää. Perusajatuksena tälle voi olla, että jokaisen testin ei itsessään tarvitse käyttää tällöin aikaa testidatan alustukseen ja siivoukseen, jolloin testisetin ajoaikaa voidaan nopeuttaa ja testejä yksinkertaistaa. Tämä voi kuitenkin epäsuorasti sitoa testejä toisiinsa, koska ne täytyy tällöin suunnitella käyttämään samaa testidataa ja yhden testin muuttaminen voi vaatia muutoksia testidataan ja sitä kautta muihin testeihin. Käytännössä poikkeuksetta testien ylläpito vaikeutuu, kun testisetin koko kasvaa ja tällöin joudutaan palaamaan atomisiin testeihin. [1.s.338]

7.4 Testidatan käyttö testiautomaation eri tasoilla

Jokaisella testauksen kerroksella on omat tarkoituksensa osoittaa, että ohjelma täyttää sille annetut vaatimukset. Millaista dataa nämä testauksen kerrokset sitten vaativat ja kuinka sitä tulee hallinnoida?

7.4.1 Ensimmäisen vaiheen testit

Testidatan kannalta katsottuna on elintärkeää, että sen käyttö ensimmäisen vaiheen testeissä on äärimmäisen nopeaa, koska tämän vaiheen valmistumista suunnittelijat joutuvat eniten odottamaan ja testejä halutaan ajaa mahdollisimman paljon. Hyvät ensimmäisen vaiheen testit eivät vaadi laajoja datasettien ylös ajoja, ja mikäli näin on, se on tietynlainen ilmentymä huonosta testien suunnittelusta. Tehokkaat testit eivät ole data vetoisia, vaan sen sijaan ne tarvitsevat mahdollisimman vähän testidataa osoittaakseen järjestelmän toiminnan tietyssä tilanteessa. Enem-

män dataa tarvitseville testeille, data on hyvä generoida automaattisesti käyttäen testausta varten tehtyjä apuluokkia. Tällöin muutokset järjestelmän käyttämissä tietorakenteissa eivät saa aikaan kaaosta testitasolla. [1.s.339]

7.4.2 Hyväksyntätestaus

Jotta hyväksyntätason testeistä ei tule liian kankeita, tulee niiden yksikkötasoa monimutkaisempi testidata suunnitella huolellisesti. Testidataan pätevät kuitenkin samat säännöt verrattuna yksikkötesteihin, eli pyritään viimeiseen saakka välttämään monimutkaisia datarakenteita. Määriteltäessä testidataa hyväksyntätestausta varten, on data hyvä jakaa suoraan kolmeen eri osa-alueeseen. [1.s.340]

- Testikohtainen data ohjaa testin suoritusta ja edustaa testattavan tapauksen tilannekuvaa, jota testin on tarkoitus todentaa
- Testin referenssidata on oltava olemassa, mutta sillä on varsin vähäinen vaikutus testattavaan toiminnallisuuteen
- Koko ohjelmiston käyttämä referenssidata ei ole relevanttia testin toiminnallisuuden kannalta, mutta sitä tarvitaan esimerkiksi ohjelmiston käynnistämiseen

Testikohtaisen datan on hyvä olla uniikkia ja noudattaa strategiaa testien eristämisestä siten, että testi käynnistyy selkeästi määritellystä tilasta, joka on immuuni muille suoritettaville testeille. Testien referenssidata voi olla esimerkiksi esitäytettyä siemendataa, jota uudelleen käytetään useissa testeissä luomaan puitteet testien suorittamiselle, mutta mihin testi ei itsessään aiheuta muutoksia. Ohjelmatason referenssidata voi olla käytännössä mitä vaan dataa, kuten esimerkiksi tietokannan varmuuskopio. Ohjelmatason referenssidata tulee olla versioituna ja lisäksi tulee varmistaa, että niiden mahdolliset tietokantamigraatiot ovat tehtynä osana ohjelman käynnistysrutiineja, mikä onkin hyvä tapa testata automatisoituja tietokantamigraatioita. Vaikka testidatan rajat näiden kategorioiden välillä voi olla hämärä, on tärkeintä, ettei testidata pelkästään vaan ole jossakin hallitsemattomasti. [1.s.340]

Hyväksyntätason testejä ei ole hyvä alustaa lähtötilaan tietokannan varmuuskopion tai erillisen ohjelmakoodin avulla. Testit tulisi alustaa lähtötilaan käyttäen ainoastaan ohjelman omia rajapintoja, koska normaalin toimintalogiikan ohittavalla ohjelmakoodilla järjestelmä voidaan asettaa

epäjohdonmukaiseen tilaan. Käyttämällä ohjelman omia rajapintoja, on ohjelma aina normaalia toimintaa vastaavassa tilassa. [1.s.340-341]

7.4.3 Kapasiteettitestaus

Kapasiteettitestauksessa tarvittava data tuo omat haasteensa esimerkiksi testisyötteille tarvittavien datamäärien kasvaessa. Lisäksi sopivan referenssidatan määrittely useille samanaikaisesti ajettaville testeille voi olla haastavaa. Tämänkaltaisten datamäärien generointi automaattisilla mekanismeilla esimerkiksi käyttäen apuna hyväksyntätestidataa voi olla järkevää sen sijaan, että hallitaan suuria määriä kiinteää dataa. Tämä lähestymiskulma auttaa vahvistamaan hyväksyntätestauksessa käytettävää dataa sekä sen uudelleenkäyttöä mahdollisimman laajassa mittakaavassa. Strategia datan automaattisesta generoinnista perustuen hyväksyntätesteissä käytettyyn dataan juontaa juurensa ajatuksesta, jonka mukaan hyväksyntätestauksessa tapahtuvat vuorovaikutukset sekä niiden puitteissa käsitelty data ovat pääasiassa suoritettavia vaatimusmäärittelyjä järjestelmän toiminnalle. Kapasiteettitestaus perustuu vastaaviin vuorovaikutuksiin, vaikka sen puitteissa mitataan hieman eri asioita. Näillä perustein kapasiteettitestauksessa käytettävä data voidaan täysin perustaa vastaavaan dataan hyväksyntätestauksen kanssa. [1.s.342]

Eräs strategia kapasiteettitestauksessa tarvittavan testidatan hallintaan voisi olla datan monistaminen tiettyjen työkalujen avulla tietyin parametrein valittujen hyväksyntätestien testidatasta. Tällöin järjestelmää voidaan kuormittaa useilla generoiduilla syötteillä, jotka perustuvat yhteen tiettyyn testiin. Lisäksi malli auttaa keskittymään datasettien hallinnan sijaan luomaan tehokasta dataa yksittäisille syötteille. [1.s.342]

8 Testausstrategian suunnittelu ohjelmistotuotteelle

Lyhyesti sanottuna testausstrategian suunnittelu on prosessi, jonka puitteissa tunnistetaan sekä priorisoidaan projektiin liittyvät mahdolliset riskit sekä päätetään mahdolliset toimenpiteet niiden minimoimiseksi. Hyvällä testausstrategialla voi olla paljon hyviä vaikutuksia projektin onnistumiseen ja parhaimmillaan jokainen työntekijä tietää oman roolinsa strategisen tahtotilan saavuttamiseksi. Kattavaksi suunniteltu testaus tuo luottamusta ohjelman toimintaan, minimoi ylläpitokustannuksia sekä parantaa ohjelmiston mainetta. Kattava testiautomaatio luo ajantasaisen dokumentaation ohjelman toiminnasta sen sijaan, että kehitystiimillä on oletus, kuinka ohjelman pitäisi toimia. Ohjelmistotestausstrategiaa voidaan pitää yhtenä prosessin vaiheena, jossa tuotteen testaukseen liittyvä visio muutetaan konkreettiseksi tekemiseksi. [1.s.84; 12.s.82, 53].

Strategia laaditaan toiminta-ajatuksen mukaisesti ja sitä pitää voida muuttaa ajoissa, mikäli huomataan, että valitut toimintamallit johtavat joko umpikujaan, tai vievät väärälle uralle vision saavuttamisen kannalta. Strategisten linjausten konkretisointi alemman tason tehtäviksi on yleensä päälliköiden ja esimiesten vastuulla. Esimerkiksi alemman tason vaatimuksen suoritettavista testispesifikaatioista tulee periytyä ylemmän tason linjauksesta, jonka puitteissa testaussuunnitelma kirjoitetaan yleisellä tasolla sisältäen linkityksen suoritettavina testeinä toimiviin tarkempiin kuvauksiin. Hyväksytty strategia esitetään henkilöstölle, mutta tarkemmat toimenpidesuunnitelmat voidaan pitää pelkästään johdon tietona. [11. s. 116, 125; 12.s.251]

8.1 Analyysit strategian perustana

Strategia voidaan laatia kulloinkin sopivaksi katsotulle aikavälille, mutta yleinen aikaväli strategian laadinnalle on kolme vuotta, minkä jälkeen sitä päivitetään vuosittain. Vuosittaisen strategiatyön lisäksi merkittävät muutokset toimintaympäristössä voivat johtaa strategian korjaamiseen. Normaalisti strategiatyö alkaa ja päättyy organisaation johdon toimesta. Päävastuu strategiatyöstä on organisaation johtajalla, käyttäen apunaan organisaation keskijohtoa esimerkiksi taustatiedon keruuseen. [12.s.250]

Strategisen suunnittelun kannalta oleellinen tietolähde onkin hyvin usein organisaation keskijohto, jolla on paljon suunnittelussa tarvittavaa operatiivista kokemusta ja näkemystä. Tätä voimavaraa onkin hyvä hyödyntää aktiivisesti strategisessa suunnittelussa. Mikäli keskijohdon näke-

mys poikkeaa radikaalisti osaston tilasta tai ympäristötekijöistä verrattuna johdon tai ulkopuolisen tahon tekemään selvitykseen, on johdolla tällöin väärää tietoa strategian perustasta tai harhakäsityksiä organisaation osastojen tilasta [11. s. 112-113].

Strategisen aseman määrittäminen on ensimmäinen osa strategiatyötä sisältäen sisäisen ja ulkoisen analyysin. Seuraavassa vaiheessa asetetaan tai valitaan päämäärät sekä tavoitteet ja lopuksi tehdään toimenpideohjelman määritys. Strategiatyökaluja tehokkuuden parantamiseen on useita. Tero Vuorinen on kirjassaan esitellyt 5 eri työkalua tähän tarkoitukseen, joista tähän työhön käytettäviksi on valittu liitteessä 9 esitetyt SWOT-analyysi, tasapainotettu mittaristo sekä strategiakartta. [12.s.250]

8.2 Testiautomaatio osana tuotteen testausstrategiaa

Testiautomaatio on hyvä ottaa osaksi ohjelmiston testausstrategiaa jo heti projektin alkumetreillä. Sen mukaan ottaminen ei ole myöhäistä myöhäisemmissäkään vaiheissa, vaikkakin ponnistelut tuossa tilanteessa voivat olla jo huomattavasti suurempia vastaavien hyötyjen saavuttamiseksi. Usein voidaan päätyä tilanteeseen, jossa ohjelmakoodi ei ole automaattisesti testattavaa ja vaatii huomattavia rakennemuutoksia testattavuuden parantamiseksi. Lisäksi tulee aina huomioida varsinaisen kehitystyön samanaikainen eteneminen, kun ohjelmalle opetellaan kirjoittamaan automaattitestejä. [1.s.84]

Suunniteltaessa ohjelmistotuotteen testausta ohjaavia strategisia linjauksia, tulee testiautomaatio liittymään näihin linjauksiin tavalla tai toisella. Selkeiden strategisten linjausten puuttumisen myötä voi ohjelmiston laatu heiketä merkittävästi esimerkiksi, mikäli ohjelmiston toiminnallisten- sekä ei-toiminnallisten vaatimusten luotetaan olevan täytettyinä pelkän manuaalitestauksen perusteella. Strategisista linjauksista periytyvien toimenpiteiden sekä prosessien avulla tulee myös huolehtia, että tehdyt testit eivät pääse vanhentumaan puuttuvan ylläpidon vuoksi. Vanhentuneista testeistä seuraa yleensä turvautuminen manuaalitestaukseen testikattavuuden parantamiseksi. Edellä mainituista syistä ohjelmiston testausstrategia onkin hyvä laatia siten, että se huomioi kattavan ja huolella ylläpidetyn automaation osana tuotteen laatua. [1.s.83]

Myös resursoinnilla on oleellinen merkitys laadittaessa strategisia tavoitteita. Testaus on koko tiimiä koskevaa, rajat ylittävää toimintaa, johon kaikkien tulee osallistua heti projektin alkumet-

reiltä sen päättymiseen saakka. Erittäin hyvä strategia testiautomaation toteutukselle on sisällyttää siihen testausta kaikilla sen tasoilla sisältäen yksikkö-, integraatio- sekä hyväksyntätestaus. Strategiaa suunniteltaessa tulee kuitenkin huomioida tiimien kyky tuottaa testiautomaatiota kaikilla näillä tasoilla. Yleisesti ottaen ohjelman laadun parantaminen tarkoittaa jatkuvaa strategiatyötä sekä nopeaa reagointia muuttuviin tilanteisiin. [1.s.83]

Testiautomaation tarkoituksena ei ole pelkästään testata ohjelmaa toiminnallisesta näkökulmasta, vaan siihen on hyvä sisällyttää myös ei-toiminnallisten vaatimusten testaus. Automatisoituja suorituskykytestejä tarvitaan, jotta esimerkiksi muutoksista johtuva ohjelman suorituskyvyn lasku voidaan huomata ennen vikojen uimista syvemmälle ohjelman rakenteisiin. Mikäli tämän kaltaisia asioita ei huomioida osana tuotteen testausstrategiaa, voivat ne aiheuttaa ongelmia jossakin vaiheessa tuotteen elinkaarta. [1.s.84]

8.2.1 Uudet projektit

Uusissa projekteissa testausstrategiaan tulee kirjata automaattitestien tekeminen toiminnoille heti projektin alkumetreillä. Strategiatyön perusteella tulee heti alkuun valita ohjelmiston testiautomaatiota parhaiten tukevat testialustat, menetelmät sekä muut testaukseen tarvittavat työkalut. [1.s.92]

8.2.2 Käynnissä olevat projektit (mid-projects)

Valitettavan usein ohjelmistoprojektin käynnistyessä ohjelmiston koodikantaa aletaan kasvattaa huomattavalla vauhdilla ja pian ollaan tilanteessa, jossa aliresursoitu tiimi kehittää alati muuttuvaa ohjelmistoa kovassa toimituspaineessa. Usein vasta tässä tilanteessa ymmärretäänkin testiautomaation tärkeys. Kun tähän tilanteeseen on ajauduttu, hyvä strategia testiautomaation mukaan ottamiseksi on tuoda se aluksi ominaisuuksiin, joilla on asiakkaalle eniten käyttöarvoa. Tällöin strategiset tavoitteet testiautomaatiolle kannattaa asettaa siten, että testit kattavat aluksi yleisimmät käyttötapaukset. Tämän strategian toteutus edellyttää erittäin kattavaa testiautomaatiota yksikkö sekä integraatiotasoilla. Automatisoinnin puitteissa kannattaa myös yrittää maksimoida testien avulla suoritettavan toiminnallisen koodin lukumäärä. Vasta seuraavassa vaiheessa testaus laajennetaan tasolle, jolla se tyypillisesti on työstettäessä uusia ominaisuuksia. [1.s.87; 94]

Tämän strategian mukaisesti testiautomaation on hyvä painella käyttöliittymästä mahdollisimman monta painiketta ja täyttää mahdollisimman monta syötekenttää. Vaikka jokaista syötettä ei verifioitaisikaan, niin antaa tämä hyvän lisän manuaalitestaukselle, jolloin jokaista painiketta ei tarvitse enää käydä läpi. Näin voidaan saada tietynlainen varmuus ohjelman perustoiminnallisuudesta, vaikka jokaista ohjelman toimintoa ei testattaisikaan syvällisesti. [1.s.94]

Ohjelman perustoiminnallisuuden automatisoinnin jälkeen, voi olla vaikea päättää lähdetäänkö automaatiota laajentamaan kattamaan muita toiminnallisia ohjelmapolkuja vai mahdollisia virhetilanteita. Mikäli ohjelma on stabiilissa tilassa, on järkevää lisätä automaatiota kattamaan vaihtoehtoisia polkuja. Mikäli ohjelma puolestaan sisältää paljon ohjelmistovikoja ja kaatuu usein, voi järkevää olla lähteä tukemaan kehitystiimiä automatisoimalla virhetilanteita kattavia testejä. [1.s.88]

Lisäksi luottamusta ohjelmistoregressioita vastaan voidaan parantaa kirjoittamalla mahdollisuuksien mukaan jokaisen löytyneen ohjelmistovian löytävä automaattitesti. Mallin avulla voidaan tehokkaasti löytää esimerkiksi koodikantojen yhdistämisestä (merge) johtuvat, jo korjattujen ohjelmistovikojen uudelleenesiintymiset järjestelmässä. Malli auttaa myös lisäämään testiautomaation kattavuutta todennäköisissä vikapaikoissa. [1.s.212]

Edellä esitellyn strategian valitseminen tarkoittaa käytännössä kattavampaa ja yksityiskohtaisempaa manuaalitestausta, kunnes automatisointi on saatu vietyä syvällisemmälle tasolle. Voimakkaasti muuttuvien ominaisuuksien testaaminen sekä manuaalisesti, että automaattisesti voi olla tietyissä tilanteissa haastavaa, koska myös testin sisältö muuttuu toiminnallisuuden muuttuessa. Kun esimerkiksi manuaalitestin sisältö ei enää muutu, voidaan se automatisoida. Vastakohtaisesti mikäli automaattitestiä täytyy päivittää usein, todennäköisesti myös toiminnallisuus sen takana muuttuu. Tämänkaltaisissa tilanteissa voi olla hyvä keskustella kehitystiimin kanssa testin väliaikaisesta ohittamisesta riittävien kommenttien kanssa, kunnes toiminta ei enää muutu. Mikäli toiminnan stabiloiduttua testi ei ole enää validi tai toiminta on muuttunut radikaalisti, poistetaan testi ja tehdään uusi vastaamaan uutta toiminnallisuutta. [1.s.94]

8.2.3 Ylläpidossa olevat projektit (legacy systems)

Järjestelmiin, jotka jo vuosia ovat painottuneet manuaaliseen testaukseen, kannattaa automatisoida ainoastaan tehtyihin muutoksiin liittyvät toiminnot. Sovellettaessa strategiaa on tärkeää

kohdistaa tukitoimet järjestelmän avain ja kipukohtiin, joiden toiminnallisuus on kriittisessä asemassa ja joiden suojaaminen regressioita vastaan on tärkeää. [1.s.95]

Mikäli testiautomaatiota laajennetaan muutettavien toimintojen ulkopuolelle, kannattaa se toteuttaa kahdessa eri vaiheessa. Aluksi tehdään testit kattamaan yleisimmät vikapaikat ja toiminnan kannalta kriittinen perustoiminnallisuus. Vasta tämän jälkeen keskitytään kattavammin tiettyä ominaisuutta testaaviin testeihin. Uusien ominaisuuksien kehittäminen sekä testaus on hyvä toteuttaa käyttäen uusille projekteille määriteltyä testausstrategiaa. Vanhojen järjestelmien automaattitestaus voi joskus olla erittäin hankalaa johtuen koodiarkkitehtuurista, jota ei ole suunniteltu testiautomaatiota varten. Siten on yleistä, että muutos yhdessä kohtaa voi rikkoa toiminnallisuutta jossain odottamattomassa paikassa. Tämänkaltaisia ei-toivottujen sivuvaikutusten löytämistä voidaan parantaa laajentamalla testiautomaatiota toimintoihin, joihin muutettu koodi vaikuttaa suorasti tai epäsuorasti. [1.s.95-96]

On tärkeää muistaa, että automatisointia lisätään pelkästään sinne, missä se tuottaa lisäarvoa kokonaisuudelle. Kehitettävä ohjelmakoodi voidaan olennaisesti jakaa kahteen pääosaan. Ohjelman käyttäjävaatimukset toteuttavaan kerrokseen sekä kirjastokoodiin tämän kerroksen alapuolella. Valtaosa ohjelmistojen regressioista johtuu kirjastotason muutoksista. Silloin kun ohjelmaan lisättävät uudet ominaisuudet eivät muuta alempaa kooditasoa, voi alemman kooditason testiautomaation lisääminen olla vähemmän arvoa tuottavaa. [1.s.96]

9 Tutkimus ohjelmistotuotantoprosessin kehittämisestä

Tutkimusongelmana työssä oli testauksen saumattomampi integrointi varsinaiseen ohjelmistotuotantoprosessiin. Työn puitteissa tehty tutkimus koostui nykytilanteen kartoittamisesta sekä ohjelmistotuotantoprosessimallin muutoksista, jotta testaus integroituisi paremmin osaksi kehitysprosessia. Tutkimusmenetelmä nykytilanteen kartoittamista varten valittiin kvantitatiivinen eli laadullinen tutkimus. Tutkimusstrategiaksi valittiin case-tutkimus, jonka puitteissa paneuduttiin ilmiötä edustavaan tapaukseen. Aineisto tutkimukseen saatiin haastattelemalla sekä havainnoimalla. Haastatteluiden avulla oli tarkoituksena kartoittaan nykytilan ongelmakohdat sekä mahdolliset kehitysideat ongelmien selättämiseksi.

9.1 Laadullinen tutkimus

Puhuttaessa laadullisesta tutkimuksesta, liittyy siihen vahvasti käsitteellisyys merkityksestä sekä merkitykselliseen toimintaan liittyvästä tutkimuksesta [9.s.31]. Laadullisen tutkimuksen pyrkimyksenä on parantaa tai uudistaa tutkimuksen alla olevaa toimintaa. Laadullinen tutkimus on erottamaton kokonaisuus, jonka puitteissa kerätään ja analysoidaan tutkittavaan ilmiöön liittyvä aineisto. [8.s.68]

Tämän tutkimuksen tutkimusmenetelmäksi valittiin laadullinen tutkimus, koska sen avulla pyritään parantamaan tai kehittämään tutkittavaa kohdetta. Tässä työssä tutkimus kohdennettiin merkitykselliseen toimintaan, eli ohjelmistokehitysprosessiin. Työn puitteissa toimintaa tutkittiin keräämällä ja analysoimalla toimintaa koskevaa aineistoa. Tutkimuksen tavoitteena oli myös kehittää prosessia.

Aineiston keruu laadullista tutkimusta varten voidaan toteuttaa usein eri menetelmin. Riippuen tutkimusasetelmasta, aineistoa voidaan kerätä yksistään käyttäen tiettyä metodia tai useita menetelmiä yhdistelemällä. Aineiston keruussa voidaan hyödyntää esimerkiksi kyselyitä, haastatteluita sekä olemassa olevan dokumentaation havainnointia. Yleisinä menetelminä käytetyissä kyselyissä ja haastatteluissa on mahdollista selvittää esimerkiksi kohteena olevien henkilöiden ajattelumalleja tai kokemuksia tutkittavasta ilmiöstä. Edellä mainittujen lisäksi myös havainnointia käytetään yleisenä menetelmänä aineiston keräämiseksi. Havainnointi perustuu tutkimuskohteen seurantaan sekä tutkimukseen liittyvien havaintojen tekemiseen tutkimuskohteesta. [8.s.71]

Tämän tutkimuksen puitteissa aineistoa kerättiin haastattelemalla prosessin osana toimivia henkilöitä. Haastatteluiden perusteella saatiin muodostettua kuva nykyprosessin tilasta perustuen eri asemissa toimivien henkilöiden näkemyksiin. Yhdistettynä haastatteluihin tutkimuksessa käytettiin myös havainnoimalla saatua aineistoa.

Tutkimuksen puitteissa kerättyä aineistoa on hyvä tarkastella analyyttisesti yhtenä kokonaisuutena, jotta sen avulla voitaisiin saada kokonaiskäsitys tutkittavasta ilmiöstä. Analyyttisessä mielessä laadullisen analyysin voidaan ajatella koostuvan kahdesta eri vaiheesta, jotka käytännön tasolla ovat kuitenkin vahvasti nivoutuneet yhteen. Aluksi havainnot pelkistetään perustuen aineiston tarkasteluun tietystä näkökulmasta teoreettisen viitekehyksen ohjaamana. Havaintojen pelkistys puolestaan tuottaa avaimet varsinaisen ongelman ratkaisemiseen. Valitun tutkimusmetodin avulla määritellään havaintojen tuottamiseen, tulkitsemiseen sekä arviointiin liittyvät käytännöt sekä toimenpiteet. Lisäksi teoreettiseen viitekehykseen soveltuvan tutkimusmetodin avulla erotellaan havainnot varsinaisista tutkimustuloksista. [9.s.32-33, 63]

Tutkimustavaksi valittiin tapaustutkimus, koska sen puitteissa tutkitaan aina tässä hetkessä olevaa ilmiötä, josta halutaan muodostaa syvällinen kuvaus. Tutkimus toteutetaan ilmiölle luonnollisessa asiayhteydessä ja sen aineisto koostetaan yhdistelemällä eri aineistoja ja menetelmiä. [10.s.54]

Tapaustutkimus on pääsääntöisesti laadullista tutkimusta ja siinä tuotetaan ongelmaan ratkaisu, mutta tutkijan tehtävänä ei ole suorittaa käytännön työtä ongelman ratkaisemiseksi. Tapaustutkimuksessa yhdistyvät kvalitatiivinen sekä kvantitatiivinen tutkimus, joten se ei varsinaisesti ole oma lähestymistapansa. Sillä ei myöskään ole omaa tiettyä metodologiaa tai tutkimusmenetelmiä. Tapaustutkimus on yhdistelmä laadullisen ja määrällisen tutkimuksen tiedonkeruu sekä analyysimenetelmiä. [10.s.9,15]

Tapaustutkimuksen vaiheet etenevät noudattaen perinteisen laadullisen ja määrällisen tutkimuksen vaiheita. Tutkimus lähtee liikkeelle tutkimusongelman määrittelystä, sekä ongelmasta johdetuista tutkimuskysymyksistä. Seuraavaksi valitaan tutkittava tapaus tai tapaukset. Tämän jälkeen voidaan valita tiedonkeruu- sekä analysointimenetelmät. Suunnittelun jälkeen siirrytään toteutusvaiheeseen, jonka puitteessa kerätään materiaali tutkimusta, analysointia sekä raportointia varten. [10.s.59-60]

Teoreettisen viitekehyksen tehtävänä on toimia ohjenuorana aineiston keruulle sekä analyysille. Teoreettisen viitekehyksen perusteella päätetään kerättävän aineiston tyyppi sekä menetelmät,

joilla sitä analysoidaan. Teoreettisen viitekehyksen sekä tutkimusmetodin yhteisvaikutus tutkimuksen tekemiseen on merkittävä, koska riippuen valinnoista, tutkimusta voidaan tarkastella useista eri näkökulmista. Laadullista tutkimusta varten kerättävän aineiston täytyykin antaa mahdollisuus tarkastella asioita useista eri tulokulmista. [9.s.63-64]

Teoreettisena viitekehyksenä tutkimukselle on ollut jatkuvan integroinnin sekä toimituksen prosessimalli sekä henkilöstön motivointi sekä sitoutumisen tärkeys prosessin onnistumisen kannalta. Tutkimusta varten tehty aineiston keruu suunniteltiin ja analysoitiin perustuen teoreettisen viitekehityksen rajauksiin.

9.1.1 Tutkimusaineiston kerääminen

Työhön liittyvän tutkimusaineiston kerääminen perustui pääpainotteisesti organisaation eri asemissa toimivien henkilöiden haastatteluihin. Haastatteluiden perusteella pyrittiin saamaan läpileikkauskuva testauksen sekä ohjelmistokehityksen keskinäisistä suhteista nykytilassa sekä mahdollisista parannusehdotuksista. [8.s.72]

Teemahaastattelu perustuu etukäteen valittuihin teemoihin, sekä niitä tarkentaviin kysymyksiin. Haastattelun teemat perustuvat teoreettiseen viitekehykseen. Haastattelun luonne määrittää, pitäydytäänkö tiukasti etukäteen päätetyissä kysymyksissä, vai sallitaanko intuitiivinen eteneminen poiketen alkuperäisistä kysymyksistä. [8.s.75]

Varsinainen haastattelu toteutettiin lähettämällä haastattelukysymykset ennakkoon haastateltaville, jotta he saivat etukäteen paremmin valmistautua varsinaiseen haastattelutapahtumaan. Kaikille haastateltaville lähetettiin samat kysymykset riippumatta haastateltavan asemasta prosessissa. Työn puitteissa haastateltiin kolmea henkilöä, joiden asema sekä kokemus alalta on esitetty taulukossa 1. Haastattelun osa-alueisiin kuuluivat testausstrategia, testauksen integroituminen kehitystyöhön sekä testauksen tuottama palaute. Haastattelulomake on esitetty liitteessä 11.

Taulukko 1. Haastateltavat

Toimenkuva	Yrityksen palveluksessa	Kokemus alalta
Tiimitestaaja	1 V	1 V
Suunnittelupäällikkö	15 V	15 V
Sovelluskehittäjä	10 V	14 V

Haastatteluiden lisäksi tutkimukseen liittyvää aineistoa voidaan täydentää havainnoimalla tutkimusongelmaan liittyvää aineistoa. Vaikka havainnointi yksinään onkin haasteellinen menetelmä tiedon keruuseen, toimii se hyvin rinnalla käytettävänä tiedonkeruumenetelmänä. Havainnointia voidaan myös käyttää muista tiedonkeruumenetelmistä saadun tiedon yhdistämistä oikeisiin asiayhteyksiin. Havainnointia voidaan suorittaa esimerkiksi osallistumalla toimintaan tai ilman osallistumista. [8.s.81]

Työssä varsinainen havainnointi perustui toiminnan seuraamiseen sivusta, sekä olemalla itse osana toimintaa. Lisäksi havainnoinnin piiriin kuului aiheeseen liittyvän dokumentaation tutkiminen, mikä auttoi muodostamaan esimerkiksi käsitystä testauksen suunnitelmallisuudesta.

9.1.2 Tutkimuksesta saadun materiaalin analysointi

Laadullisen tutkimuksen tuloksia voidaan analysoida usein eri menetelmin, mutta yleisin menetelmä aineiston analysoinnille on sisältöanalyysi. Malli soveltuu lähes kaikkiin tutkimuksiin. Menetelmän avulla aineistojen, kuten dokumenttien tai haastatteluiden sisällön analysointi voidaan tehdä objektiivisesti ja systemaattisesti. [8.s.92,103]

Sisältöanalyysin periaatteena on tarkastella aineistoa eritellen, sekä etsien yhtäläisyyksiä sekä eroavaisuuksia aineistosta. Tutkittavasta ilmiöstä havaittujen asioiden tiivistäminen selkeäksi kokonaisuudeksi on myös olennainen osa sisältöanalyysiä. Menetelmiä aineiston analysoinnille sisältöanalyysissä ovat aineistolähtöinen analyysi, teorialähtöinen analyysi sekä teoriaohjaava analyysi. [8.s. 98,105]

Työn puitteissa tehtävän analyysin muodoksi valittiin teorialähtöinen analyysi, koska tällöin analyysi pohjautuu CI/CD-prosessimallin teoriaan ja analyysia ohjasi aikaisemman tiedon pohjalta

määritelty viitekehys. Tutkittavaksi ilmiöksi valikoitui testauksen integroituminen ohjelmistokehitysprosessiin. Haastattelut taltioitiin ja niistä kirjattiin yksityiskohtaiset vastaukset (liitteet 12-14). Tutkittavaan ilmiöön liittyvät vastaukset sekä muista havainnoista saadut huomiot koottiin haastattelussa käytettyjen teemojen mukaisesti. Tutkimuksen puitteissa saatu tieto käytiin läpi ja analysoitiin teema-alueittain ja lopputuloksena saatiin yhteenveto, jonka avulla voitiin lähteä miettimään toimenpiteitä prosessin parantamiseksi.

9.2 Haastattelut

Haastattelut pidettiin videoneuvottelun muodossa siten, että jokainen haastattelu oli oma itsenäinen sessio. Haastattelut tallennettiin videotiedostoiksi, sekä litteroitiin kirjalliseen muotoon. Jokaiselle haastateltavalle lähetettiin kysymyslomake ennen varsinaista haastattelutapahtumaa.

9.2.1 Testausstrategia -osio

Testausstrategia -osiolla oli tarkoitus kartoittaa yrityksen ohjelmistotestausstrategian tuntemusta eri asemissa työskentelevien henkilöiden osalta. Lisäksi haluttiin saada käsitys siitä, millaisena haastateltavat henkilöt näkivät oman roolinsa strategisten tavoitteiden saavuttamiseksi. Kysymyksen avulla voidaan kartoittaa sitä, kokeeko haastateltava henkilö olevansa osallinen jotain suurempaa, vai pelkästään prosessin irrallinen osa ilman selkeää päämäärää.

Haastateltavilta henkilöiltä haluttiin myös näkemykset yleisesti strategian laadinnassa apuna käytettävän SWOT -analyysin nelikenttiin, joihin kuuluivat vahvuudet, heikkoudet, uhat sekä mahdollisuudet. Näitä vastauksia puolestaan tarvittiin tueksi testausstrategiaa varten laadittavalle SWOT -analyysille.

Tulokset

Ohjelmistotestausstrategiaa koskevan osion päällimmäinen anti oli, että nykyisellään yrityksessä ei ole valmiina kirjattua strategiaa, jonka puitteissa toimintaa kehitettäisiin johdonmukaisesti eteenpäin. Ylemmällä tasolla työskentelevillä henkilöillä oma osuus strategisten tavoitteiden saavuttamiseksi oli suhteellisen selkeä. Suorittavassa portaassa oma toimenkuva yhteisössä oli hyvinkin selkeä, mutta toimenkuvan suhde strategisten tavoitteiden saavuttamiseksi hieman epäselvä.

Testauksen vahvuuksiksi haastatteluiden puitteissa esille nousivat hyvät työkaverit, tiivis yhteisöllisyys ja vahva osaamisen taso. Lisäksi avun koettiin olevan aina tarvittaessa helposti saatavilla, jolloin asioiden kanssa ei tarvinnut jäädä yksin painiskelemaan. Vahvuudeksi koettiin myös hyvät, testauksen käytössä olevat työkalut, sekä monivaiheinen testaus sisältäen myös asiakkaiden koneilla tapahtuva testaustoiminta.

Heikkouksina puolestaan nähtiin, että testiautomaation kehitys ei nykyisellään ole kovinkaan suunnitelmallista. Lisäksi sen suhteellisen vähäinen määrä nykyisellään koettiin heikkoudeksi. Heikkoudeksi koettiin myös vaihteleva osaamisen taso testiautomaation kehityksessä. Testaussimulaattoreiden osalta heikkoudeksi koettiin niiden jälkeen jääminen kehityksestä, jolloin esimerkiksi toteutetuille ominaisuuksille ei välttämättä ole kunnollista tukea testausta ajatellen. Sulautettujen järjestelmien osalta yksikkötestien puuttuminen lähes kokonaan koettiin heikkoudeksi, kuten myös kunnollisen testausstrategian puuttuminen.

Ohjelmistotestaukseen liittyvät uhkakuvat liittyivät simulaattoreihin sekä mahdollisen taloustilanteen tuomiin haasteisiin työkalujen hankinnan suhteen. Lisäksi uhkana nähtiin nykyisten automaattitestien ylläpidettävyys sekä monimutkaistuvien automaatiojärjestelmien sekä anturointien vaikutukset testaukseen. Mahdollisuuksina puolestaan nähtiin kunnollisen testausstrategian laadinta.

9.2.2 Testauksen integroituminen kehitystyöhön -osio

Toinen haastattelun teemoista pureutui testauksen integroitumiseen varsinaiseen ohjelmistokehitystyöhön. Haastattelussa haluttiin selvittää, millaiseksi testauksen rooli koettiin nykyisessä ohjelmistokehitysprosessissa. Tuotteen kokonaisvaltainen testaaminen yksikkötestauksesta hyväk-

syntätestaukseen vaatii huomattavan määrän tietotaitoa ohjelmoinnista sekä ymmärrystä järjestelmän toiminnasta yleensä. Edellytyksiä tämänkaltaiselle kokonaisvaltaiselle testaukselle haluttiin selvittää kysymällä mielipiteitä testaus- sekä suunnitteluvastuun jakautumiselle tiimissä.

Ohjelmiston automaattiseen testattavuuteen vaikuttaa merkittävästi testauksen huomioiminen heti suunnittelun käynnistyttyä, joten myös tätä osa-aluetta haluttiin selvittää haastattelussa. Testausstrategiaan merkittävästi vaikuttava seikka on käytettävissä oleva resursointi, koska hyvin harvoin testauksen resurssit ovat riittäviä ohjelmiston täydellistä testaamista varten. Ohjelmistokehityksen sekä testauksen välistä resursointiin liittyvää tasapainoa haluttiin myös selvittää, jotta voitaisiin muodostaa realistinen käsitys siitä, millainen osa ohjelmistosta on ylipäänsä mahdollista testata.

Koska testauksen tehtävänä on toimia turvaverkkona kehitykselle ja sitä kautta varmistaa asiakkaalle toimitettava laadukas ohjelmisto, haluttiin lopuksi selvittää mielikuvia testauksen tämänhetkisestä kyvystä tukea kehitysprosessia. Ajatuksen tasolla kysymyksellä haluttiin selvittää juuri testauksen kykyä toimia turvaverkkona koodimuutoksille, sekä puutteita ja parannuskeinoja, jotta turvaverkosta voitaisiin saada entistä parempi.

Tulokset

Testauksen rooli osana kehitystyötä koettiin merkittävänä osana lähes kaikkea kehitystyötä. Esille nousi esimerkiksi manuaalisen testauksen tuoma nopea palaute suunnittelijoille tehtäessä uusia toimintoja tai korjattaessa nykyistä toiminnallisuutta. Toisaalta kehityksen ja testauksen välinen sidos koettiin nykyisellään vähän löyhäksi, mitä täytyisi parantaa entisestään esimerkiksi kehityksen kanssa samassa rintamassa etenevällä testiautomaatiolla.

Vastaukset suunnittelu- ja testausvastuun jakautumisesta tiimissä vaihtelivat, mutta tietyllä tavalla painottuivat kuitenkin suuntaa, jossa suurimman osan vastuusta kantaa tiimitestaaja. Hyväksymistestaus katsottiin olevan täysin testaajien vastuulla pois lukien suunnittelijoiden antama suunnittelun aikainen tuki. Jakautumisen kiteytti kommentti, jonka mukaan suunnittelijat varmistavat, että tuotos jotenkin toimii, mutta lopullisen vastuun kantaa testaus.

Vastaus testausnäkökulman huomioimiseen suunniteltaessa uusien ominaisuuksien toteutusta oli hyvin yksimielinen. Kaikkien vastaajien mielestä testausta ei käytännössä huomioida lainkaan suunnitteluvaiheessa. Ainoana poikkeuksena oli mallipohjainen suunnittelu, jonka puitteissa tilanne ei ollut aivan niin heikko.

Testaukseen ja kehitykseen käytettävien resurssien välinen tasapaino koettiin hyväksi, hyvin kehityspainoiseksi tai siltä väliltä. Tasapainon korjaamiseksi käytettävillä resursseilla nousi suunnittelijoiden vahvempi osallistuminen testaustyöhön, jolloin alituisen uuden tekemisen sijaan keskityttäisiin vahvemmin varmentamaan enemmän toimitusten laatua.

Kehitystyötä testauksen katsottiin tukevan nykyisellään hyvin. Tosin kattavampaa regressiotestausta kaivattiin luotettavuuden parantamiseksi. Kehityskohteeksi nostettiin osaamistason parantaminen entisestään, jotta tiimien tekemien automaattitestien määrää ja laatua saataisiin nostettua. Testiautomaatioympäristön kehitys koettiin myös kehityskohteeksi, jotta testien tekeminen helpottuisi. Kehitystarpeita löydettiin myös uusien ominaisuuksien suunnitteluvaiheesta, jonne toivottiin muun muassa testauksen vahvempaa läsnäoloa.

9.2.3 Käännös- ja testausprosessista saatava palaute -osio

Kolmantena haastattelun teemana oli käännös- ja testausprosessista saatava palaute. Kysymyksillä haluttiin selvittää, pystyykö prosessi nykyisellään täyttämään palautteen osalta sille määriteltyä tärkeintä tehtävää, eli kertoa tarvittaville henkilöille ohjelmiston sen hetkisen tilan. Lisäksi haluttiin selvittää, kuinka näkyvänä prosessin tuottama palaute koettiin, sekä palautteen kyky tukea haastateltavia heidän jokapäiväisessä työssänsä.

Olipa prosessin tuottama palaute sitten miten hyvää tahansa, on se turhaa, mikäli siihen ei reagoida asianmukaisella tavalla. Haastattelussa haluttiin selvittää myös prosessin tuottamaa palautetta tällä hetkellä, sen laatua, sekä organisaation alttiutta reagoida palautteeseen.

Lopuksi haluttiin selvittää haastateltavan näkemys siitä, mikä olisi palautteeseen liittyen tärkeintä hänen oman työnsä kannalta, sekä mahdolliset kehitysideat palautteeseen liittyen. Näillä kysymyksillä pyrittiin pohjustamaan tärkeimpiä palautteeseen, sekä sen kehittämiseen laadittavia toimenpide-ehdotuksia.

Tulokset

Nykyisen käännös- ja testausprosessin tuottaman palautteen ajateltiin antavan kohtuullisen realistinen kuva ohjelmiston nykytilasta sen suhteen, onko käännös mennyt läpi vai ei. Käännöstä koskevien testitulosten näkyvyys puolestaan sai hylsyn, koska suurimmalla osalla henkilöstöstä ei

edes ole tietoa, mistä tulokset löytyvät. Lisäksi palautteen epäluotettavuutta lisää vähäinen testiautomaation määrä, jolloin palaute ei luonnollisestikaan voi olla yksityiskohtaista ja sen puitteissa saadaan tietoon ainoastaan murto-osa ohjelmistossa olevista vioista.

Prosessin tuottaman palautteen näkyvyyttä ja sen kykyä tukea haastateltavia heidän omissa töissään arvioitiin käännösprosessin osalta hyväksi. Manuaalitestauksen tuottama palaute koettiin hyväksi ja automaattisen huonoksi.

Palautteen asianmukaisuudesta ei juurikaan tullut kommenttia suuntaan, eikä toiseen, mutta palautteeseen reagointi organisaatiossa vaihtelee merkittävästi. Parhaiten reagoidaan manuaalitestauksen tuottamaan palautteeseen. Testiautomaation tuottamaan palautteeseen reagointia pidettiin puolestaan hitaanpuoleisena tai joskus siihen ei reagoida ollenkaan. Tietyillä tahoilla tätä pidettiin jo huolestuttavana seikkana, ja seikka onkin hyvä saada korjattua tulevaisuudessa asian tärkeyden edellyttämälle tasolle.

Oman työn kannalta tärkeimmäksi seikaksi palautteeseen liittyen koettiin nopeus liittyen tehtyihin muutoksiin sekä palautteen luotettavuus. Prosessin tuottaman palautteen tulee voida toimia vakuutuksena järjestelmän toiminnasta. Merkittävimmiksi kehityskohteiksi listattiin palautteen helpompi saatavuus sekä ymmärrettävyys. Vikatiloissa tarkempi analyysi, miksi asiat eivät toimi.

9.2.4 Muut aihepiiriin liittyvät asiat -osio

Tämän osion puitteissa kysyttiin, kuinka testaukseen liittyvä dokumentointi voisi paremmin tukea haastateltavien työtä. Perusteena tälle kysymykselle oli kartoittaa tarvittavan, testaukseen liittyvän dokumentoinnin määrää ja tasoa, koska väärillä toimenpiteillä dokumentoinnista voidaan helposti tehdä kohtuuttoman raskas ja hankalasti ylläpidettävä kokonaisuus.

Viimeisenä kysymyksenä listalla oli mahdolliset muut kehitysideat koko prosessin parantamiseksi. Tämä kysymys otettiin mukaan siitä syystä, että sen avulla voitiin saada arvokkaita ajatuksia koko prosessista useilta eri organisaation tasoilta.

Tulokset

Dokumentointiin liittyen tärkeimpinä seikkoina listattiin testaussuunnitelmien helppo löydettävyys sekä kunnollinen raportti. Lisäksi kaivattiin listausta testitapauksista, jotta tiettyjen muutosten perusteella voitaisiin testata nopeasti ne asiat, joihin muutos on voinut vaikuttaa. Testatun ohjelmaversion näyttäminen myös manuaalisen testauksen raportoinnissa koettiin tärkeäksi. Juridiselta kannalta katsottuna koko ketju vaatimusmäärittelystä testituloksiin onkin hyvä jatkoa ajatellen dokumentoituna asianmukaisesti.

Muita koko prosessin toimintaa parantavia seikkoja puolestaan nousi esiin seuraavasti. Uusien ominaisuuksien suunnitteluun käytetty aika koettiin riittämättömäksi toimintojen perusteellisen ymmärryksen kannalta. Samoin myös testiautomaation kehittämisen aloittaminen samanaikaisesti suunnittelun kanssa koettiin parannustarpeeksi. Tietynlaista selkeyttämistä eri roolien vastuualueisiin kaivattiin tuotehallinnasta lähtien ja lopuksi usein aikaisemminkin esille noussut selkeä hyväksymiskriteerien sekä käyttötapausten määrittely heti suunnittelun alkumetreiltä siten, että kaikki toiminnan ymmärtämisen kannalta tärkeät henkilöt ovat paikalla.

9.3 Havainnointi

Tärkeimmät tutkimuksen aikaiset havainnot liittyivät hyvin pitkälti myös haastatteluissa esille nousseihin asioihin. Merkittävä, tietynlaista epätietoisuutta aiheuttavana seikkana voidaan pitää tulevan ohjelmistosukupolven testausta määrittävän testausstrategian puuttumista. Strategian puuttuminen suo tiimeille hyvin paljon vastuuta esimerkiksi automatisoidun testauksen piiriin otettavien toiminnallisuuksien suunnittelusta, mutta kokonaisuuden kannalta katsottuna tämä menettely ei välttämättä tuo parasta mahdollista lopputulosta.

Katselmointeihin liittyvät käytännöt vaihtelivat tiimeittäin myös merkittävästi. Esimerkiksi testiautomaation suhteen katselmointikäytäntöjen yhtenäistäminen voisi tuoda tietynlaista yhdenmukaisuutta myös kirjoitettuihin testeihin, eteenkin mikäli katselmoinnit suoritettaisiin sekä tiimin, että muiden testiautomaatiota tekevien henkilöiden kesken. Malli auttaisi myös jakamaan osaamista automaation kirjoittamisen suhteen tiimien välillä.

Testauksen suunnittelun vähäinen dokumentoinnin määrä oli myös eräs esille noussut seikka. Ilman selkeää suunnitelmaa siitä, kuinka toimintoa testataan milläkin testauksen eri tasolla, jää

tiimille merkittävä epätietoisuus siitä, mikä osa-alue on kenenkin vastuulla. Epätietoisuuden seurauksena osa-alueita voi jäädä testaamatta ja toisaalta voidaan tehdä osittain myös päällekkäistä testausta. Suunnitelmallisuuden puute on myös osasyynä sille, että asioita ei testata niille luonnollisissa paikoissa, vaan esimerkiksi hyväksyntätasolla yritetään testata jopa asioita, jotka todellisuudessa kuuluvat yksikkötestaukseen.

10 Tämänhetkinen tilanne sekä siihen liittyvät kehitystarpeet

Yrityksen ohjelmistokehitys perustuu nykyisellään jatkuvan integroinnin prosessimalliin, joten radikaaliin perusprosessin muuttamiseen ei ole tarvetta. Sen sijaan suurimmat kehitystarpeet liittyvät toimintamallien muutoksiin, sekä prosessin hiomiseen tietyiltä osin.

10.1 Testausstrategia

Nykyisellään yrityksessä ei ole dokumentoitua, uuden ohjelmistosukupolven testausta varten laadittua testausstrategiaa. Koska kunnollisen strategian laadinta edellyttää useiden henkilöiden osallistumista laadintaprosessiin, päädyttiin tämän työn puitteissa ainoastaan tuottamaan sekä analysoimaan taustatietoa strategian laatimisen tueksi. Lisäksi laadittiin malli strategiakartasta sekä alustava listaus mahdollisista toimenpiteistä. Strategian laadinnan tueksi tehtiin SWOT -analyysi, joka perustui haastatteluista saatuun materiaaliin, sekä omiin havaintoihin. Lisäksi työn puitteissa laadittiin tasapainotettu mittaristo, jota käytettiin luotaessa alustava strategiakartta, jonka jatkokehitys tulee olemaan osa yrityksen testausosaston strategiatyötä.

Liitteessä 10 on esitettynä laaditut analyysit sekä strategiakartta tulevalle kolmelle vuodelle.

10.2 Prosessi

Työssä haluttiin selvittää testauksen sekä ohjelmistokehityksen välistä suhdetta tällä hetkellä sekä kuinka prosessia voitaisiin kehittää, jotta toimitusprosessi saataisiin paremmin toimivaksi. Haastatteluiden, sekä havainnoidun aineiston pohjalta esille nousi muutamia kehitysehdotuksia toiminnan tehostamiseksi.

Merkittäväksi ohjelmiston laatua heikentäväksi tekijäksi koettiin vähäinen testiautomaation määrä. Vähäisen määrän voidaan katsoa suurilta osin johtuvan muun muassa seuraavista seikoista:

- Hyväksyntätason testiautomaatiota on kirjoitettu suhteellisen vähän aikaa verrattuna ohjelmiston kehityshistoriaan

- Testiautomaation suunnitelmallisuus
- testauksen sekä kehityksen välinen resurssi tasapaino
- suhteellisen vähäinen kokemus testiautomaation kirjoittamisesta
- Manuaalitestaukseen käytetty aika
- Testiautomaatio ei ole osana hyväksyntäkriteereitä

Kattavan testiautomaation puutetta voidaan pitää kumulatiivisena seurauksena edellä listatuille asioille. Seuraavan sukupolven ohjelmistoa on kehitetty jo useita vuosia ilman merkittävää panostusta testiautomaatioon. Nykyisellään kehitys on jo huomattavan matkan päässä suhteutettuna toiminnallisuutta testaavaan testiautomaation määrään. Tämän asian korjaamiseksi ei nykyresursseilla voida paljoakaan, koska valtaosa testauksen resursseista tarvitaan tukemaan uusien ominaisuuksien kehitystyötä. Suositeltava toimintamalli on pyrkiä määrittelemään osa-alueet, jotka eivät vielä ole testiautomaation piirissä ja luoda näille alueille yleisimmät skenaariot kattavat hyväksyntätestit (ks. s.44-45).

Useissa yhteyksissä esiin nousi testauksen edustuksen puuttuminen ominaisuuksien suunnitteluvaiheessa. Tähän kannattaakin kiinnittää nykyistä enemmän huomiota ja muuttaa prosessia suuntaan, jossa testaus otetaan mukaan kehitykseen jo suunnittelun alkumetreillä (ks. s.7-8). Suunnitelmallisuuden puuttuessa asioita ei tehdä organisoidusti ja kaikki ovat vähän epätietoisia, mitä missäkin vaiheessa pitäisi testata ja mikä on sopiva testauksen taso. Esimerkiksi merkittävä hyväksyntätestauksen kehitystä hidastava seikka on, että sen puitteissa yritetään testata esimerkiksi yksikkötestauksen puitteissa testattavia asioita.

Automaattitestien tekemiseen suunnatut vähäiset resurssit ovat myös merkittävä syy vähäiseen testiautomaation määrään. Nykytilassa suhde tiimitestaajien, sekä ohjelmistokehittäjien välillä on noin 1/6. Tätä voidaan pitää suhteellisen pienenä lukuna, kun huomioidaan ettei tiimitestaajilla ole mahdollisuutta antaa täyttä osaa työpanoksesta testiautomaation kehittämiseen. Uusien henkilöiden tuominen prosessiin ei välttämättä paranna tilannetta, mikäli tekeminen ei ole kunnolla organisoitua. Eräs suositeltava vaihtoehto tässä tilanteessa resurssien tasaamiseksi on kehittäjien nykyistä painokkaampi osallistuminen automaattitestien kirjoittamiseen (ks. s.10).

Yrityksessä on vastikään siirrytty käyttämään edeltävään ohjelmistosukupolveen verrattuna täysin uusia, hyväksyntätason testien tekemiseen käytettäviä työkaluja. Osaltaan tämä, sekä manuaalitestauspainotteinen historia ovat merkittävimpiä syitä vähäiseen kokemukseen testiautomaation kirjoittamiselle. Lisäksi aikaisempina vuosina testauspuolen rekrytointeihin ei ole vaadittu merkittävää ohjelmoinnin osaamista, mikä osaltaan nykyisellään heijastuu tehokkuuteen, jolla automaattitestejä voidaan kirjoittaa.

Koska testattavaa järjestelmää voidaan pitää erittäin monimutkaisena kokonaisuutena, jossa järjestelmän syvällinen tuntemus on edellytyksenä hyväksyntätason testien kirjoittamiselle, eivät yleismaalliset koulutukset välttämättä riitä parantamaan osaamista juuri kyseistä järjestelmää silmällä pitäen. Nykyisessä organisaatiomallissa jokaisessa tiimissä on vähintään yksi testaaja, mutta testaajien välinen yhteistyö tiimien välillä on erittäin vähäistä. Jotta tasoerot tiimien tuottamien automaattitestien välillä voidaan minimoida, tulee tiimien välistä yhteistyötä lisätä merkittävästi. Eräs vaihtoehto yhteistyön lisäämiseksi voisi olla esimerkiksi virtuaalinen testaajien muodostama tiimi. Tällöin testaajat olisivat normaalisti osa omaa kehitystiimiään, jolloin tiimin työlista pitkälti määrittelisi tehtävät työt. Virtuaalitiimin rooli korostuisi varsinaisessa tekemisessä, jolloin testaajat saataisiin paremmin tukemaan toistensa työtä ja päivittäiset ongelmat automaattitestien kirjoittamisen suhteen voitaisiin tuoda helpommin esille esimerkiksi virtuaalitiimin omissa päivittäisissä tiedonvaihtotilaisuuksissa (scrum daily). Malli toisi suurella todennäköisyydellä yhtenäisemmän linjan testien tekemiseen sekä jakaisi ammattitaitoa tiimitestaajien välillä.

Historian painolastin saattelemana testausta tehdään edelleen paljon manuaalisesti. Eräs syy tähän voi olla edelleen suhteellisen korkea kynnys lähteä automatisoimaan toimintojen testausta. Nykyisellään tehtävä manuaalitestaus on lähes poikkeuksetta automatisoitavissa, koska se on luonteeltaan hyvin pitkälti samojen asioiden toistuvaa läpikäyntiä (ks. s.26). Eräs menetelmä kynnyksen madaltamiseksi voisi olla edellä mainittu virtuaalitiimi, jolloin tehostetun yhteistyön piiriin otetut, manuaalitestaukseen orientoituneet tiimitestaajat voitaisiin asteittain perehdyttää testiautomaation kirjoittamiseen.

Mikäli prosessi ei edellytä tiettyjä toimia, saattavat nämä kyseiset toimet jäädä helposti tekemättä. Esimerkkinä tästä voidaan pitää testiautomaation puuttumista siitä listauksesta, jonka perusteella toteutettavat ominaisuudet määritellään valmiiksi. Mikäli työn määrittely valmiiksi edellyttäisi vähintään yhtä automaattitestiä jokaista määriteltyä hyväksymiskriteeriä kohden, olisi se tässä vaiheessa suhteellisen järeä keino, mutta sen sijaan kannattaa kuitenkin miettiä automaation mukaan ottamista aluksi joltain osin ja rutinoitumisen myötä sitten lisätä tuota osaa asteittain (ks. s.7-8).

Erääksi ongelmaksi nykyisellään koettiin luottamuksen puute liittyen siihen, ettei ohjelmisto-regressioita löydetä tarpeeksi ajoissa. Tämä johtuu pääsääntöisesti järjestelmän kompleksisuudesta sekä kattavan testiautomaation puutteesta. Luottamusta on hankala rakentaa nopealla aikavälillä, mutta noudattamalla edellä listattuja ehdotuksia, voitaisiin automaattitestien kehitystä tehostaa, ja sitä kautta parantaa niiden tuomaa kattavuutta ohjelmistoregressioita vastaan. Lisäksi voitaisiin muuttaa prosessia löytyneiden ohjelmistovikojen osalta siten, että jokaista löytynyttä ohjelmistovikaa vastaan tehdään vähintään yksi vian löytävä automaattitesti (ks. s.45).

Koska testiautomaation eräs tärkeimmistä tehtävistä on tukea suunnittelijoita ohjelmistokehitystyössä mahdollisimman hyvin, ei se nykyisellään pysty täyttämään tätä tehtävää, koska suunnittelijoilla ei ole mahdollisuutta ajaa hyväksyntätason testejä omilla työasemillaan tehtyjen koodimuutosten jälkeen. Tämä puute tuo merkittävän viiveen kehitystyöhön, koska nykyisellään suunnittelijat saavat testitulokset muutoksista vasta seuraavana päivänä. Tämän epäkohdan poistamiseksi tuleekin tehdä toimenpiteitä, jotta testejä voidaan jatkossa ajaa omilla työasemilla. (ks. s.24).

Käännös sekä testausprosessien tuottama palaute ei myöskään nykyisellään ole sillä tasolla, jolla sen kuuluisi olla. Suuri osa kehittäjistä ei edes tiedä mistä testiautomaation tuottamat raportit löytyvät. Palautteen paremman näkyvyyden saavuttamiseksi tarvittavat toimenpiteet eivät ole mittavia, koska näytettävät raportit sekä statistiikka ovat jo hyvin pitkälti olemassa. Jatkokehitysmielessä testitulosten sekä raporttien esittäminen esimerkiksi tiimihuoneen info tv:llä voisi olla hyvä ajatus (ks. s.11).

Palautteeseen reagointi koettiin testiautomaation osalta nykyään hitaaksi ja joskus jopa täysin olemattomaksi. Käytännössä tämä näkyy siten, että testit ovat pitkään punaisina, mistä seuraa tietynlainen hiljainen hyväksyntä sille, että testit eivät mene läpi. Tämä puolestaan nostaa entisestään kynnystä pitää hyväksyntätestit vihreänä. Tähän epäkohtaan on mahdollista puuttua esimerkiksi siten, että hyväksyntätason testien pitäminen vihreänä nostetaan samalle prioriteetille muun tekemisen kanssa. Eräs usein huomiotta jäävä seikka läpimenemättömien testien kohdalla on se, että testattaviin toimintoihin voi ilmetä uusia vikoja tuona aikana, kun testi ei mene läpi, ja näihin vikoihin ei tällöin voida reagoida. Lisäksi suunnittelijoiden kannalta katsottuna on helpompi korjata ohjelmistoviat mahdollisimman pian niiden ilmennyttyä. (ks. Liite 7)

11 Johtopäätökset ja pohdinta

Monitasoisen ohjelmistokokonaisuuden toimitusprosessin toimittaminen ja julkaisu vaatii uskomattoman paljon erinäisiä, työvaiheita sekä rautaista ammattitaitoa näiden vaiheiden tekijöiltä. Hyvin harvoin laadukkaan tekemisen esteenä on tekijöiden ammattitaito, vaan todennäköisempänä syynä voidaan pitää puutteellisesti koordinoitua toimintaa.

Näennäinen kiire vaikuttaisi olevan eräs merkittävimmistä, laatuun heikentävästi vaikuttavista tekijöistä. Kun tekemisestä itsessään nipistetään keinotekoisen kiireen antamin ”valtuuksin”, joudutaan tuo voitettu aika maksamaan myöhemmin moninkertaisena takaisin erinäisten tulipalojen sammuttelun muodossa.

Perinteinen ajattelu testauksesta ravintoketjun pohjalla on vanhanaikainen ja uudistusta kaipaava ajattelumalli. Omien havaintojen pohjalta voidaan todeta, että suunta on oikea, vaikka taannoin eräs työhaastattelussa testaajanpaikkaa hakeva kokelas tuumasikin alentuvansa ohjelmistosuunnittelijasta testaajaksi, mikäli hänet paikkaan olisi valittu. Hänen onnekseen alentumista testaajaksi ei päässyt tapahtumaan. Muurin takana olevan erillisen osaston sijaan testaus voitaisiin ajatella ohjelmistosuunnittelun turvaverkkona, jonka turvin suunnittelija voi rohkeasti tehdä oman työnsä ilman pelkoa putoamisen aiheuttamasta tuskasta.

Tämän työn puitteissa keskityttiin tutkimaan ainoastaan testauksen integroitumista ohjelmistokehitysprosessiin ja tavoitteena oli luoda selkeä kuva nykytilasta. Mahdollisimman totuuden mukainen kokonaiskuva nykytilasta haluttiin muodostaa lähestymällä ilmiötä eri tulokulmista, jolloin voitiin saavuttaa laajempi tietopohja prosessin kehittämiseksi. Kaikkinensa työ oli äärimmäisen mielenkiintoinen ja silmät avaava kokemus. Aiheeseen liittyvä teoria toi vahvistusta haastatteluiden pohjalta saatuun tietoon sekä omiin näkemyksiin asioista. Ennen tähän työhön ryhtymistä esimerkiksi termi ”ketterät ohjelmistokehitysmenetelmät” oli ennalta hyvinkin tuttu, mutta nykyisellään termi on omalla kohdallani saanut huomattavasti syvemmän merkityksen. Tästä termistä on puhuttu paljon ja usein siihen viitataan esiteltäessä omaa kehitysprosessia, mutta oma näkemykseni on, että kehitys voisi olla huomattavasti nykyistä ketterämpääkin.

Haastattelut tiedonkeruumenetelmänä soveltuivat tutkimusosioon erittäin hyvin. Kysymysten laadinta tosin oli hieman haastava, jotta organisaation eri asemissa toimivat henkilöt pystyivät niihin järkevissä puittein vastaamaan siten, että vastaukset tuottivat lisäarvoa tutkimukselle itsessään. Kysymyksillä pyrittiin tässä yhteydessä välttämään kyllä ja ei vastauksia.

Vaikka testaus saataisiinkin täysin integroitumaan kehitystyöhön tiimin sisäisessä dynamiikassa, kokonaisuuden kannalta merkittävä seikka on testauksen yhteistyö tiimien välillä. Merkittävä huomioitava seikka koko ohjelmiston laadun kannalta katsottuna on testauksen tasoerot tiimien välillä. Näitä tasoeroja tulee pyrkiä minimoimaan esimerkiksi lisäämällä tiimitestaajien välistä yhteistyötä tai vaihtoehtoisesti järjestämällä säännöllisin väliajoin koulutusluontoisia tapahtumia, joiden puitteissa osaamista sekä hyväksi havaittuja käytäntöjä voitaisiin jakaa tiimien kesken.

Merkittävimmät henkilökohtaiset tavoitteet tämän työn tuomaan antiin liittyen sijoittuvat vahvasti testausnäkökulman huomioon ottamiseen jo suunnitteluvaiheessa sekä siihen, että julkaistavan tuotteen laatu on koko kehitystiimin vastuulla. Jokaisen prosessiin osallistuvan henkilön olisi myös hyvä ymmärtää oman tekemisen vaikutus kokonaisuuteen ja saada vahvistusta sille, että omalla työpanoksella todellakin on merkitystä.

Tietyllä tavalla strategia liitetään usein johonkin kaukaiseen termiin, jolla ei käytännön työn kanssa ole mitään tekemistä. Oma näkemykseni strategiasta, tai lähinnä sen puuttumisesta vastaa hyvin paljon nykyisen toiminnan tilaa. Strategia ei voi olla lista toteuttamiskelvottomia pinnallisia itsestäänselvyyksiä, kuten esimerkiksi että *kaikki toimii*, mikä luonnollisesti olisi hyvä asia. Henkilökohtainen käsitys sen sijaan on, että strategian tulee sisältää realismia hieman optimistisemmat tavoitteet suhteutettuina resursseihin, sekä laaditut toimenpiteet näiden tavoitteiden saavuttamiseksi. Henkilöstön täytyy tiedostaa nämä tavoitteet ja ymmärtää, että omalla, sekä muiden työpanoksella näiden tavoitteiden saavuttaminen on mahdollista, kun noudatetaan määriteltyä toimenpidesuunnitelmaa. ”Kaikki toimii” -ratkaisu ei välttämättä aiheuta henkilöstössä ponnisteluja, koska lähtökohtaisesti tiedostetaan tavoitteen mahdottomuus.

Lähteet

Kirja:

1. Jez Humble & David Farley (2011). Continuous Delivery. Addison-Wesley.

2. Haikala, I., & Mikkonen, T. (2011). Ohjelmistotuotannon käytännöt (12. uud. p. ed.). Helsinki: Talentum.

3. Haikala I & Merijärvi J. (2004) Ohjelmistotuotanto (10. uudistettu painos), Talentum media Oy, 2004

4. Tapio, Tero, 2010, Riskienhallinta perinteisessä ja ketterässä ohjelmistokehityksessä. Pro gradu –tutkielma, Tampereen yliopisto. Luettu 4.4.2020. https://tam-pub.uta.fi/bitstream/handle/10024/81663/gradu04362.pdf?sequence=1

5. Sommerville, I. (2001). Software engineering (6th ed ed.). Harlow, England: Pearson Education.

6. McConnell, S., Toikkanen, T., & Arola, J. (2002). Ohjelmistotuotannon hallinta. Helsinki: Edita: IT Press.

7. Wiegers, K. E. (2003). Software requirements (2nd ed ed.). Redmond (WA): Microsoft Press.

8. Tuomi, J. & Sarajärvi, A. (2009), Laadullinen tutkimus ja sisältöanalyysi, Kustannus-osakeyhtiö Tammi, Helsinki.

9. Alasuutari, P. (2011) Laadullinen tutkimus 2.0, Vastapaino, Tampere.

10. Kananen, J. (2013) Case-tutkimus opinnäytetyönä. Jyväskylän ammattikorkeakoulun julkaisuja 143. Jyväskylä: Suomen Yliopistopaino Oy.

11. Kesti M: Strateginen henkilöstötuottavuuden johtaminen

12. Vuorinen T. (2013), Strategiakirja: 20 työkalua (e-kirja), Talentum, Helsinki

Artikkelit:

13. Lahtinen, N. 2015. Suomen Digimarkkinointi. Mitä on A/B-testaus. Luettu 09.04.2020. https://www.digimarkkinointi.fi/blogi/mita-ab-testaus

Internet-sivut:

10. Wikiyhteisö (2017. 22. huhtikuuta) Selitys termille Validointi. Haettu 1.4.2020, sivustolta Wikipedia: https://fi.wikipedia.org/wiki/Validointi

11. Wikiyhteisö (2020. 11. huhtikuuta) Software release life cycle. Haettu 13.4.2020, sivustolta Wikipedia: https://en.wikipedia.org/wiki/Software_release_life_cycle

12. Wikiyhteisö (2019. 1. marraskuuta) Selitys termille Paradigma. Haettu 1.4.2020, sivustolta Wikipedia: https://fi.wikipedia.org/wiki/Paradigma

13. Wikiyhteisö (2019. 22. syyskuuta) Selitys termille Heurestiikka. Haettu 1.4.2020, sivustolta Wikipedia: https://fi.wikipedia.org/wiki/Heuristiikka

14. Tutorialspoint (2020) Agile Testing - Quadrants. Haettu 4.4.2020, sivustolta Tutorialspoint: https://www.tutorialspoint.com/agile_testing/agile_testing_quadrants.htm

15. Anastasija Reshkova (2018. 8. elokuuta) The difference and relationship between Use case and User story. Haettu 4.4.2020, sivustolta: https://medium.com/@a.reskova/the-difference-and-relationship-between-use-case-and-user-story-25e24df777a3

16. Guru99 (2020) What is Gherkin? Write Gherkin Test in Cucumber. Haettu 4.4.2020, sivustolta: https://www.guru99.com/gherkin-test-cucumber.html#3

17. Kirsten Aebersold (2020) Test automation frameworks. Haettu 4.4.2020, sivustolta Smartbear: https://smartbear.com/learn/automated-testing/test-automation-frameworks/

18. Guru99 (2020) What is TEST HARNESS? Tools & Examples. Haettu 4.4.2020, sivustolta: https://www.guru99.com/what-is-test-harness-comparison.html

19. Wikiyhteisö (2019. 22. toukokuuta) Artikkeli Scrum. Haettu 4.4.2020, sivustolta Wikipedia: https://fi.wikipedia.org/wiki/Scrum

20. Wikiyhteisö (2020. 3. huhtikuuta) Artikkeli Dependency injection. Haettu 6.4.2020, sivustolta Wikipedia: https://en.wikipedia.org/wiki/Dependency_injection

21. Wikiyhteisö (2020. 12. tammikuuta) Artikkeli Test stub. Haettu 6.4.2020, sivustolta Wikipedia: https://en.wikipedia.org/wiki/Test_stub

22. Wikiyhteisö (2020. 13. helmikuuta) Artikkeli Mock object. Haettu 6.4.2020, sivustolta Wikipedia: https://en.wikipedia.org/wiki/Mock_object

23. Cucumber (2019) Behavior-Driven Development. Haettu 6.4.2020, sivustolta: https://cucumber.io/docs/bdd/

24. Microsoft docs (2020) Design the infrastructure persistence layer. Haettu 6.4.2020, sivustolta: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design

25. Wikiyhteisö (2020. 2. huhtikuuta) Artikkeli Relational database. Haettu 8.4.2020, sivustolta Wikipedia: https://en.wikipedia.org/wiki/Relational_database

26. High Impact Project Management (2020) What Is an Agile Developer? How Is the Role Different? Haettu 8.4.2020, sivustolta: https://managedagile.com/what-does-it-mean-to-be-an-agile-developer/

27. Ian Spence and Kurt Bittner (2005. 15. maaliskuuta) What is iterative development? -- Part 1: The developer perspective. Haettu 8.4.2020, sivustolta IBM Developer: https://www.ibm.com/developerworks/rational/library/mar05/bittner/index.html

28. Prescio (2020) V-model and Agile Methology. Haettu 19.4.2020, sivustolta: http://prescio.com/services/software-development/v-model-and-agile-methodology/

29. Joonas Koski (2020) Ketterät menetelmät, agile, LEAN ja scrum. Haettu 8.5.2020, sivustolta itewiki: https://www.itewiki.fi/opas/ketterat-menetelmat-agile-lean-ja-scrum/

Deployment pipeline:	Automatisoitu julkaisuputki on erinäinen joukko laadunvarmennusta sekä testausta sisältäviä vaiheita, joiden läpi muutokset ajetaan automaattisesti ennen päätymistä tuotantoon.
Continuous integration:	Malli, jossa ohjelmistokehittäjät lisäävät tuottamansa ohjelmakoodin säännöllisesti versionhallintaan ja käyttävät automatisoitua käännösprosessia muutosten automatisoituun verifiointiin.
Continuous delivery:	Jatkuvan julkaisun malli, jossa ohjelmistomuutokset käyvät läpi kattavan, automatisoidun regressiotestauksen, ja jossa ne toimitetaan automaattisesti. Päätöksen tuotantoonviennistä tekee ihminen.
Continuous deployment:	Kuten jatkuvan julkaisun malli, mutta toimitus tuotantoon täysin automaattinen.
Testing environment:	Ympäristö, jossa ohjelmaa voidaan testata, mutta joka ei kuitenkaan välttämättä vastaa tuotantoympäristöä. Esimerkiksi ohjelman suoritus virtuaalikoneella ilman tuotantoympäristön hardwarea.
Staging environment:	Tuotantoympäristöä vastaava testausympäristö, jossa ohjelmisto testataan ennen sen julkaisua tuotantoon. Käytännössä ympäristöjen HW, SW sekä konfiguraatiot tulee olla vastaavat.
Check-in / commit:	Ohjelmakoodin lisääminen versionhallintaan, minkä jälkeen ne ovat osa ohjelman koodikantaa
Check-out:	Päivittää tiedostot versionhallinnasta paikalliselle työasemalle ja jättää versionhallintaan tiedon, että joku editoi valittuja tiedostoja.
Deployment:	Prosessi, jonka puitteissa ohjelmaversio toimitetaan haluttuun ympäristöön.
Commit stage:	Automaattisen julkaisuputken ensimmäinen vaihe, jonka puitteissa ohjelmakoodi käännetään, sille suoritetaan yksikkötestit sekä ajetaan erinäiset koodianalyysit.
Commit test suite:	Ensimmäisessä käännösputken vaiheessa määriteltäväksi ajetut nopeat testit
Release candidate:	Julkaisu kandidaatti on ohjelmistoversio, joka on toimiva, mutta ei ole vielä valmis julkaistavaksi tuotantoon
CI-server:	Jatkuvan integroinnin ytimenä toimiva serverikone, joka monitoroi versionhallintaan tehtävät muutokset, kääntää koodit, ajaa

	testit sekä lisää ohjelmistokomponentit niiden hallintajärjestelmään. Lisäksi serveri voi tuottaa tiimeille statistiikkaa käännöksestä.
Artifact repository:	Hallintajärjestelmä, joka on suunniteltu taltioimaan kaikentyyppisiä tiedostoja. Hallintajärjestelmän avulla sinne taltioidut tiedostot voidaan esimerkiksi versioida, sekä erinäisten käyttöoikeuksien avulla pääsyä tiedostoihin voidaan kontrolloida.
Branch	Versionhallintaan tehtävä kehityshaara, jossa ominaisuutta voidaan kehittää ilman vaikutusta muiden työhön tai koko ohjelman toimintaan
Test driven development	Testivetoinen ohjelmistokehitys, jossa uutta toiminnallisuutta testaavat testit tehdään ennen varsinaista uuden toiminnon koodaamista.
Behavior driven development	BDD on ohjelmistotiimien työskentelymalli, joka perustuu vahvaan vuorovaikutukseen asiakasrajapinnan sekä kehittäjien välillä, jolloin voidaan helpommin rakentaa yhteinen käsitys ratkaistavista ongelmista. [23]
Pretested commit	Esitestattu versionhallintaan vienti, jolloin muutoksille ajetaan commit -vaiheen testit ennen niiden tallentamista versionhallintaan.
Refactoring:	Koodin uudelleenjärjestelyä, jonka puitteissa koodin rakennetta tai arkkitehtuuria muutetaan esimerkiksi yksinkertaisemmaksi, tai muuten toimintaa paremmin tukevaksi.
Code linter:	On ohjelmistokehitystyökalu, joka analysoi lähdekoodia ohjelmointivirheiden, bugien, tyyliseikkojen ja epämääräisten rakenteiden varalta.
Quality gate:	Laatuportti, joka voidaan määritellä käännösprosessiin esimerkiksi siten, että mikäli yksikkötestien koodikattavuus putoaa alle 60%, niin käännös menee rikki.
Happy case:	Testauksen termi tietylle ohjelman ominaisuudelle tyypillinen yleisimmin suoritettu koodin suorituspolku.
Testing quadrant:	
Use case:	Käyttötapaus on kuvaus tietystä sarjasta järjestelmän ja käyttäjän välisiä vuorovaikutus tapahtumia. [15]
Canary releasing:	Ohjelmiston lopputestauksen malli, jonka puitteissa eri käyttäjille julkaistaan ohjelmistoa eri ominaisuuksilla varustettuna, ja

	statistiikan perusteella päätetään mitä seuraavaan versioon otetaan mukaan.
Test framework:	Kuvaa säännöstöä, sekä työkaluja, joiden puitteissa ohjelmiston testiautomaatiota tehdään. [17]
Test harness:	Ohjelmistotestauksessa käytettävä testipeti sisältäen pieniä testauksessa käytettäviä ohjelmia, jotka keskustelevat testattavan ohjelman kanssa [18]
Repository pattern:	On luokka tai komponentti, joka kapseloi sen logiikan, joka tarvitaan esimerkiksi, jotta päästään käsiksi tiettyyn tietolähteeseen, kuten esimerkiksi tietokanta. [24]
Scrum:	Ketterä ohjelmistokehityksen viitekehys. [19]
Dependency injection:	Ohjelmistosuunnittelussa käytettävä tekniikka, jossa testattavan komponentin riippuvuussuhde muihin järjestelmän osiin neutralisoidaan käyttäen hyväksi erillistä, esimerkiksi parametrina testattavalle komponentille annettavaa objektia. [20]
Test stub:	Ohjelmistotestauksessa hyödynnettäviä ohjelmia, jotka simuloivat ohjelmakomponenttien tai moduulien käyttäytymistä. [21]
Mock object:	Todellista objektia simuloiva objekti, jota käytetään eristämään testattava komponentti hankalista riippuvuuksista. [22]
Test double:	Yleinen termi testeissä käytettäville simuloiduille objekteille
Event programming	Tapahtumaohjattu ohjelmointi
Database dump:	Tietokannan varmuuskopio
Relational database:	Tietokanta, jossa data on esitettynä relaatioiksi (taulu) ryhmiteltyinä, äärellisinä listoina. [25]
Application driver layer:	Erillinen kerros esimerkiksi käyttöliittymän sekä ohjelman toiminnallisen kerroksen välissä, jonka avulla ohjelman toimintoja on mahdollista käyttää.
Timeout:	Aika, joka odotellaan esimerkiksi ennen kuin päätetään, että testi ei ole mennyt läpi.
User acceptance testing:	Ohjelmistotestauksen viimeinen vaihe, jossa todelliset käyttäjät testaavat ohjelmistoa.
Extreme Programming:	(lyh. XP) on ketterän ohjelmistokehityksen eräs metodologia.

Säännölliset lisäykset versionhallintaan

Jatkuva integrointi ei voi toimina, mikäli versionhallintaan tehtävien lisäysten välit ovat pitkät. Muutosten lisääminen versionhallintaan riittävän usein tuo paljon etuja ja yleensä muutokset kannattaakin lisätä sinne ainakin pari kertaa päivässä. Tiheään tehtävät lisäykset pitävät muutokset pieninä ja ne rikkovat ohjelman pienemmällä todennäköisyydellä. Mikäli jokin menee pieleen, on palauttaminen viimeisimpään toimivaan versioon nopeaa ilman, suurta määrää hukkaan mennyttä työtä ja väärälle polulle lähtenyttä kehitystä. Muutokset kohdistuvat pienempään määrään tiedostoja, jolloin todennäköisemmin vältytään konflikteilta ja työläältä koodikantojen yhdistämiseltä. Vahingoista palautuminen on tällöin myös huomattavasti kivuttomampaa. Esimerkiksi tällöin ei voida vahingossa poistaa kahden viikon työtä. [1. 59]

Älä vie muutoksia versionhallintaan sisään, kun käännös on rikki!

Jatkuvan integrointiprosessin pahin synti on tehdä muutoksia versionhallintaan käännöksen ollessa rikki. Muiden kehittäjien velvollisuus tässä vaiheessa on odottaa, että koodin rikkoneet henkilöt korjaavat käännöksen ja vasta tämän jälkeen on lupa lisätä muutokset versionhallintaan. Kun tämä asia on ymmärretty ja siihen on sitouduttu, ovat asetelmat parhaat mahdolliset käännöksen rikkoneiden syiden selvittämiseksi, sekä niiden korjaaminen huomattavasti nopeammalla aikataululla. Parhaat asemat koodin korjaamiselle puolestaan mahdollistaa se, että rikkinäisen koodin päälle ei lisätä uutta koodia. Käytännössä poikkeuksetta tämän säännön rikkomisesta seuraavat pidempikestoiset jaksot, jolloin käännös on rikki. [1.s.66]

Älä lähde kotiin, jos olet rikkonut käännöksen

Mikäli muutosten versionhallintaan vienti on rikkonut käännöksen, ei rikkinäistä koodia saisi missään nimessä jättää versionhallintaan ja lähteä kotiin. Mitä kauemmin käännöksen rikkomisesta on kulunut, sitä hatarammaksi muistikuvat sen rikkojalla mahdollisista syistä muuttuvat. Käännöksen jättäminen rikkinäiseksi voi hidastaa huomattavasti kaikkien muiden tekemistä ja mikäli käännöksen rikkonut henkilö ei jostain syystä heti seuraavana aamuna pääse korjaamaan vahinkoa, on todennäköistä, että käännöksen rikkovan henkilön tekemät muutokset palautellaan muiden kehittäjien toimesta viimeisimpään toimivaan versioon ilman suurempia kyselyitä. Vahinkojen ehkäisemiseksi ei versionhallintaan myöskään lisätä mitään työpäivän viimeisinä minuutteina.

Sen sijaan toimenpiteen voi suorittaa tulevan työpäivän ensimmäisenä tehtävänä, tai vastaavasti työpäivän loppupuolella siten, että mahdolliset korjaavat toimenpiteet ehditään tehdä ennen työpäivän päättymistä. [1.s.68]

Kehitysympäristön hallinta

Ohjelmistokehityksen kannalta on erittäin tärkeää, että kehitystyötä voidaan tehdä hyvin hallituissa ympäristöissä. Kehittäjällä tulee aina olla nopea ja yksinkertainen tapa palata tunnettuun toimivaan tilaan ilman yhtään ylimääräistä manuaalista toimenpidettä. Heidän tulee voida kääntää ohjelma, suorittaa testit sekä toimittaa käännetty ohjelma esimerkiksi omalle työasemalleen. Ohjelma on hyvä toimittaa kehittäjän työasemalle käyttäen samaa, jatkuvassa integroinnissa käytettävää automaattista prosessia, jonka avulla ohjelma toimitetaan lopulta myös tuotantoon. [1.s.62]

Lähtötilanne ohjelmistokehitykselle on aina viimeisin tunnetusti toimiva versio, joka on läpäissyt testauksen hyväksytysti. Toisena askeleena voidaan pitää kolmannen osapuolen komponenttien hallintaa, sekä niiden ajantasaisuutta. Kehittäjällä tulee olla käytössään oikeat versiot käytettävistä kolmannen osapuolen komponenteista. Komponenttien käytössä tulee olla huolellinen, koska esimerkiksi aina uusimman komponentin käyttäminen voi johtaa ongelmatilanteisiin. Komponentit päivittyvät harvoin ja hyvä vaihtoehto on jäädyttää tietty versio komponentista ja käyttää sitä ainoastaan versionhallinnan kautta. Menetelmän avulla käytössä on aina toimivaksi testattu komponentti. [1.s.62]

Jotta kehittäjät voivat paremmin osallistua testaustyöhön, tulee heidän voida ajaa automaattisia hyväksyntätestejä paikallisesti omissa kehitysympäristöissään. Kehittäjän tulee helposti voida ajaa esimerkiksi tietyn ohjelmistovian löytävä hyväksyntätesti omalla työasemallaan sen jälkeen, kun muutokset vian korjaamiseksi ovat valmiina. Mikäli testien ajaminen kehitysympäristössä ei onnistu, tulee mahdolliset esteet pyrkiä ratkaisemaan ensisijaisesti. [1.s.125]

Kattava testiautomaatio

Ilman kattavaa testiautomaatiota, ohjelman kääntyminen kertoo ainoastaan, että ohjelman lähdekoodeista on tehty ajettavaksi tarkoitettu ohjelmatiedosto. Vaikka tämä joskus voikin tuntua suurelta saavutukselta, tulee ohjelman toiminnallisuus varmistaa edes jollakin tasolla. Käytännössä tämä tarkoittaa paneutumista yksikkö-, integraatio-, sekä hyväksyntätestaukseen. Hyödyntämällä näitä kolmea eri tasoa, ohjelmiston testaus voidaan toteuttaa joustavasti sekä kattavasti. Lisäksi oikein suunniteltuna ajansäästö voi olla huomattava verrattuna testaamiseen pelkästään toiminnallisella tasolla. [1.s.60]

Nopeat testit muutoksille ennen niiden viemistä versionhallintaan

Automatisoitu kääntäminen sekä sitä seuraavat nopeat testit (commit stage) täytyy pitää mahdollisimman lyhytkestoisena. Mikäli vaiheen päättymistä joudutaan odottamaan liian kauan, on todennäköistä, että muutokset viedään versionhallintaan entistä harvemmin. Jotta kehittäjät jaksaisivat odottaa vaiheen valmistumista ennen siirtymistä seuraavaan tehtävään, ei se saisi kestää yli viittä minuuttia. Toiminnan sujuvuuden kannalta ehdoton yläraja vaiheen kestolle on 10 minuuttia. Vaikka kattavan testiautomaation suorittaminen minimaalisessa ajassa voikin kuulostaa ristiriitaiselta, on se kuitenkin täysin toteutettavissa erilaisin tekniikoin. Jotta vältytään rikkinäisen koodin lisäämiseltä versionhallintaan. tulee tämä vaihe suorittaa ennen minkä tahansa muutoksen lisäystä versionhallintaan. [1.s.60-61, 120-121]

Tekemisen suhteen ohjelmoinnissa pätee usein samat säännöt verrattuna mihin tahansa muuhun tehtävään työhön. Useimmat ihmiset tarkistavat tekemänsä työn laadun ennen tulosten esittelyä, olivat ne sitten missä muodossa tahansa, eikä ohjelmointityön tule poiketa käytännöstä millään tavalla. Ensimmäisen vaiheen testien (commit test suite) ajo ennen koodimuutosten julkaisua antaa tietynlaisen varmuuden siitä, että tehty työ on esittelykelpoinen. Kun kehittäjä on valmis lisäämään muutokset versionhallintaan, synkronoidaan viimeisimmät koodit versionhallinnasta paikalliselle työasemalle ja lopuksi suoritetaan putken ensimmäisen vaiheen testit. Vasta kun nämä vaiheet on suoritettu onnistuneesti, voidaan muutokset lisätä versionhallintaan. [1.s.66-67]

Joskus voi tuntua turhalta suorittaa ensimmäisen vaiheen testit paikallisesti ennen muutosten lisäystä versionhallintaan, koska tiedossa on niiden ajaminen käännös putkessa heti versionhallintaan tallentamisen ja koodin kääntämisen jälkeen. On olemassa kuitenkin kaksi yleistä skenaariota, joissa malli auttaa käännöstä pysymään ehjänä. [1.s.67]

Skenaario 1:

Suunnittelija B on lisännyt muutokset versionhallintaan sen jälkeen, kun suunnittelija A on edellisen kerran hakenut koodit versionhallinnasta. Versionhallinnasta paikalliselle työasemalle haetut uusimmat koodit yhdistettyinä suunnittelijan A tekemiin muutokset voivat hyvinkin aiheuttaa sen, etteivät testit mene läpi. Noudattamalla mallia vältytään käännöksen rikkoutumiselta, koska muutosten aiheuttamat ongelmat ovat jo tiedossa ennen muutosten viemistä versionhallintaan.

Skenaario 2:

Suunnittelija A on unohtanut lisätä ohjelmiston tarvitsemat konfiguraatiotiedostot versionhallintaan ja testit menevät läpi paikallisesti, mutta eivät CI-putkessa. Tämä voi olla suora ja nopea viesti suunnittelijalle A, että hän on unohtanut viedä joitakin uusia tiedostoja versionhallintaan. Mikäli kaikki tarvittava on viety versionhallintaan voi suunnittelija A päätellä, että suunnittelija B on lisännyt versionhallintaan jotain tällä välillä.

Esitestattu versionhallintaan vienti (Pretested commit) on ominaisuus, jonka modernit CI-serverit tarjoavat tukemaan käännösputken ensimmäisen vaiheen testien ajamista ennen koodin lisäystä versionhallintaan. Tällöin kehittäjän tekemä muutosten vienti ei mene suoraan versionhallintaan, vaan ne käännetään ja testataan CI-serverin toimesta versionhallinnassa olevien uusimpien koodien kanssa. Muutokset tallennetaan versionhallintaan ainoastaan, mikäli testit menevät läpi. Muussa tapauksessa kehittäjä saa CI-serveriltä välittömän palautteen ja mitään ei lisätä versionhallintaan. [1.s.67]

Siirrytään eteenpäin vasta, kun ensimmäisten vaiheen testit ovat menneet läpi

CI-systeemi on tiimin jaettu resurssi, mikä jokaisen kehittäjän täytyy ymmärtää. Kun muutokset on viety versionhallintaan, seurataan sen jälkeen käännöksen etenemistä, kunnes tulokset siitä ovat valmiina. Seuraavaan tehtävään ei siirrytä, ennen kuin koodi on käännetty ja testit ovat menneet läpi. Mikäli putki ei jostain syystä mene läpi, on kehittäjän velvollisuus istua takaisin työn

ääreen ja korjata ongelma. Mikäli korjaamiseen ei sillä hetkellä ole aikaa, palautetaan versionhallintaan viimeisin toimiva versio ja tehdystä työstä otetaan varmuuskopio, kunnes vian aiheuttaja voidaan selvittää. [1.s.68]

Ole aina valmis palauttamaan edelliseen versioon

Vaikka kuinka yrittäisimme vältellä käännöksen rikkomista, tulee se jokaisen kehittäjän eteen jossain vaiheessa työuraa. Yleensä korjaukset ovat yksinkertaisia muutaman rivin muutoksia, mutta joissakin tilanteissa juurisyytä voi olla todella vaikea löytää. Tämän kaltaisissa tilanteissa onkin usein parempi palata versionhallinnan viimeisimpään toimivaan versioon ja korjata ongelma paikallisessa kehitysympäristössä. Helppo palauttaminen viimeisimpään toimivaan versioon onkin versionhallinnan yksi päätarkoitus. Ennen muutosten lisäämistä versionhallintaan onkin aina hyvä varautua, sekä olla tietoinen tarvittavista toimenpiteistä viimeisimmän toimivan version palauttamiseksi, mikäli tehdyt muutokset rikkovat käännöksen. [1.s.69]

Rikkinäisen käännöksen korjaus aika ikkunassa

Eräs tiimien kesken sovittava sääntö voi olla käännöksen korjaaminen tietyssä aika ikkunassa. Sääntö voi mennä esimerkiksi siten, että mikäli käännöstä ei ole onnistuttu korjaamaan 10 minuutin kuluessa, palautetaan versionhallinnan viimeisin toimiva versio. Tietenkään mikään ei saa olla kiveen hakattua ja mikäli aikaikkunan täyttyessä ollaan lähellä ratkaisua, kannattaa kortti katsoa ja usein se kannattaakin. Mikäli toivottua korjaavaa toimenpidettä ei saada aikaiseksi määritellyssä ajassa, palataan edelliseen versioon. [1.s.70]

Ota vastuu seurauksista, jotka ovat aiheutuneet tekemistäsi muutoksista

Mikäli kehittäjä lisää muutokset versionhallintaan ja kaikki hänen itsensä tekemät testit menevät läpi, mutta muiden tekemät eivät, on käännös kuitenkin rikki. Yleensä tämä tarkoittaa, että kehittäjä on onnistunut tekemään ohjelmistoon regression, eli muuttunut ohjelman toimintaa tahattomasti. Tämän kaltaisissa tilanteissa regressiobugin korjaaminen on kehittäjän vastuulla, kunnes myös muiden tekemät testit menevät läpi. [1.s.70]

Älä kommentoi testejä, jotka eivät mene läpi

Joissain tilanteissa kehittäjillä voi olla suuri kiusaus kommentoida käännöksen rikkovat testit, jotta muutokset voidaan lisätä versionhallintaan. Tämä ei luonnollisestikaan edusta oikeita toimintatapoja. Usein voi olla hankala lähteä selvittämään syitä miksi testit eivät mene läpi ja yleensä ensimmäinen keino itsensä huijaamiseksi onkin ajatus siitä, että testi ei enää ole validi järjestelmään tehtyjen muutosten vuoksi. Näin voi olla, mutta on myös mahdollista, että testit ovat löytäneet ohjelmasta regression. Oli syy mikä tahansa, kommentoimisen sijaan tulee se selvittää tarvittavien henkilöiden kesken. Vasta seuraavassa vaiheessa korjataan koodi, mikäli regressio on löytynyt, tai muokataan testiä, mikäli testin läpimenoehdot ovat muuttuneet. Testi voidaan myös poistaa, mikäli sen testaamaa ominaisuutta ei enää ole. Mikäli tätä periaatetta ei noudateta, voidaan helposti päätyä tilanteeseen, jossa suuri osa testeistä on kommentoituina. [1.s.70]

Testivetoinen kehitys

Kun muistetaan, että toimivan CI-prosessin tärkein hyöty on nopea palaute, on selvää, että se voidaan saavuttaa ainoastaan kattavan yksikkötestauksen avulla. Hyväksyntätestaus on myös tärkeää, mutta sen tuoma palaute kestää kauan. Kokemuksen mukaan kattavin yksikkötestaus voidaan saavuttaa testivetoisen kehityksen avulla. Periaatteellisella tasolla TDD tarkoittaa uutta koodia testaavan testin tekemistä ennen uuden koodin implementointia. Ideaalisella tasolla testi menee läpi vasta, kun ominaisuus on valmis tai ohjelmistovika korjattu. [1.s.71]

Koodin uudelleenjärjestely

Koodin uudelleenjärjestelyllä (refactor) tarkoitetaan koodin rakenteen muuttamista ilman, että ohjelman toiminta muuttuu. Käytännössä tämä on yleensä teknisen velan maksamista. CI-prosessi sekä testivetoinen kehitys tekevät koodin uudelleenjärjestelystä turvallista, koska tällöin voidaan suurella todennäköisyydellä olettaa, etteivät muutokset ole rikkoneet toiminnallisuutta. Tällöin myös laajalti ohjelmiston eri osiin koskevat uudelleenjärjestelyt ovat turvallisia tehdä. [1.s.72]

Arkkitehtuuria testaavat testit

Joissakin tilanteissa voi olla hyvä tehdä testejä hajautettujen järjestelmien välisien arkkitehtuurisäännösten noudattamiselle. Testeistä voi olla hyötyä eteenkin silloin, kun tiettyjen sääntöjen

noudattamatta jättäminen aiheuttaa ongelmia vasta pidemmän ajan kuluttua arkkitehtuuristen virheiden tekemisestä. Tässä vaiheessa kehitys voi jo olla niin pitkällä, että toteutusta on enää hankala korjata. [1.s.72-73]

Käännöksen rikkominen liian pitkäkestoisten testien vuoksi

Toimivan CI-prosessin perustuessa usein pienten muutosten lisäämiseen versionhallintaan, voi tuottavuus kärsiä huomattavasti, mikäli putken ensimmäisen vaiheen testit kestävät liian kauan. Mikäli kehittäjät joutuvat odottamaan liian kauan käännösten valmistumista, ei muutoksia todennäköisesti jakseta tehdä enää riittävän usein. Tämä puolestaan tekee jokaisesta lisättävästä muutoksesta kompleksisemman, aiheuttaen todennäköisemmin koodikantojen yhdistämisestä (merge) aiheutuvia konflikteja ja kaikki hidastuu entisestään. Eräs keino pitää kehitystiimin huomio myös testien suoritusajoissa on asettaa testeille maksimikesto, jonka ylittäminen rikkoo käännöksen. Aika voidaan asettaa esimerkiksi kahteen sekuntiin, mikä esimerkiksi yksikkötestille on jo huomattavan pitkä suoritusaika. Aikarajan asettaminen voi auttaa myös kehittäjiä tekemään fiksumpia testejä, jotka testaavat asioita lyhyemmässä ajassa. [1.s.73]

Käännöksen rikkominen laatukriteerein

Erilaisten kooditarkistimien (code linter), analyysityökalujen sekä kääntäjän antamiin varoituksiin on tavallisesti jokin perusteltu syy. Vaikka tiukkojen sääntöjen noudattaminen voi joskus tuntua ikävältä, tekevät ne kuitenkin koodista yleensä kokonaisuudessaan yhdenmukaisempaa sekä laadullisesti parempaa. Erilaisilla analyysityökaluilla voidaan tehdä lukematon määrä erilaisia tarkistuksia halutuin parametrein ja niiden avulla ohjelmakoodin laatu voidaankin aluksi saada näyttämään toivottomalta kaikkine virheineen ja varoituksineen. Tässä tilanteessa ei välttämättä olekaan paras vaihtoehto rikkoa käännöstä, kunnes kaikki varoitukset on korjattu, vaan sen sijaan asettaa laatuportteja (quality gate), jotka esimerkiksi tarkistavat, ettei varoituksia ole tullut lisää, tai ne ovat vähentyneet tietyllä määrällä. [1.s.74]

Tuotetaan nopeaa ja hyödyllistä palautetta

Kun ohjelmistoviat havaitaan mahdollisimman pian niiden tultua järjestelmään, muutokset ovat vielä tällöin tuoreessa muistissa sekä mekanismi virheen paikantamiseen yksinkertaisempi. Kun koodimuutokset aiheuttavat käännöksen rikkoontumisen, ei syy ole välttämättä heti ilmeinen. Tällöin etsintä voidaan keskittää viimeisimmän toimivan kokoonpanon jälkeen tehtyihin muutoksiin. Mikäli versionhallintaan tehdyt lisäykset ovat olleet kooltaan pieniä, on muutos helppo paikantaa pieneen määrään koodia. [1.s.171]

Vaikka puhutaan nopeimmasta mahdollisesta palautteesta, ei ensimmäistä vaihetta välttämättä kannata pysäyttää ensimmäiseen virheeseen. Putki pysäytetään ainoastaan, mikäli virhe estää sen jälkeen tapahtuvia toimintoja. Näin toimitaan esimerkiksi käännösvirheen kohdalla, mikä selkeästi on este suorittaa loput putken vaiheista. Sen sijaan putki on mahdollisuuksien mukaan hyvä suorittaa loppuun, jolloin palautteena saadaan koottu virheraportti yhdellä kertaa. [1.s.172]

Käännöksen hylkäämiseen liittyvät käytännöt

Perinteisen mallin mukaan ensimmäinen vaihe joko menee läpi, tai sitten ei. Tässä tilanteessa onkin hyvä miettiä, antaako läpi mennyt vaihe todellisen kuvan laadusta, mikäli se sisältää satoja kääntäjävaroituksia tai yksikkötestejä on ainoastaan muutamia. Vaiheen tuloksen esittäminen pass/fail -statuksella voikin helposti muodostua virhepositiiviseksi. Varteenotettava vaihtoehto onkin esittää tulos esimerkiksi liikennevaloina perustuen vaiheesta mitattuun statistiikkaan. Käännös voidaan esimerkiksi hylätä, jos yksikkötestien kattavuus on alle 60% ja vastaavasti päästää läpi keltaisena, jos se on alle 80%. [1.s.172]

Olipa hylkäysperuste sitten mikä tahansa, niin perussääntöä noudatellen käännöksen rikkoneet muutokset toimittanut tiimi keskeyttää muun tekemisen ja korjaa käännöksen toimivaksi tarvittavien toimenpiteiden avulla. Käännöstä ei myöskään saisi hylätä turhin perustein. Koko tiimin tulee esimerkiksi hyväksyä syyt, joiden vuoksi tietty testi ei ole mennyt läpi. Muutoin testejä ei enää oteta vakavasti ja koko prosessi menettää merkityksensä. [1.s.172]

Annetaan suunnittelijoille omistajuus

Vaihe sisältää erinäisiä skriptejä kääntämistä ja testien ajamista varten. Näiden skriptien ylläpidon tulee olla samalla viivalla minkä tahansa muun ohjelman osan kanssa. Mikäli tästä osiosta tehdään hankala ja heikosti ylläpidettävä, joudutaan putken ylläpitoon tällöin käyttämään turhaan resursseja ja kaikki on poissa uuden kehityksestä. Putkien kehitystä sekä ymmärrystä niiden toiminnasta ei pidä eriyttää pelkästään muutamalle spesialistille, vaan sen sijaan niihin liittyvä perustoiminnallisuus on hyvä antaa kehitystiimien hallintaan. Kun tiimeille annetaan tietynlainen omistajuus julkaisuputkiin, ei spesialistien tarvitse tämän jälkeen tehdä jokaista muutosta tai lisäystä julkaisuputkeen. Sen sijaan näiden henkilöiden ammattitaitoa voidaan paremmin hyödyntää esimerkiksi uusien ratkaisujen löytämiseksi tarvittavassa tutkimustyössä. [1.s.173-174]

Vältetään käyttöliittymän kautta tapahtuvaa testausta

Käyttöliittymän kautta tapahtuvaa testausta on hyvä välttää ensimmäisen vaiheen testeissä, koska se on huomattavan hidasta. Tästä syystä käyttöliittymän kautta tapahtuva testaus onkin hyvä ottaa mukaan vasta hyväksyntätestausvaiheessa. [1.s.178-179]

Käytetään riippuvuus injektioita

Luokkien testattavuutta voidaan mahdollisuuksien mukaan parantaa käyttämällä esimerkiksi riippuvuus injektioita (dependency injection). Mallin avulla voidaan poistaa luokkien riippuvuuksia ulkoisiin komponentteihin injektoimalla riippuvuuskomponentti testattavaan luokkaan suoraan testistä. Yleensä injektointi tarkoittaa testissä alustetun luokkainstanssin välittämistä testattavaan luokkaan esimerkiksi parametrina. Tämä tekniikka auttaa tekemään ohjelmakoodista modulaarista ja helpottaa yksikkötestausta, koska testit voidaan kohdentaa juuri haluttuihin luokkiin, ilman että niiden riippuvuusketjuja täytyy ottaa mukaan. [1.s.179]

Vältetään tietokantoja

Perinteinen automaattitesti esimerkiksi tietokantasidonnaisissa ohjelmissa on, että testi kommunikoi jonkin ohjelmakerroksen kanssa, tallentaa arvot kantaan ja lopuksi tarkistaa, että ne löytyvät sieltä. Vaikkakin malli on helppo ymmärtää, ei se välttämättä ole tehokas lähestymistapa ongelmaan. Tämän kaltaiset testit ovat erittäin hitaita ajaa, eivätkä testin suoritusolot ole välttämättä samat esimerkiksi peräkkäisillä ajokerroilla. Testin vaatima infran kompleksisuus lisää myös testin kompleksisuutta niin tekemisen, kuin ylläpitämisenkin kannalta katsottuna. Mikäli tietokannan eristäminen testeistä on hankalaa, kertoo se huonosta arkkitehtuurista. Vaatimus ohjelman testattavuuden parannustarpeista onkin hyvä tapa aiheuttaa hienovaraista painetta kehitystiimeille paremman koodin kehittämiseksi. Yksikkötestien kytkeytymistä suoraan tietokantaan tulee välttää. Tämä edellyttää hyvää arkkitehtuuria sekä esimerkiksi riippuvuus injektioiden käyttöä. [1.s.179-180]

Vältetään asynkronisia yksikkötestejä

Asynkronisen toiminnallisuuden testaus yksikkötestillä on haastavaa. Yksinkertainen lähestymistapa on välttää asynkronisia testejä esimerkiksi jakamalla asynkronisen toiminnon testi kahteen osaan. Perinteisesti asynkroninen toiminta liittyy vuorovaikutukseen, jossa vastausta ei saada paluuarvona heti kysymyksen lähettämisen jälkeen. Tämänkaltaisissa tilanteissa testi voidaan jakaa kahteen osaan ja ensimmäisellä testillä voidaan tällöin testata viestin lähetys ja toisella vastaanottaminen. Viestin vastaanottamista testaava testi voi hyödyntää esimerkiksi viestinkäsittelijärajapintaa simuloivaa komponenttia. Asynkronisia testejä tässä vaiheessa tuleekin välttää. [1.s.180]

Käytetään testituplia

Modulaarisesti suunnitellun järjestelmän testaamisessa yleiseksi ongelmaksi muodostuu yleensä riippuvuusketjujen keskivaiheilla olevien moduulien tai luokkien testaus, mikä voi vaatia pitkiä ja hitaita riippuvuusketjujen alustuksia. Eräs ratkaisu tämän kaltaisiin ongelmiin on esimerkiksi korvata riippuvuusketjun osa moduulia simuloivalla koodilla (test stub). Simuloitu koodi tekee tällöin juuri sen mitä sen halutaan testissä tekevän, jolloin ulkoiset vaikutteet testiin voidaan eliminoida. [1.s.180-181]

Simulointia nykyaikaisempi tapa eristää luokkia ilman moduuleja simuloivan koodin kirjoittamista on matkia riippuvuutta aiheuttavan luokan ulospäin näkyvää käyttäytymistä (mock object). Luokkien eristäminen tekniikkaa hyödyntäen toimii käytännössä siten, että tietyn työkalun avulla generoidaan luokka, joka voi teeskennellä olevansa esimerkiksi tyyppiä X. Käytännössä menetelmän avulla testattavan koodin toiminnallisuus voidaan määritellä muutaman yksinkertaisen tarkistuksen avulla, ja lisäksi voidaan, että koodi on vuorovaikutuksessa mock -objektin kanssa odotetulla tavalla. Tekniikan hyödyt ovat selkeät, koska sitä hyödyntämällä testikoodin tarve on huomattavasti vähäisempi ja sen avulla voidaan säästää paljon tärkeää työaikaa. [1.s.181-182]

Minimoidaan testien riippuvuus ohjelman tilasta

Ideaalisessa tilassa yksikkötestit todentavat järjestelmän toimintaa, mutta usein voidaan havaita niiden todentavan testien ulkopuolisia asioita. Mitä monimutkaisimmiksi testit menevät, sitä enemmän niissä on testin ulkoisia riippuvuuksia. Varsinaisten testien lisäksi tuleekin kiinnittää huomiota niiden vaatimaan infraan. Monimutkaiset testit ja niiden ympäristöt kertovat testattavan koodin arkkitehtuurisista ongelmista ja kielivät koodin huonosta testattavuudesta. [1.s.184]

Tehdään testit aika riippumattomiksi

Aika on perinteisesti automaattitestauksen yksi suurimmista ongelmista, koska yleisesti aikaan tai ajastimiin liittyvä toiminnallisuus on toteutettu käyttäen suoraan järjestelmän kelloja. Eräs strategia aikariippuvaisten toiminnollisuuksien testaamista varten on eristää aikaan liittyvä toiminnallisuus omaan luokkaan. Tällöin ajan hallinta on omissa käsissä ja se voidaan antaa testattavalle metodille esimerkiksi luokan riippuvuus injektiona, jolloin esimerkiksi vuosi saadaan helposti kulumaan kymmenessä millisekunnissa. [1.s.184]

Lisätään rautaa

Vaikka ensimmäisen vaiheen nopeus asetetaankin usein kriittisimmäksi tekijäksi, kannattaa vaihe kuitenkin tasapainottaa virheidenlöytämiskyvyn sekä nopeuden välillä. Tämä on prosessi, joka voi onnistua ainoastaan yrittämisen ja erehtymisen kautta. Joskus on paikallaan hyväksyä hieman pidempi kesto, kuin optimoida testejä liikaa tai tinkiä niiden kyvystä löytää ohjelmistovirheitä. Jotta vältyttäisiin liian pitkään kestävältä testaukselta, voidaan testit edellä mainittujen toimenpiteiden lisäksi ajaa esimerkiksi rinnakkain usealla koneella. Perus muistisääntönä voidaankin pitää, että laskentateho on halpaa ja henkilötyö kallista. Palautteen saaminen nopeasti on paljon arvokkaampaa, kuin muutaman serverikoneen kustannus. Eräs vaihtoehto on myös muokata ensimmäisen vaiheen testejä siten, että pitkäkestoiset testit, jotka löytävät harvoin vikoja siirretään hyväksyntätestausvaiheeseen. [1.s.185]

Testattavan ohjelman ajaminen tiettyyn tilaan suoritettavaa testiä varten

Hyväksyntätestien tarkoituksena on simuloida ja tuottaa järjestelmään sitä kuormittavia syötteitä ja todistaa, että se täyttää sille asetetun käyttäjätason vaatimuksen. Sen tarkoitus on varmentaa, että käyttäjä voi luottaa järjestelmän käsittelemään dataan, ja ilman tätä kykyä on järjestelmä merkityksetön. Kun puhutaan tilallisista testeistä, tarkoitetaan siinä lähinnä, että testattaessa ohjelman tiettyä ominaisuutta tai tilannetta, tulee testi käynnistää jostakin tunnetusta tilasta. Usein ohjelman määrittäminen tiettyyn luotettavaan lähtötilanteeseen voi kuitenkin olla vaikeaa. Se on kuitenkin edellytyksenä luotettavalle testille, jonka tilaan ja tulokseen voi luottaa. Vaikka testien tilannesidonnaisuutta tuskin koskaan voidaan täysin poistaa, niin kannattaa niiden riippuvuutta kompleksisiin tiloihin välttää. [1.s.204]

Kompleksisuutta voidaan vähentää esimerkiksi järkevällä testidatan hallinnalla. Testidatan ei ole hyvä koostua tuotantodatasta otettua tietokannan varmuuskopiota (database dump), vaan mieluummin helposti hallittavia pieniä datasettejä. Tärkeimpänä seikkana on saada luotua tunnettu lähtötilanne. Mikäli testidatalla halutaan jäljitellä täydellisesti tuotantoympäristön tilaa, voi datasetin toimintaan saattaminen viedä enemmän aikaa, kuin varsinainen testaus. Testauksen fokus tulee pitää järjestelmän käyttäytymisessä testidatan käyttäytymisen sijaan. Testidatan on hyvä olla mahdollisimman tehokas ja yhtenäinen setti dataa, jonka avulla järjestelmän toiminta voidaan verifioida. Ideaalisessa tilanteessa testit voidaan ajaa lähtötilanteeseen käyttäen ohjelman julkista rajapintaa, mikä on huomattavasti turvallisempi malli verrattuna massiiviseen tietokantaan perustuvaan lähtötilanteeseen. [1.s.205]

Hyväksyntätason testit eivät saa olla riippuvaisia toisistaan, eli niitä tulee voida suorittaa mielivaltaisessa järjestyksessä. Atominen testi luo kaiken minkä se tarvitsee suorittamista varten ja lopuksi siivoaa jälkensä jättäen taakseen ainoastaan testituloksen. Eräs keino on eristää testauksen kohde muusta järjestelmästä, mutta tämä voi usein vaatia tietynlaisia arkkitehtuurisia ratkaisuja sekä tarvittavien toiminnallisuuksien tekemistä sovelluksen ohjauskerrokseen (application driver layer). Testien eristäminen itsenäisiksi toisistaan riippumattomiksi tapahtumiksi voi myös mahdollistaa niiden rinnakkaisen ajamisen tietyissä tilanteissa, mikä puolestaan voi lyhentää ajettavan testisetin suoritusaikaa. [1.s.205]

Asynkronisten toimintojen sekä ajan hallinta

Asynkronisten järjestelmien testaus luo aina omat ongelmansa, mutta hyväksyntätestauksessa asynkronisia toimintoja on hyvin vaikea välttää. Usein vastaavanlaisia ongelmia tuottavat säikeistykseen sekä transaktioihin perustuvat toiminnallisuudet. Tämänkaltaisissa systeemeissä yksi kutsu voi joutua odottamaan toisen säikeen valmistumista. Ongelmaksi tällöin muodostuu, kuinka kauan odotellaan transaktion valmistumista ennen kuin tiedetään, ettei testi mene läpi. Eräs ratkaisumalli on synkronoida asynkroniset tapahtumaketjut esimerkiksi keskeytysten avulla. Tällöin kiinteiden maksimiaikojen (timeout) sijaan voidaan rekisteröidä jokaiselle syötteelle oma keskeytys, joka palauttaa testille halutun datan vasteen ollessa valmiina. [1.s.207-209]

Testituplien käyttö hyväksyntätesteissä

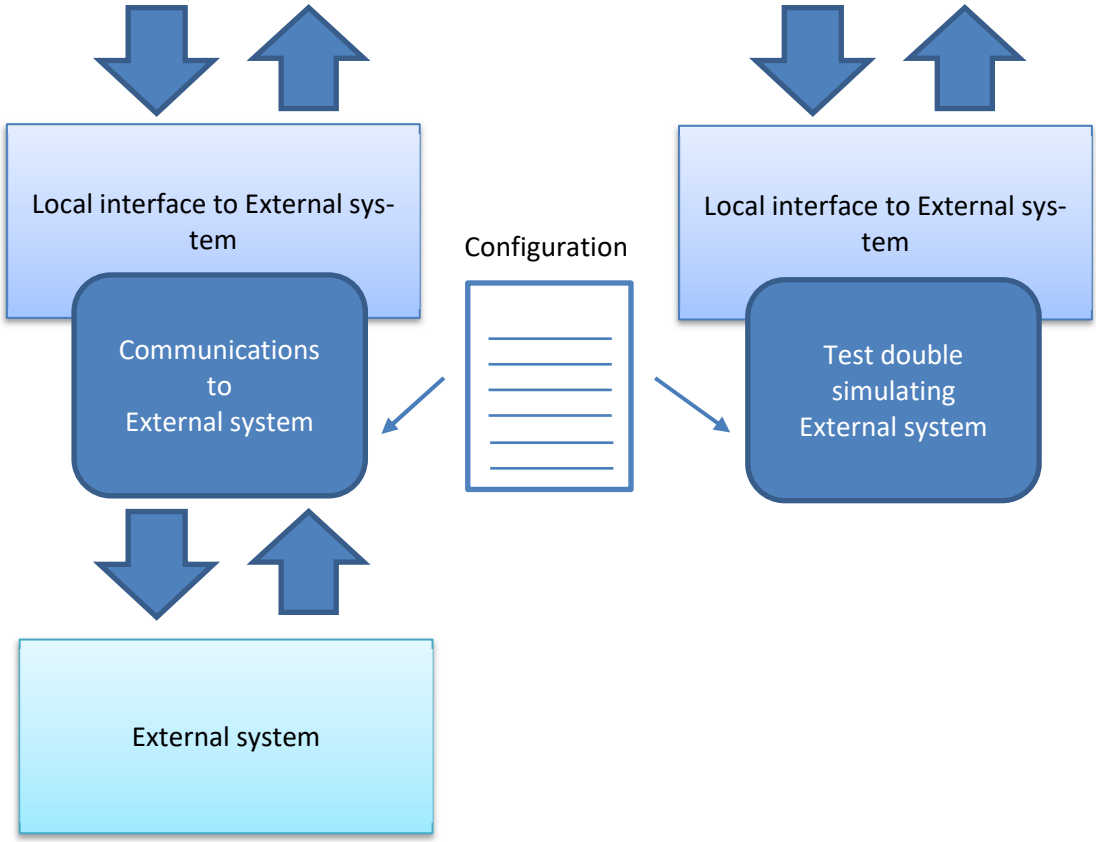
Eräs automaattitestauksen kulmakivistä on korvata jokin ohjelman osa tunnetulla simuloidulla versiolla (test double), jonka ajonaikainen käyttäytyminen on tunnettu sekä muuteltavissa halutuksi tarpeen mukaan. [1.s.91-92]

Hyväksyntätestaus luottaa automatisoitujen testien suorittamiseen tuotantoa vastaavassa ympäristössä, jonka tärkeimpiä ominaisuuksista on sen kyky tukea automaattista testausta. Poiketen manuaalisesta hyväksyntätestauksesta (user acceptance testing) automaattisia hyväksyntätestejä ei tule ajaa ympäristössä, joka on integroitu oikean elämän ulkoisiin järjestelmiin. Sen sijaan automaattisten hyväksyntätestien ajoympäristöä tulee voida täysin kontrolloitavissa, jolloin se esimerkiksi voidaan ajaa tunnettuun lähtötilaan. Tämä ei useinkaan ole mahdollista, mikäli järjestelmä on integroituneena useisiin ulkoisiin järjestelmiin. [1.s.210]

Ulkoisten riippuvuuksien vaikutus hyväksyntätestaukseen tulee voida minimoida. Liittyminen ulkoisiin järjestelmiin aiheuttaa usein hankalasti paikannettavia ongelmia ja lisäksi kontrolli ohjelman ajamiseksi tiettyyn tilaan usein menetetään. Lisäksi tietynlaiset testit voivat aiheuttaa odottamatonta kuormaa myös ulkoisten järjestelmien suuntaan. Hyväksyntätestausvaiheessa ulkoiseen järjestelmään kytkeytymistä voidaan simuloida luomalla kommunikointia mallintava komponentti simulaatio. Mallin avulla voidaan tunnetun lähtötilanteen lisäksi kontrolloida ja simuloida esimerkiksi erinäisiä virhetiloja, yhteysvirheitä, vasteaikoja, kuormitusta ynnä muuta sellaista. [1.s.210]

Hyvä malli on luoda erillinen, kuvan 17 mukainen kommunikaatiorajapinta tai komponentti jokaiselle erilliselle ulkoiselle järjestelmälle. Tämä komponentti tiivistää kommunikoinnin sekä kaiken

siihen liittyvän yhteen paikkaan ja eristää siihen liittyvät ongelmat muusta järjestelmästä. Tämä komponentti on linkki ohjelman ja ulkoisen järjestelmän kanssa ja sen toiminta tulee voida todistaa molempiin suuntiin. Simulaation avulla voidaan näyttää toteen, että järjestelmän kommunikointi rajapinnan kautta toimii molempiin suuntiin. [1.s.211]



Kuva 17. Simulaatio ulkoiselle järjestelmälle [1.s.211]

Integraatiopisteet ulkoisten järjestelmien kanssa ovat yleisiä ongelmanlähteitä, koska mikä tahansa muutos voi tehdä järjestelmistä yhteensopimattomia keskenään. Tästä syystä integraatiota testaavien testien onkin hyvä keskittyä mahdollisiin ongelmapaikkoihin ja yleensä ne ovat hyvin järjestelmäkohtaisia. Kokemuksen mukaan jokaisessa integraatiossa on kuitenkin olemassa muutamia perustyypin skenaarioita, joita voidaan simuloida. Jokaiselle näistä skenaarioista on hyvä luoda pieni setti perus testejä. Tämän jälkeen voidaan jokaista löytynyttä bugia vastaan tehdä automaattitesti. Tällöin saadaan kohtuullisessa ajassa luotua integraatiota testaava setti, joka löytää yleisimmät viat. [1.s.212]

Käyttöliittymän kautta tapahtuva testaus

käyttöliittymätestaukseen liittyy muutamia yleisesti tunnettuja ongelmia. Ohjelman käyttöliittymä muuttuu yleisesti usein ohjelman kehityksen ollessa käynnissä, jolloin huonosti suunnitellut käyttöliittymätestit yleensä rikkoontuvat. Käyttöliittymän kautta tapahtuvat testiskenaarioiden alustukset voivat olla huomattavan monimutkaisia ja hitaita. Tällöin yleensä joudutaan navigoimaan ohjelman tiettyyn osioon ja asettamaan sieltä tietyt arvot ennen kuin varsinainen testi voidaan suorittaa. Testien tarvitsemat asiat voivat joskus olla myös hankalasti saatavilla käyttöliittymän kautta. Tietyt käyttöliittymäteknologiat ovat erittäin hankala testata automaattisesti, joten niiden valintaan kannattaa kiinnittää huomiota jo projektin suunnitteluvaiheessa. [1.s.192]

Tietynlaisissa sovelluksissa ohjelman käyttö käyttöliittymän kautta voidaan ohittaa kokonaan, mikäli ohjelman arkkitehtuuri on suunniteltu siten, että käyttöliittymäkerros sisältää pelkästään käyttöliittymäkoodia, joka puolestaan käyttää varsinaisen toiminnallisen kerroksen rajapintaa. Tällöin ohjelman testipeti voi käyttää suoraan rajapintaa, jota käyttöliittymä normaalisti käyttää. Tämä kuitenkin vaatii hyvää suunnittelua ja kuria kehitystiimeiltä, jotta käyttöliittymäkerroksen koodissa keskitytään ainoastaan pikselin viilaamiseen sen sijaan, että siellä tehtäisiin ohjelman muuhun toiminnallisuuteen liittyviä asioita. [1.s.192-193]

Testien sovitus kerros (Test implementation layer)

Sovitus kerros erillinen rajapinta ohjelman julkisten rajapintojen sekä niitä käyttävien testien välillä. Kerroksen tarkoituksena on vähentää pienistä ohjelmamuutoksista johtuvaa testien särkymistä, jolloin työlään testien päivittämisen sijaan korjaukset voidaan tehdä yhteen paikkaan rajapinnassa. [1.s.191]

Ohjelman ajurikerros (Application driver layer)

Kerros, jonka avulla ohjelman toiminnallisuutta voidaan käyttää selkokielisin funktiokutsuin. Hyvin suunniteltuna se voidaan sovittaa suoraan hyväksyntäkriteereihin, jotka siten voidaan siirtää testeihin lähes sellaisenaan. Esimerkkinä voidaan käyttää perinteistä käyttäjän luontia, jossa kerros voisi sisältää funktion luoKäyttäjä(nimi, sukunimi, …). Kutsu on itsessään selkokielinen ja se

voi käsittää yhden rivin automaattitestistä, jolloin testeistä tulee luettavia ja helposti ymmärrettäviä. Vaikka ohjelma tekisikin paljon asioita konepellin alla, on tämä malli erittäin käyttökelpoinen, kun testit halutaan paketoida lyhyiksi, selkeiksi kokonaisuuksiksi, joita sitten pystytään ajamaan erillisin parametrisetein. [1.s.198-199]

Eräs seuraus hyvin suunnitellusta ajuri kerroksesta on parantunut testien luotettavuus. Testeissä käytettävät toiminnalliset avainsanat voidaan kirjoittaa kerran ja niitä voidaan uudelleen käyttää useissa testeissä. Mikäli testeissä ilmenee ongelmia, ovat ne helposti jäljitettävissä ja korjattavissa yhteen paikkaan, mikä entisestään lisää testien luotettavuutta. Kerroksen rakentaminen voidaan aloittaa muutamasta testistä ja sen käyttö tulee opettaa tiimille. Tämän jälkeen tiimit lisäävät sinne uusien ominaisuuksien testaamiseen tarvittavat kutsut tai päivittävät olemassa olevia testien niin vaatiessa. [1.s.200]

Gherkin -syntaksi

Yleisesti ottaen jokaiselle ohjelman vaatimukselle voidaan tunnistaa käyttäjän yleisimmistä toimista koostuva säännönmukainen polku (happy case). Polku voidaan helposti muuttaa käyttämään Gherkin syntaksia (Given-When-Then), jossa avainsanoilla on seuraavat merkitykset: [1.s.85] [16]

Feature: Skenaarion otsikko / nimi

Given: Muutama järjestelmän tilaa kuvaava ehto ennen testin käynnistymistä

When: Setti käyttäjän tekemiä toimenpiteitä

Then: Muutama avainsana tilasta, johon järjestelmän tulee siirtyä

Useimpia käyttötapauksia (use case) voidaan lähteä suorittamaan useista eri tilanteesta, tapauksen aikana voidaan antaa monenlaisia syötteitä ja näiden seurauksena voidaan päätyä monenlaiseen lopputulemaan. Nämä variaatiot muodostavat usein selkeitä eri käyttötapauksia, jotka voidaan tunnistaa vaihtoehtoisina polkuina yleisimmälle käyttötapaukselle. Muiden, kuin ohjelmakoodin tukemien käyttötapausten tulee johtaa jonkinlaiseen virheenkäsittelyyn. Kaikkia mahdollisten yhdistelmien testaaminen voi joskus olla haastavaa, mutta usein riittääkin, että testeihin otetaan mukaan potentiaalisimmat vikoja aiheuttavat kombinaatiot. Hyväksyntätestit on hyvä tehdä aina ympäristössä, joka on mahdollisimman lähellä tuotantoympäristöä. [1.s.85-86]

Testit suoritettavina vaatimusmäärittelyinä

Hyvin suunniteltuna automaattinen testaus ei enää ole pelkkää testaamista, vaan automaattiset hyväksyntätestit toimivat samalla suoritettavina testispesifikaatioina järjestelmälle. Tämä on merkittävä oivallus, joka on tuonut uuden näkökulman automaattiseen testaukseen. Käytännössä tämä voi tarkoittaa, että hyväksyntäkriteerit kirjoitetaan muotoon, joka vastaa asiakkaan odotuksia ohjelman toiminnallisuudesta (behavior-driven development). Hyvin tehdyn pohjatyön jälkeen kirjoitettujen hyväksymiskriteerien pohjalta voidaan tällöin suoraan generoida automaattitestejä. Vastaavasti testit on mahdollista kirjoittaa sellaiseen muotoon, että ne ovat luettavia ja

toimivat itsessään ajettavana spesifikaationa. Tämä malli tuo merkittävän edun siihen, että luodut spesifikaatiot eivät vanhene, vaan ne elävät toiminnallisuuden mukana. [1.s.195]

Käytännössä suoritettavien vaatimusmäärittelyiden käyttöönotto vaatii prosessimallia, jossa hyväksymiskriteerit on määritelty selkeästi ja kirjoitettu ne ylös suoritettavaan muotoon. Mikäli testattavan ohjelman ajurikerros ei tue määritellyn vaatimuksen täydellistä suorittamista, tulee sinne lisätä toteutus, jotta testi voidaan suorittaa. Käytännössä lisättävät toiminnot voivat olla avainsanoina toimivia funktiokutsuja, jotka suorittavat ohjelman toiminnallisia lohkoja. [1.s.197-198]

INVEST -periaate

Laadukkaiden automaattitestien tekeminen edellyttää hyvää toteutettavien töiden suunnittelua. Yksittäiset työt on hyvä suunnitella ja työstää kuvassa 18 esitettyä INVEST –periaatetta hyödyntäen, ja niille tulee olla määriteltynä selkeät hyväksymiskriteerit. Mikäli auki kirjoitettu työ ei täytä jotakin näistä periaatteista, voi sen uudelleenmäärittely tulla tällöin kysymykseen. [1.s.92] [15]

Kuva 18. INVEST periaate [15]

Ei sidota hyväksyntä testejä liian tiukasti ohjelmakoodiin

Hyväksyntätestit on hyvä suunnitella siten, etteivät ne ole sidottuna liian tiukasti ohjelmakoodiin, vaan sen sijaan niiden on ennemminkin voitava osoittaa ohjelman kyky tuottaa arvoa sen käyttäjille. Liian tiukasti ohjelmakoodiin sidottujen testien ylläpito muodostuu yleensä huomattavan työlääksi, koska pienet toiminnalliset muutokset rikkovat testit. Kirjoitettiinpa testit sitten millä kielellä / työkalulla hyvänsä, tulee niiden abstraktiotason vastata ohjelman hyväksymiskriteereiden kirjoitustasoa. Tämä voidaan havainnollistaa esimerkiksi seuraavasti: Toiminnon x käynnistämisen hyväksyntätesti on hyvä mieluummin kirjoittaa tasolla ”käynnistä toiminto x”, kuin että ”paina toiminnon x käynnistys nappia”. Väärin toteutettuna automaattinen hyväksyntätestaus voi tulla myös hyvin kalliiksi. Mikäli tämänkaltaiseen tilanteeseen on jouduttu, tulee ensi tilassa analysoida tapa, jolla testejä tehdään ja ylläpidetään. Oikein tehtynä testiautomaation kustannuksia voidaan pienentää dramaattisesti ja niiden hyötysuhdetta voidaan parantaa merkittävästi. [1.s.126]

Hyväksyntätestien omistajuus

Perinteinen malli hyväksyntätestien vastuunalaisuudesta on se, että ne ovat erillisen testaustiimin vastuulla. Tyypillinen ongelma tässä ajattelutavassa on, että testaustiimi on yleensä ravintoketjun pohjalla, joten hyväksyntätestit viettävät suurimman osan ajastaan punaisena. Tilanteessa, jossa kehitystiimi lisää muutoksia versionhallintaan ymmärtämättä niiden vaikutusta hyväksyntätestaukseen, tulevat muutokset testaustiimille suhteellisen myöhäisessä vaiheessa. Mikäli testaustiimillä on tässä vaiheessa lisäksi muutaman muun kehitystiimin testit korjattavana, seuraa tästä poikkeuksetta, että testit laahaavat auttamatta kehityksen perässä. Tällöin korjauksia on priorisoitava ja uusien testien tekemisestä joudutaan tinkimään. Ratkaisuna ongelmaan on vastuun siirtäminen kehitystiimeille, jolloin esimerkiksi kehittäjät kiinnittävät paremmin huomiota vaatimusten hyväksymiskriteerien täyttymiseen. Lisäksi he ovat nopeammin tietoisia muutosten vaikutuksesta hyväksyntätestaukseen. [1.s.215]

Pidetään hyväksyntätestit vihreänä

Hyväksyntätestausvaihe voi tunnetusti kestää useita tunteja, mistä syystä sen suoritus tulee vasta julkaisuputken loppuvaiheessa. Tästä puolestaan seuraa, että siinä ajettavat testit eivät yleensä riko käännöstä ja siksi ne voi olla helppo jättää huomiotta. Hyväksyntätestien pitäminen vihreänä vaatii tietynlaista kurinalaisuutta niistä vastuussa olevalta kehitystiimiltä. Kun jokin hyväksyntävaiheen testi ei mene läpi, tulee tiimin mahdollisimman pian ottaa asia käsittelyyn ja selvittää seuraavat asiat: [1.s.215]

- Onko testi itsessään rikki
- Johtuuko vika konfiguraatiosta
- Onko testi enää validi
- Onko kyseessä oikea ohjelmistovika.

Toimenpiteitä tulee kuitenkin tehdä heti, jotta testi ei jää epämääräiseen tilaan. Mitä sitten tapahtuu, mikäli hyväksyntätestien annetaan mädäntyä? Todennäköinen skenaario tilanteessa on, että julkaisun lähetessä hyväksyntätestejä yritetään kiireellä saada läpi, jotta julkaisun laadusta voitaisiin saada tietty varmuus. Kun testejä aletaan viimein käymään läpi, on jo vaikea sanoa miksi

ne eivät mene läpi, koska syitä voi olla monia. Paha, mutta yleinen virhe tässä vaiheessa on, että testejä poistetaan, tai muuten vaan jätetään huomiotta, koska enää ei ole aikaa juurisyiden selvittämiseen. On palattu takaisin lähtöruutuun, eli tilanteeseen, johon jatkuvan integroinnin avulla ei olisi ikinä pitänyt päätyä. On kiire saada kaikki toimimaan ilman tietämystä siitä kauanko asioiden korjaaminen kestää, koska koodin toiminnallisesta tilasta ei ole tarkkaa tietoa. Tärkeintä on siis pitää hyväksyntätestit vihreänä, jolloin se myös tuottaa sille määritellyn arvon ja maksaa takaisin siihen sijoitetun työpanoksen. [1.s.216]

Liiketoiminnan edustajat

Vaatimusten analysointia sekä asiakkaalle esittelyä varten tulee projektissa olla aina tehtävään määritelty henkilö. Henkilön on hyvä olla mukana kehitystiimin toiminnassa ja varmistaa, että suunnittelijat ovat ymmärtäneet ohjelman toiminnan käyttäjän näkökulmasta. Lopuksi varmistutaan, että valmistuttuaan toteutetut ominaisuudet tuottavat niille määritellyn käyttöarvon. Liiketoiminnan edustajat ovat mukana luomassa hyväksymiskriteereitä, varmistavat että ne on määritelty asiallisesti ja ominaisuus on valmis vasta, kun nämä määritykset on täytetty. Iteratiivisissa toimitusprosesseissa liiketoiminnan edustajat käyttävät paljon aikaa hyväksymiskriteerien määrittelyyn. Näiden kriteerien perusteella tiimit katsovat tietyn vaatimuksen täytetyksi. [1.s.193-194]

Testaajat

Testaajat ovat olennainen osa jokaista projektia. Heidän vastuullansa on viime kädessä varmistaa ohjelman tuotantokelpoisuus sekä riittävä laatu. Tehtävään kuuluu myös jakaa tietoa ohjelman tilasta kehitystiimille sekä asiakkaalle. Tätä työtä tehdään työskentelemällä asiakasrajapinnassa sekä määrittelemällä hyväksyntäkriteereitä liiketoiminnan edustajien kanssa. Lisäksi testaajat kirjoittavat automaattisia hyväksyntätestejä, sekä tekevät manuaalista testausta, kuten esimerkiksi tutkivaa testausta, hyväksyntätestausta sekä käyvät läpi näyttötapauksia. [1.s.193]

Kehittäjät

Ketterien ohjelmistokehitysmenetelmien puitteissa sovelluskehittäjän rooli yltää pelkkää ohjelmointia syvemmälle, joten siinä mielessä se poikkeaa jonkin verran perinteisestä ohjelmistokehityksestä. Kehittäjien velvollisuus on ymmärtää toteutettava ominaisuus käyttäjän kannalta katsottuna sen todellisessa käyttöympäristössä. Suunnitella sekä arvioida tehtävää työtä muiden tiimin jäsenten kanssa, jotta kaikilla toteutukseen osallistuvilla on riittävät tiedot toteutuksen tekemiseksi. Vaikka kehittäjä ei itse osallistuisi testaukseen, niin on tärkeää, että laatuvastuun ei kuitenkaan koeta olevan jonkun muun käsissä. [26]

Tiimi

Kehitystiimien tulee reagoida välittömästi, mikäli jokin hyväksyntätason testi ei mene läpi. Tässä tilanteessa tuleekin tutkia, johtuuko virhe regressiosta, muutoksista ohjelman toiminnallisuudessa vai virheestä testissä. Tämän jälkeen tehdään tarvittavat toimenpiteet, jotta testi saadaan takaisin vihreäksi, tai poistetaan testi. [1.s.124]

SWOT -analyysi

Eräs strategisessa suunnittelussa käytetty menetelmä on SWOT -analyysi. SWOT -analyysin ajatuksena on pohtia esimerkiksi tuotteiden ja palveluiden strategisia osa-alueita [11. s. 111], kuten:

- S -> vahvuuksia (toivottuja ylläpidettäviä tiloja)
- W -> heikkouksia (realisoituneita uhkia)
- O -> mahdollisuuksia (oikein hoidettuna tila realisoituu vahvuutena)
- T -> uhkia (toteutumaton heikkous, ennalta havaittuna voidaan kääntää mahdollisuudeksi)

Työkalun avulla on tarkoituksena saada selkeä kokonaiskuva organisaation tilasta strategisten valintojen tueksi. Hyvän SWOT -analyysin laadinta edellyttää resursseihin sekä toimintaympäristöön liittyvien analyysien tekemistä, koska ilman näitä, ymmärrys toimintaympäristön tilasta ei ole riittävä analyysin tekemiseksi oikeaoppisesti. Hyvin laaditun SWOT -analyysin avulla voidaan nostaa esiin pari keskeistä teemaa. Huonosti laadittu puolestaan tuottaa todennäköisesti pelkästään listan latteuksia ja itsestäänselvyyksiä. Pelkän asioiden listaamisen sijaan tavoitteeksi kannattaa asettaa analyysiin perustuvien strategisten valintojen ja toimintasuunnitelmien tekeminen. [12.s.88, 94]

Ongelmana analyysin käytössä strategisessa suunnittelussa on kuitenkin sen tulkinnanvaraisuus, koska strategisen suunnittelun perustana on tällöin johdon omasta näkökulmasta tehty analyysi, joka ei välttämättä vastaa todellista operatiivista henkilöstön toimintaa. Perusideana analyysissä on havaittujen uhkien kääntäminen mahdollisuuksien kääntäminen mahdollisuuksiksi ja heikkouksien vahvuuksiksi [11. s. 112].

Balanced Scorecard – tasapainotettu mittaristo (BSC)

Tasapainotettu mittaristo on strategiatyökalu tehokkuuden parantamiseen ja sen keskeisenä ideana on muuttaa strateginen tahtotila mittareiden sekä toimintasuunnitelmien avulla operatiiviseksi toiminnaksi. Jotta työkalua voidaan käyttää, täytyy strategian sisältö jakaa neljään eri näkökulmaan, joita ovat talous, asiakas, prosessit sekä oppiminen/uudistuminen. Jokainen näkökulma puretaan *strategisiin tavoitteisiin*, niiden saavuttamiseksi vaadittaviin *kriittisiin menestystekijöihin*, tärkeimpiin menestystekijöitä mittaaviin *avainmittareihin* sekä *toimintasuunnitelmiin*,

joiden avulla mittareiden tavoitearvot voidaan saavuttaa. Kantavana ajatuksena työkalun käytölle ovat osa-alueiden välisten sekä sisäisten kausaalisuhteiden miettiminen. [12.s.52-53]

Mittaristossa on hyvä olla mukana kahdenlaisia eri mittareita. Ajurimittareihin työntekijät voivat vaikuttaa suoraan omalla toiminnallaan ja seurantamittarit ovat tarkoitettu johdolle kertomaan muutoksen suunnasta. [12.s.53]

Strategiakartta

Tasapainotetun mittariston neliulotteiseen tarkasteluun perustuvien strategiakarttojen tarkoituksena on avata tarvittavia toimintamalleja, jotta organisaatio voi saavuttaa tavoitteensa. Mallin ideana on havainnollistaa toimintojen suhteita menestykseen purkamalla ylätason strategiset tavoitteet ja visio syyseuraus suhteina alas aina käytännön toimiksi saakka. Laaditun strategiakartan tulee mahtua yhdelle A4-paperille, ja sen tulee olla selkeä ”tarina”, jossa kuvataan kuinka organisaatio saavuttaa menestystä pitkällä aikavälillä. [12.s.82]

Ensimmäinen askel strategiakartan laadintaa varten on tiedostaa organisaation visio. Vision alapuolelle strategiakartta rakennetaan neljälle eri tasolle, joista ylimpänä on taloudellinen näkökulma. Taloudellinen menestys voidaan saavuttaa asiakkaan avulla, joten asiakasnäkökulma on sijoitettuna suoraan taloudellisen perspektiivin alapuolelle. Prosessitaso on sijoitettuna asiakasnäkökulman alapuolelle, koska sen avulla voidaan määritellä toimet, joiden avulla asiakasulottuvuuden tavoitteet voidaan saavuttaa. Alimmalla tasolla kuvataan mitä on osattava ja opittava, jotta prosessit saadaan toimimaan halutulla tavalla. [12.s.82]

Näistä neljästä eri tasosta tärkeimpänä voidaan pitää oppimisen ja kasvun tasoa, koska strategiakarttojen perimmäinen idea on juuri konkretisoida tätä vaikeasti mitattavaa pääomaa. Tällä tasolla pyritään havainnollistamaan ne aineettomat pääomat, jotka on saatava tuottamaan arvoa. Sisäisen prosessin näkökulmalla määritellään ne prosessit, joilla aineeton pääoma muutetaan asiakkaalle näkyviksi tuloksiksi. Asiakasnäkökulmalla puolestaan pyritään havainnoimaan, kuinka asiakas saa mahdollisimman paljon lisäarvoa. Lopuksi taloudellinen näkökulma määrittelee, kuinka aineeton omaisuus muuttuu aineelliseksi arvoksi. [12.s.83-85]

Koska strategiakartan on hyvä sopia yhdelle A4 paperille, kannattaa siihen valita teemoja rajallisesti, ja strategian kommunikoinnin tulee pysyä yksinkertaisena ja selkeänä. Hyvässä kartassa jokainen ulottuvuus sisältää muutaman teeman, jotka sittemmin avataan ja käydään läpi henkilökunnalle järjestetyissä koulutuksissa. Strategian toimeenpano edellyttää sisällön purkamista hankkeiksi ja työntekijöiden vastuiksi. [12.s.85]

Ei Julkinen

1. Testausstrategia

1.1. Kuvaile omin sanoin yrityksen ohjelmistotestausstrategia?
1.2. Millaiseksi koet oman osasi strategisten tavoitteiden saavuttamiseksi?
1.3. Mitkä ovat ohjelmistotestauksen tämänhetkiset vahvuudet yrityksessä?
1.4. Mitkä ovat ohjelmistotestauksen tämänhetkiset heikkoudet yrityksessä?
1.5. Onko asioita, jotka voivat hankaloittaa ohjelmistotestausta tulevaisuudessa?
1.6. Onko asioita, jotka tietyin toimenpitein toisivat merkittävää lisäarvoa testaukselle?

2. Testauksen integroituminen kehitystyöhön

2.1. Millaiseksi koet testauksen roolin osana kehitystyötä tällä hetkellä?
2.2. Mikä on näkemyksesi testaus-, sekä sen suunnitteluvastuun jakautumisesta tiimissä?
2.3. Kuinka testausnäkökulma on mielestäsi huomioitu suunniteltaessa uuden ominaisuuden toteutusta ohjelmaan?
2.4. Kuvaile ohjelmistokehityksen ja testauksen välistä tasapainoa resursoinnissa?
2.5. Kuinka mielestäsi testaus tukee kehitystyötä tällä hetkellä, ja mitkä olisivat tärkeimmät asiat tuen parantamiseksi?

3. Käännös- ja Testausprosessin tuottama palaute

3.1. Millaiseksi koet mahdollisuutesi muodostaa realistinen käsitys ohjelmiston nykytilasta perustuen saatavilla olevaan tietoon?
3.2. Kuvaile, kuinka näkyvänä koet käännös- ja testausprosessin tuottaman palautteen ja tukeeko se työtäsi riittävästi.
3.3. Kuvaile, kuinka mielestäsi käännös- ja testausprosessin tuottamaan palautteeseen reagoidaan tällä hetkellä ja onko palaute asianmukaista.
3.4. Mikä olisi oman työsi kannalta tärkeintä testauksen tuottamaan palautteeseen liittyen?
3.5. Kehitysideoita testauksen tuottamaan palautteeseen liittyen?

4. Muut asiat

4.1. Kuinka testaus tulisi dokumentoida, jotta se tukisi paremmin omaa työtäsi?
4.2. Muita kehitysideoita ohjelmistokehitysprosessin parantamiseksi?

Ei julkinen

Ei julkinen

Ei julkinen