TAMPEREEN TEKNILLINEN YLIOPISTO

JAAKKO JUUTILA

TESTIAUTOMAATION HYÖDYNTÄMINEN VERKKOPALVELUN KEHITYKSESSÄ

Diplomityö

Tarkastaja: professori Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty Tieto- ja sähkötekniikan tiedekunta-neuvoston kokouksessa 5. helmi-kuuta 2014

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
JUUTILA, JAAKKO: Testiautomaation hyödyntäminen verkkopalvelun kehityksessä
Diplomityö, 46 sivua
Helmikuu 2014
Pääaine: Ohjelmistotuotanto
Tarkastaja: Professori Hannu-Matti Järvinen
Avainsanat: Ohjelmistojen testaus, verkkopalvelut, käyttöliittymät, BDD, testiautomaatio

Nykyään yhä useampi uusi sovellus toteutetaan verkkopalveluna käyttäen joitain ketteriä menetelmiä. Triviaalia monimutkaisempien sovellusten tapauksessa testiautomaation käyttämistä sovelluksen kehityksen tukemisena voidaan pitää pakollisena. Tässä diplomityössä käydään läpi nykyaikaisten testiautomaatiomenetelmien, varsinkin JavaScript-pohjaisten verkkoselaimessa suoritettavien testien ja käyttäytymiskeskeisen testauksen käytännön soveltuvuutta ketteriä menetelmiä käyttävässä verkkopalvelun kehitysprojektissa. Työssä perehdytään tarkemmin erityyppisten automaattisten testien hyötyihin ja käyttökohteisiin.

Työn teoriaosuudessa kerrotaan muutamista yleisesti käytetyistä vaiheohjatuista ja ketteristä ohjelmistotuotannon menetelmistä ja minkälainen rooli ohjelmistotestaamisella niissä on. Tarkemmin käydään läpi ketterien menetelmien erilaisia testausmenetelmiä ja niiden hyötyjä ja käyttötapauksia. Teoriaosuudessa käydään läpi myös käyttäytymiskeskeisten liiketoimintatestien ominaisuuksia sekä tutustutaan testauksen automatisoinnin menetelmiin ja sen luomiin mahdollisuuksiin.

Seuraavaksi työn käytännön osuudessa esitellään esimerkkitapaus ei-triviaalista verkkopalvelun kehitysprojektista, jossa käytetään ketterää ohjelmistotuotannon menetelmää sekä usean eri abstraktiotason automaattisia testejä. Tässä kohdassa perehdytään erityisesti verkkoselaimessa suoritettavien automaattisten läpileikkaavien käyttöliittymätestien ominaisuuksiin ja haasteisiin.

Työn tuloksena saadaan selville mitä hyötyjä ja haasteita liittyy automaattisten käyttöliittymätestien käyttämiseen liittyy, sekä mitä tekniikoita ja työkaluja verkkoselainpohjaisen käyttöliittymän automaattisessa testaamisessa voidaan hyödyntää.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
JUUTILA, JAAKKO: Using test automation in web application development
Master of Science Thesis, 46 pages
February 2014
Major: Software Engineering
Examiner: Professor Hannu-Matti Järvinen
Keywords: User interface tests, BDD, test automation

Nowadays agile methods are commonly used when developing new applications. Using test automation to support software development can be considered as a mandatory practise. This master's thesis presents modern test automation methods, especially suitability of JavaScript based browser tests and behaviour driven tests in an agile web application development project. Different types of test automation and what possibilities it may raise are explored in detail.

The theory part of this thesis describes a few of the popular sequential design processes and agile software development methods. Testing in agile methods is explored in more detail. The theory part also introduces behaviour driven development method.

Next in the practical part of this thesis we present a case study of a non-trivial web application development project where agile methods and automatic tests from multiple abstraction levels are used. In this part we especially familiarize us with automatic web browser executable end-to-end user interface tests.

As a result, this study shows what benefits and challenges have to do with automatic user interface testing as well as what techniques and tools can be used in web browser based automatic testing.

ALKUSANAT

Tämä on diplomityö on tehty Reaktor Innovations Oy:ssä.

Kiitän ystäviäni ja työkavereitani siitä, että jaksoivat jatkuvasti kysellä diplomityön etenemisestä ja kannustaa sen loppuunsaattamisessa. Erityiskiitos Matti Vuorelle ja Hannu-Matti Järviselle työni ohjaamisesta.

SISÄLLYS

TERMIT JA NIIDEN MÄÄRITELMÄT

Ajax	Asynchronous JavaScript and XML on web-kehityksessä käytettävien teknologioiden joukko, jota käytetään asynkronisten asiakassovellusten web-sovellusten luomisessa.
Boilerplate-koodi	Koodia joka täytyy sisällyttää moneen paikkaan samanlaisena tai pienin muutoksin.
DOM	Document Object Model on puumainainen tietorakenne, joka esittää WWW-sivun HTML-sisällön.
HTML	Hypertext Markup Language on erityisesti verkkoselaimissa käytetty kieli, jolla voidaan kuvata hyperlinkkejä sisältävää tekstiä.
HTTP	Hypertext Transfer Protocol on sovellustason protokolla, joka on suunniteltu hajautettuihin hypermediajärjestelmiin.
JSON	JavaScript Object Notation on tiedonsiirtoon käytetty avoin standardiformaatti, joka käyttää ihmisluettavaa tekstiä avain-arvo-pareista koostuvien olioiden välittämiseen.
Lean-ajattelu	Lean on Toyotan autotuotannon periaatteista johdettu johtamis- ja tuotantoparadigma, jossa korostuvat osaaminen, asiakastarvelähtöisyys, tehokas tuotantoogistiikka ja joustavat tuotantomenetelmät ja –välineet sekä turhan työn välttäminen.
POSIX	Portable Operating System Interface on käyttöjärjestelmien välisen yhteensopivuuden ylläpitämiseen kohdistuva standardi.
Refaktorointi	Refaktorointi tarkoittaa prosessia, jossa tietokoneohjelman lähdekoodia muutetaan siten, että sen toiminnallisuus säilyy, mutta sen sisäinen rakenne paranee.
REST	Representational state transfer on tilaton arkkitehtuurityyli hajautettujen järjestelmien ohjelmoitavien rajapintojen luomiseen.
Roskienkeruu	Roskienkeruu on yksi automaattisen muistinhallinnan muoto, joka vapauttaa muistia käytöstä poistuneilta olioilta.
UML	Unified Modeling Language on graafinen mallinnuskieli, joka sisältää 14 erilaista kaaviota rakenteen, käyttäytymisen ja vuorovaikutuksen kuvaamiseen.
XML	Extensible Markup Language on merkintäkieli, jolla tiedon merkitys on kuvattavissa tiedon sekaan.
xUnit	xUnit tarkoittaa ohjelmistotestauskehyksiä, joiden arkkitehtuuri ja käytännöt noudattavat tiettyjä piirteitä.

1 JOHDANTO

Uusien ohjelmistojen toteutusteknologia on viime vuosina vaihtunut perinteisistä työpöytäsovelluksista verkkoselaimessa toimiviin verkkopalveluihin [38]. Tämän muutoksen on mahdollistanut verkkoselaimien kehittyminen ja verkkoselaimissa olevien JavaScript-moottoreiden suorituskyvyn parantuminen.

Suosittujen verkkopalveluiden odotetaan kehittyvän jatkuvasti, olevan luotettavia ja tarjoavan uusia toimintoja käyttäjilleen. Verkkopalveluja käytetään monilla eri verkkoselaimilla ja perinteisten tietokoneiden lisäksi muilla laitteilla, jotka vaihtelevat älypuhelimista televisioihin. Nämä seikat luovat haasteen verkkopalveluiden kehittäjille ja muille sidosryhmille, joiden tulee pystyä reagoimaan muuttuviin markkinoihin nopeasti ja luotettavasti, luomaan täysin uusia liiketoiminnallisia mahdollisuuksia ja tai jatkuvasti kehittämään yrityksen liiketoimintaprosessien tukityökaluja.

Tämä diplomityö tehtiin osana laajempaa ohjelmistoprojektia, jossa kehitetään asiakkaan ydinliiketoiminnan prosesseja ja muita järjestelmiä tukevaa työkalua. Projektin aikana asiakkaan organisaatiota opetettiin toimimaan yhteistyössä ketteriä ohjelmistotuotannon menetelmiä käyttävän kehitystiimin kanssa, jotta kehitettävä järjestelmä saataisiin käyttöön nopeasti ja muuttuviin liiketoimintavaatimuksiin pystyttäisiin reagoimaan joustavasti.

Ketterien menetelmien suosimiin työtapoihin kuuluu myös automaattisten testien käyttäminen ohjelmiston kehitystyön tukemisessa. Tässä työssä on tarkoitus analysoida miten automaattista testaamista hyödynnetään kompleksisen verkkopalvelun kehitysprojektissa. Työssä käsiteltävä käytännön osuus on rajattu koskemaan vain verkkopalvelun käyttöliittymän automaattista testaamista.

Luvussa 2 käydään läpi muutamien vaihejakoisten ja ketterien ohjelmistotuotannon menetelmien ominaisuuksia ja eroja. Luvussa 3 paneudutaan ketterissä menetelmissä käytettäviä testausmenetelmiä ja esitellään niiden käyttötapauksia ja ominaisuuksia. Luvussa 4 käydään läpi miten automaattisista testeistä saadaan irti eniten hyötyjä ja mitä hyviä käytäntöjä testausautomaatioon liittyy. Luvussa 5 keskitytään esittelemään Behaviour Driven Development -menetelmää noudattavia käyttäytymiskeskeisiä liiketoimintatestejä. Luvussa 5 perehdytään esitellään esimerkkitapaus miten verkkopalvelun käyttöliittymän automaattinen käyttäytymiskeskeinen testaaminen käytännössä tapahtuu ja mitä haasteita siinä on. Lopuksi luvussa 6 kootaan yhteen työn tulokset ja esitellään johtopäätökset.

2 OHJELMISTOTUOTANNON MENETELMÄT

Ohjelmistoprojektin päätavoitteena on toteuttaa asiakkaan vaatimuksia vastaava toimiva ohjelmisto. Ohjelmiston elinkaarella tarkoitetaan aikaa, joka kuluu ohjelmiston kehittämisen aloittamisesta sen poistamiseen käytöstä [22].

Kehitettävät ohjelmistot ovat usein hyvin monimutkaisia, joten siksi ohjelmiston elinkaaren hallintaan on kehitetty useita erilaisia menetelmiä, joiden tarkoituksena on ohjata ohjelmistotuotantoprosessia kohti toimivaa valmista ohjelmistoa. Tässä työssä tarkasteltavat ohjelmistotuotannon menetelmät ovat jaettu vaihejakomallisiin menetelmiin ja ketteriin menetelmiin.

2.1 Vaihejakomalliset menetelmät

Vaihejakomallisissa menetelmissä ohjelmiston kehitystyö tai koko elinkaari on nimensä mukaan jaettu eri vaiheisiin, jotka mahdollistavat ohjelmiston kehityksen etenemisen seuraamisen ja hallinnan. [22]. Seuraavissa luvuissa käsitellään tarkemmin vesiputousmallia ja V-mallia. Näiden lukujen pääasiallisena lähteenä on käytetty Haikalan ja Märijärven teosta Ohjelmistotuotanto [22].

2.1.1 Vesiputousmalli

Vesiputousmallia pidetään tavallisimpana vaihejakomallina. Yleensä vesiputousmallia noudattavissa ohjelmistoprojekteissa on käytössä ainakin määrittely-, suunnittelu- ja toteutusvaiheet. Määrittelyvaihetta usein edeltää esitutkimusvaihe. Muita mahdollisia vaiheita ovat integrointi-, testaus-, käyttöönotto- ja ylläpitovaihe. Käytännössä eitriviaaleja järjestelmiä on tehty jo vuosia jollain tapaa syklisellä menetelmällä, jossa vesiputous toistuu useita kertoja.

Esitutkimusvaiheessa on tarkoitus löytää kehitettävän järjestelmän yleiset järjestelmätason vaatimukset, eli minkä ongelman järjestelmän tulisi ratkaista. Nämä vaatimukset määrittelevät asiakkaan tarpeet, mutta eivät ota kantaa toteutustapaan. Esitutkimus vastaa kysymykseen miksi järjestelmä kannattaa toteuttaa. Onnistuneen esitutkimusvaiheen suurimmat haasteet ovat asiakkaan kaikkien todellisten tarpeiden esille saaminen ja niiden ymmärtäminen.

Määrittelyvaiheessa analysoidaan asiakasvaatimuksia ja ne yhdistetään suunnittelun myötä ymmärrettyihin muihin vaatimuksiin. Niiden perusteella muodostetaan toteutettavan järjestelmän toiminnalliset vaatimukset ja ominaisuudet kuvaava määrittelydokumentti (toiminnallinen määrittely).

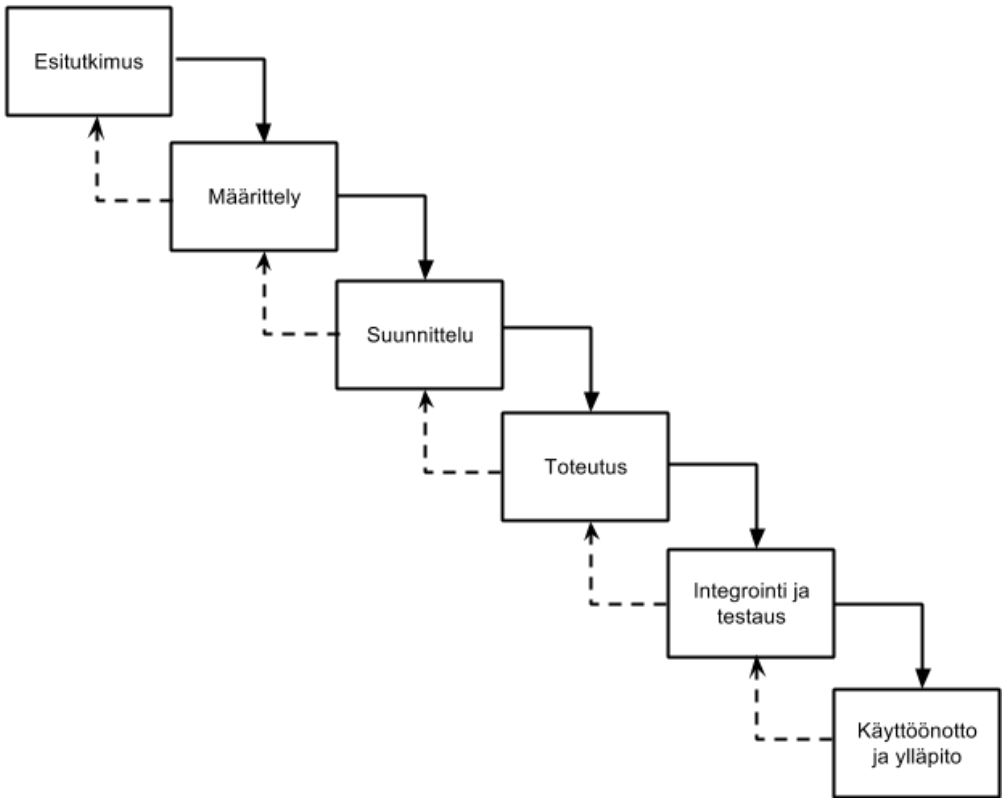
Suunnitteluvaiheessa kuvataan määrittelyvaiheessa muodostetut toiminnalliset vaatimukset järjestelmän teknisiksi vaatimuksiksi. Eli suunnitteluvaiheessa suunnitellaan, miten järjestelmä toteutetaan. Suunnitteluvaihe voidaan jaetaan usein arkkitehtuurisuunnitteluun ja moduulisuunnitteluun. Arkkitehtuurisuunnittelu tuottaa teknisen määrittelydokumentin. Moduulisuunnittelussa suunnitellaan jokaisen arkkitehtuurisuunnittelun määrittämän moduulin sisältö.

Toteutusvaiheessa suoritetaan järjestelmän toteuttaminen eli varsinainen ohjelmakoodin kirjoittaminen. Moduulit toteutetaan suunnitteluvaiheen määrittämien kuvausten perusteella.

Integrointi- ja testausvaiheessa on tarkoitus löytää toteutusvaiheessa tuotetusta ohjelmistosta mahdollisia virheitä ja moduulien välisiä yhteensopivuusongelmia. Testausvaiheen tuloksena syntyy testausraportti.

Käyttöönotto- ja ylläpitovaiheessa toteutettu järjestelmä otetaan käyttöön ja luovutetaan asiakkaalle. Ylläpitovaiheessa korjataan järjestelmän käyttämisen aikana havaittuja puutteita ja voidaan tehdään parannuksia toiminnallisuuksiin.

Jokaisen vaiheen tuottama välituote hyväksytään omassa katselmointitilaisuudessaan. Välituote voi koostua kokonaan dokumentaatiosta tai dokumentaatiosta ja koodista. Välituote on syöte seuraavalle vaiheelle ja eli se toimii lähtökohtana seuraavalle vaiheelle. Samalla välituotteet tarjoavat objektiivista näkyvyyttä projektin etenemiseen. Tällä tapaa luottamus projektin etenemiseen perustuu hyväksyttyihin lähtökohtiin. [4] Kuvassa 2.1 on esillä esimerkki vesiputousmallista, jossa on mukana kaikki edellä mainitut vaiheet.

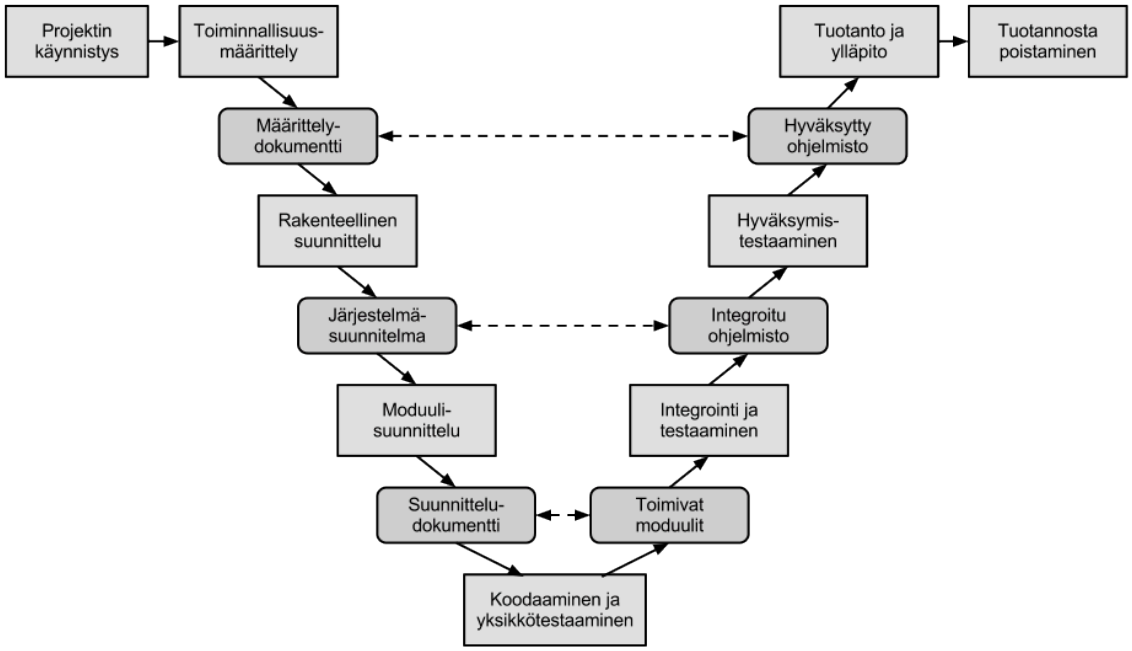


Kuva 2.1: Esimerkki vesiputousmallista [22]

2.1.2 V-malli

Perinteisestä vesiputousmallista periytyvä V-malli kuvaa yhdenlaisen vaihejakoisen ohjelmistokehitysprosessin tai elinkaarimallin ohjelmistoprojektin hallintaan [10]. Se poikkeaa vesiputousmallista siten, että siinä testaamisvaihe jaetaan moduulitestaamiseen, integraatiotestaamiseen ja hyväksymistestaamiseen. V-mallissa hyväksymistestaamisen suunnittelu tehdään ainakin osittain järjestelmän määrittelyn yhteydessä ja järjestelmän määrittelydokumenttia käytetään varsinaisessa testaamisessa. Vastaavasti integraatiotestaus suunnitellaan arkkitehtuurisuunnittelun yhteydessä ja moduulitestaus moduulisuunnittelun yhteydessä. Moduulitestauksessa etsitään vikoja yksittäisistä moduuleista, integrointitestauksessa moduulien yhteistoiminnasta ja järjestelmätestauksessa koko järjestelmän toiminnoista ja suorituskyvystä.

Tämä prosessi on havainnollistettu V-kirjaimen muotoisena kaaviona kuvassa 2.2, jossa suorakaiteen muotoisen laatikot kuvaavat eri vaiheita ja ovaalin muotoiset laatikot kuvaavat niiden tuottamia välituotteita. [4]



Kuva 2.2: V-mallin vaiheet ja välituotteet [4]

Edellä kuvattu ohjelmiston valmistuksen eteneminen on kuitenkin ideaalitilanne, sillä siinä oletetaan, että testaamiselle jää projektin loppuvaiheissa tarpeeksi aikaa. Todellisuudessa vesiputousmallissa ja parannellussa V-mallissa vaiheet venyvät ja projektin loppuun suunnitellut testaamisvaiheet puristuvat kasaan. [7]

2.2 Ketterät menetelmät

Ketterät menetelmät poikkeavat edellisessä luvussa käsitellyistä perinteisistä vaihejakomallisista menetelmistä monella tapaa. Ketterissä menetelmien yleisiin periaatteisiin

kuuluu, että niissä pyritään panostamaan muutoksiin reagoimiseen ja toimivaan ohjelmistoon sen sijaan, että vain tiukasti noudatettaisiin suunnitelmaa ja tuotettaisiin mahdollisimman kattavia dokumentteja. Ketterissä menetelmissä panostetaan yhteistyöhön asiakkaan kanssa ja keskitytään yksilöihin sen sijaan, että sopimuksissa määriteltäisiin tarkasti tehtävä työ ja käytettäisiin muodollisia prosesseja ohjaamaan työtä. [6]

Vaihejakomallisten projektien yksi ongelmakohta on, että työn tulokset näkyvät vasta projektin lopussa, jolloin vasta mahdolliset puutteet ja virheet paljastuvat. Ketterissä menetelmissä pyritään ehkäisemään vaihejakomallisissa projekteissa ilmeneviä ongelmia iteratiivisella ja inkrementaalisella kehitysmallilla, jolloin toteutettua toiminnallisuutta on mahdollista esitellä jo ennen järjestelmän käyttöönottoa. [7]

Vaihejakomallisessa projektissa toteutustyön syötteenä on sovelluskehittäjille annettava laaja määrittelydokumentti, jonka on tarkoitus sisältää järjestelmään toteutettavien ominaisuuksien kaikki yksityiskohdat. Ketteriä menetelmiä käyttävässä projektissa keskitytään mittavan dokumentaation ja muodollisten katselmointitilaisuuksien sijaan kommunikaatioon esimerkiksi siten, että asiakkaan edustajat ja kehitystiimi käyvät läpi yhdessä iteraation alussa käyttäjätarinoiden perusteella, miten seuraavaksi toteutettavan järjestelmän ominaisuuden tulisi toimia. Näin syntyvä ohjelmiston määrittely on asiakkaan ja kehitystiimin yhteisomistuksessa, eikä sitä vain ojenneta kehitystiimille toteutettavaksi. [7; 9]

Seuraavissa kohdissa tarkastellaan kahta yleisintä ketterää menetelmää, jotka ovat Scrum ja Kanban.

2.2.1 Scrum-menetelmä

Scrum on ketterien menetelmien periaatteita noudattava monimutkaisten tuotteiden kehittämiseen ja ylläpitoon tarkoitettu viitekehys. Scrum koostuu Scrum-tiimistä rooleineen, tapahtumista, tuotoksista ja säännöistä. Tässä luvussa on pääasiallisena lähteenä käytetty Schwaber et al. teosta The Scrum Guide [32].

Scrum-tiimi on itseohjautuva ja se koostuu tuoteomistajasta, kehitystiimistä ja scrummasterista. Tuoteomistaja on asiakkaan edustaja ja on vastuussa tuotteen arvon ja kehitystiimin työn arvon maksimoimisesta. Kehitystiimin jäsenet toteuttavat itseohjautuvasti kehitysjonossa olevat asiat julkaisukelpoiseksi tuoteversioksi. Kehitystiimin koko on yleensä noin 7 ± 2 henkilöä, jotta se pysyy ketteränä ja tehokkaana. Scrummasterin rooli on toimia Scrum-tiimin valmentajana ja varmistaa Scrumin käytäntöjen ja sääntöjen noudattaminen. Scrummasteri suojelee kehitystiimiä ulkopuolisilta häiriötekijöiltä, kuten yllättäviltä asiakasvaatimuksilta, ja auttaa Scrum-tiimin ulkopuolisia sidosryhmiä ymmärtämään miten Scrum-toimii. Scrummasteri toimii eräänlaisena projektipäällikkönä.

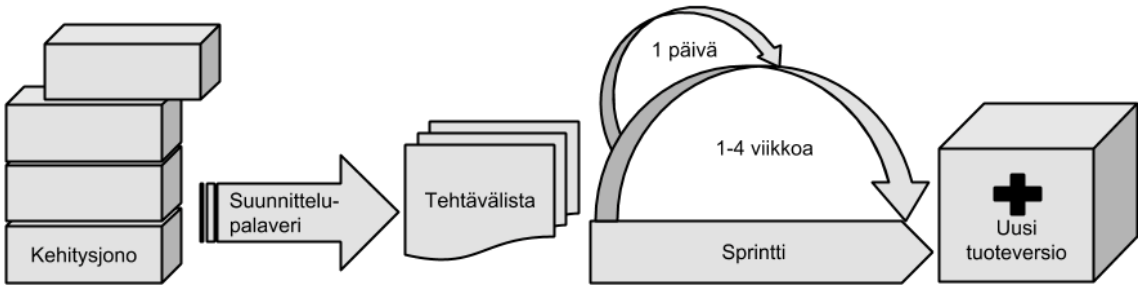
Kaikki Scrumiin kuuluvat tapahtumat ovat aikarajattuja, jotta suunnittelulle on riittävästi aikaa, mutta aikaa ei kuitenkaan käytetä liikaa. Tapahtumien ydin on sprintti, jonka kestoksi määritellään 1-4 viikkoa. Sprintin alussa koko Scrum-tiimi osallistuu 2-8 tunnin mittaiseen suunnittelupalaveriin, jossa valitaan kehitysjonosta sprintin aikana toteutettavat ominaisuudet, eli sprintin tehtävälista, ja suunnitellaan valitut tehtävät.

Sprintin aikana joka päivä pidetään enintään 15 minuutin mittainen päiväpalaveri, jossa kehitystiimin jäsenet kertovat edistymisestä ja mahdollisista ongelmista omissa työtehtävissään ja luovat suunnitelman, mitä aikovat päivän aikana tehdä. Sprintin lopussa pidetään sprinttikatselmus, jossa projektin sidosryhmille esitellään, mitä sprintin aikana on toteutettu.

Scrumin sprinttiä helposti kutsutaan ”pienoiskokoiseksi vesiputoukseksi”, tarkoittaen vesiputousmallin mukaista projektia. Johtopäätös on kuitenkin virheellinen monestakin syystä. Scrumissa suunnittelu, kehittäminen ja testaaminen ovat jatkuva aktiviteetti eikä kuten vesiputousmallia noudattavassa projektissa, jossa nämä toiminnot tehdään peräkkäin omissa vaiheissaan. Vesiputousmallissa määrittelyn jälkeen ilmenevät muutokset menevät jäykän muutostenhallintaprosessin läpi, mutta Scrumissa sprintin aikana sprintin tehtävälistalle valitut kohdat eivät vaihdu, vaan uudet muutospyynnöt laitetaan kehitysjonoon ja ne priorisoidaan. Vesiputousmallissa aikataulun venyessä aikamääreitä jatketaan, mutta Scrumissa sprintit ovat aikarajattuja. [34]

Scrumissa julkaisukelpoisen tuotteen jälkeen merkittävin tuotos on tuotteen kehitysjono, joka on tärkeysjärjestykseen järjestetty lista tuotteeseen kohdistuvista vaatimuksista. Tuoteomistajan tehtäviin kuuluu tuotteen kehitysjonon kohtien priorisointi niiden arvon, riskin ja työmäärän perusteella. Tuotteen kehitysjono ole koskaan valmis vaan se kehittyy koko ajan projektin edetessä ja järjestelmän uusien vaatimuksien paljastuessa.

Scrumin tärkeimmät tapahtumat ja tuotokset ovat havainnollistettu kuvassa 2.3.



Kuva 2.3: Scrum-menetelmän tärkeimmät tapahtumat ja tuotokset

2.2.2 Kanban-menetelmä

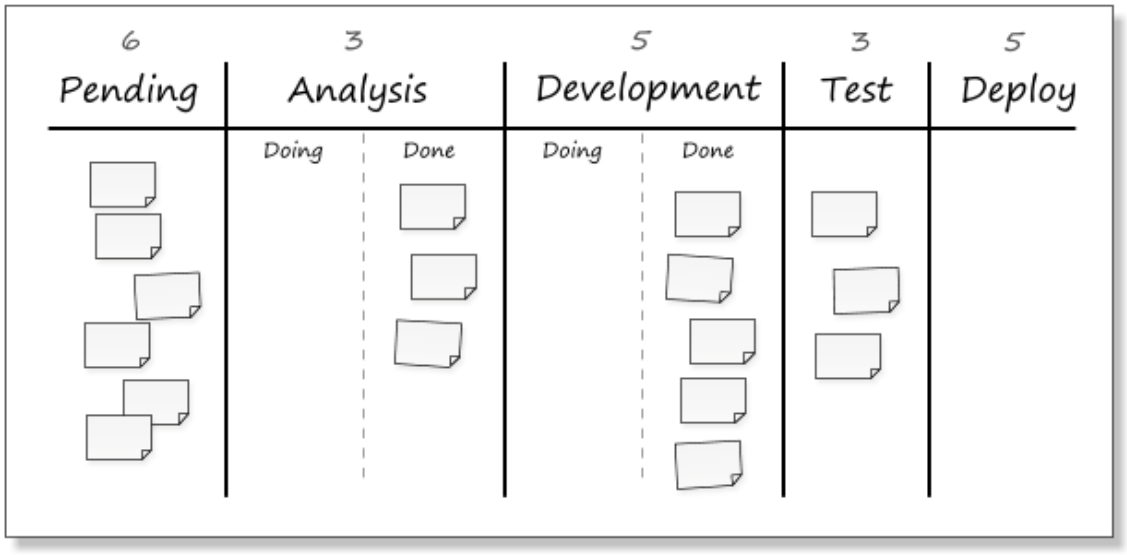
Kanban on menetelmä tietotyön hallintaan. Kanban ei varsinaisesti ole ohjelmiston elinkaaren hallintametodologia eikä se ole lähestymistapa projektinhallintaan. Se vaatii, että jokin prosessi on käytössä, jotta Kanbania voidaan soveltaa perustana olevaan prosessiin. [41]

Kanban perustuu pull-tyyppiseen mekanismiin, jossa uusia töitä otetaan tehtäväksi vasta kun siihen on kapasiteettia sen sijaan, että töitä työnnettäisiin tehtäväksi tarpeen mukaan. Kanban pyrkii kehittämään sitä harjoittavaa organisaatiota enemmän Lean-ajattelun tyyppiseen toimintatapaan. Kanban sisältää viisi ydinkäytäntöä, joista organisaatiossa muodostuu Lean-ajattelun tyyppisten käytäntöjen joukko: [41]

- visualisoiminen
- työmäärän rajoittaminen
- työkulkujen hallinta ja mittaaminen
- eksplisiittiset prosessin käytännöt ja
- mallien käyttäminen kehityskohteiden tunnistamiseen.

Ohjelmistojen kehittämiseen liittyvän työn eteneminen on yleensä luonnostaan näkymätöntä, jonka takia edistystä täytyy visualisoida jollain keinoin. Yleinen käytäntö Kanban-menelmässä on, että yleensä mallinnetaan työ, eikä esimerkiksi työntekijöitä, toimintoja tai työn luovuttamista toimintojen välillä [41].

Yleensä työ visualisoidaan mallintamalla työvaiheet isolla valkotaululla työvaihesarakkeiksi. Valkotaululla olevat laput kuvaavat yksittäisiä työtehtäviä ja liikkuvat vasemmalta oikealle. Kuvassa 2.4 on esimerkki Kanban-taulusta, jossa työvaiheet on jaettu viiteen osaan [23].



Kuva 2.4. Esimerkki Kanban-taulusta [23]

Työvaihesarakkeet on mahdollista jakaa tekeillä- ja tehty-sarakkeisiin, kuten kuvassa 2.4 on tehty Analysis- ja Development-sarakkeille. Työvaihesarakkeiden välissä voi olla myös puskurisarakkeita, jossa työtehtävät odottavat esimerkiksi tuotantoasennusta. [41]

Jokaisessa työvaiheessa tekeillä olevien tehtävien määrä on rajoitettu sovittuun määrään, joka on esillä kunkin sarakkeen yläreunassa. Tekeillä olevien työtehtävien määrän rajoittaminen estää ylituotannon ja samalla rajoittamisen tarkoitus on myös paljastaa tuotantoprosessin pullonkauloja. Andersonin mukaan työmäärärajat pitäisi sopia yksimielisesti yhdessä sidosryhmien ja johtoryhmän kanssa. Tällöin paineen allakin kehitystiimin on helppo viitata yhdessä yksimielisesti sovittuihin sopimuksiin ja työmäärärajoista pystytään pitämään kiinni. [41]

Verrattuna edellä esiteltyyn Scrum-menetelmään Kanban-menetelmää käyttävässä kehitysprosessissa ei tarvitse käyttää aikarajattuja sprinttejä ja ominaisuuksien sekä vikakorjausten valmistuessa sovitaan tarvittavista tuoteversioiden julkaisuiden tekemisistä. Kanban eroaa Scrum-menetelmästä myös ottamalla projektijohdon mukaan kehitystiimin tekemiseen tarjoamalla läpinäkyvyyttä kehitysprosessiin, kun Scrumissa kehitystiimi pyritään eristämään ulkopuolisista keskeytyksistä. [32]

3 TESTAAMINEN KETTERISSÄ MENETELMISSÄ

Perinteisissä vaihejakomallisissa ohjelmistoprojekteissa ohjelmiston testaaminen tapahtuu usein vasta projektin loppuvaiheilla ennen tuotteen julkaisua. Usein projekteissa testaamista edeltävät vaiheet venyvät, jolloin testaamiselle ei jää niin paljoa aikaa kuin alun perin suunniteltiin. Tämä aiheuttaa sen, että tuotetta joudutaan kiireellä korjaamaan ennen julkaisun aikarajaa ja mahdollisesti tuote joudutaan julkaisemaan viallisena. Ketterissä menetelmissä testataan myös ennen jokaista tuotteen inkrementin julkaisemista, mutta kaikkea testaamista ei kuitenkaan jätetä tehtäväksi juuri ennen julkaisua, vaan testaamista pyritään siirtämään aikaisemmaksi integroimalla testaaminen osaksi jokaista iteraatiota.

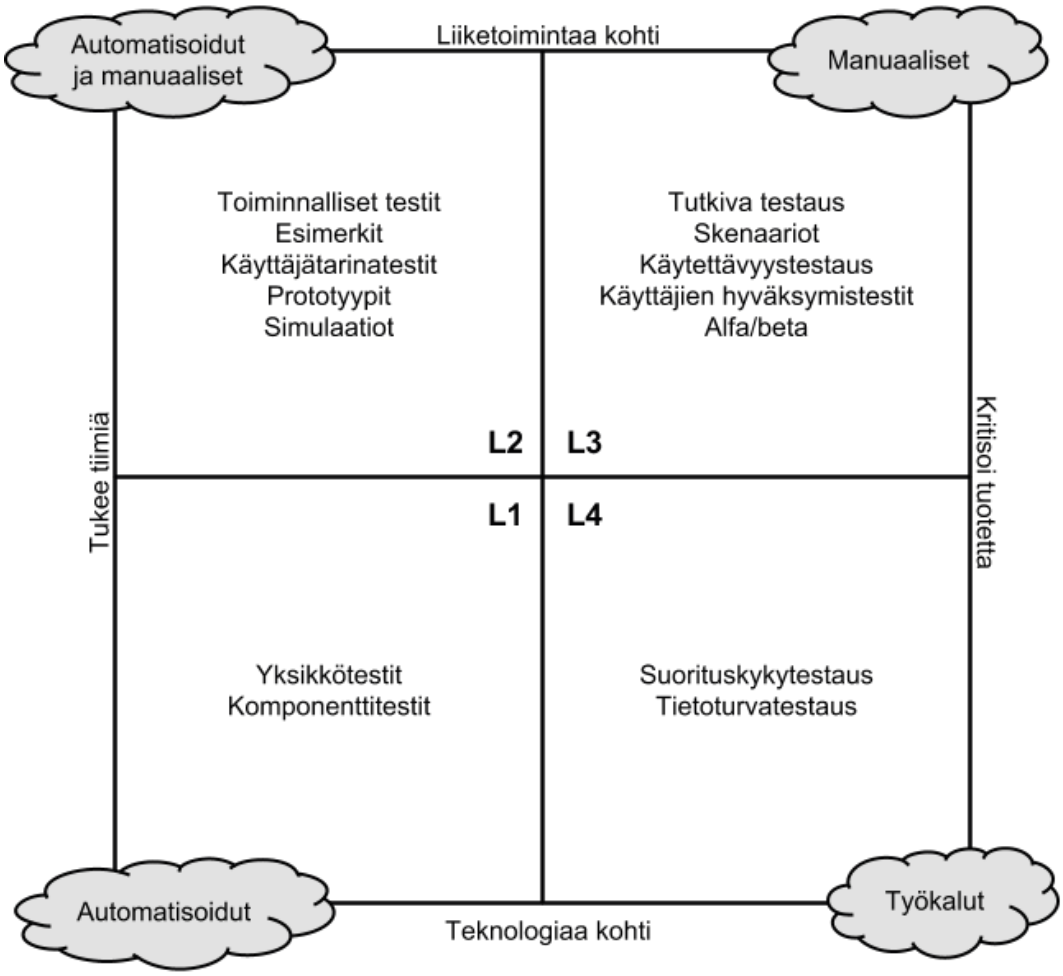
Ketterissä menetelmissä testitapauksia ei kirjoiteta mittavan määrittelydokumentin perusteella ennen kuin varsinainen järjestelmän toteutustyö on edes alkanut, vaan ne yleensä luodaan yleensä päiviä tai tunteja ennen kuin kyseinen toiminnallisuus toteutetaan. Toiminnallisuuden toteutustyön valmistuttua inkrementti testataan heti. Uusia toiminnallisuuksia ei todeta valmiiksi ennen kuin ne ovat testattu ja todettu toimiviksi. Tällä tavoin saadaan nopea palaute tehdystä työstä ja samalla saadaan valmiiksi pieniä järjestelmän palasia sen sijaan, että yritettäisiin tehdä iso kokonaisuus kerralla. [7; 9]

Suurin ero testaamisessa ketterien menetelmien ja perinteisten vaihejakoisten kehitysmenetelmien välillä on, että ketterissä menetelmissä testaaminen on koko tiimin tehtävä. Ei siis eritellä testaajia ja sovelluskehittäjiä eri tiimeihin, vaan laadukkaan ohjelmistotuotteen tuottaminen sellaisella aikajänteellä, että siitä on maksimaalinen hyöty asiakkaalle, on kehitystiimin jokaisen jäsenen vastuulla.

Tässä luvussa käydään läpi miten erilaisia testausmenetelmiä hyödynnetään ketterissä menetelmissä. Esiteltävät testausmenetelmät eivät ole sidoksissa mihinkään tiettyyn ketterään menetelmään, vaan ovat yleisesti käytössä. Luvun pääasiallisena on lähteenä käytetty Crispinin ja Gregoryn teosta Agile Testing: A Practical Guide for Testers and Agile Teams [7].

Ohjelmistoprojekteissa suoritetaan monentyyppistä testaamista monista eri syistä ja eri vaiheissa projektia. Crispin ja Gregory [7, s. 98] esittävät kirjassaan eri testausmenetelmien jaottelua neljään lohkoon menetelmän tarkoituksen ja testaamisen kohteen perusteella. Tämä jako on havainnollistettu seuraavalla sivulla kuvassa 3.1. Nelikentän yläpuoliskolla ovat liiketoimintaa kohti olevat testit ja nelikentän alapuoliskolla ovat teknologiaa kohti olevat testit. Nelikentän vasemmalla puoliskolla ovat kehitystiimiä tukevat testit ja oikealle puoliskolla on tuotetta arvioivat testit. Lohkojen järjestys ei

määrittele milloin mitäkin testejä tarvitaan. Tässä työssä keskitytään tarkastelemaan tarkemmin kehitystiimiä tukevia testejä eli lohkoja yksi ja kaksi.



Kuva 3.1: Testausmenetelmät jaettuna lohkoihin [7]

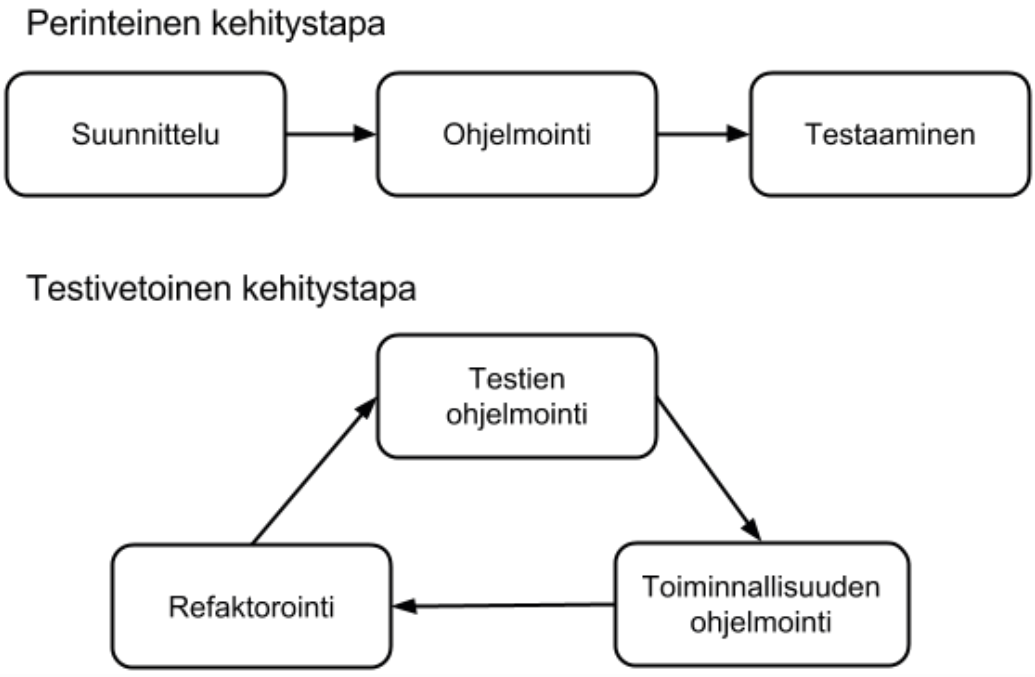
3.1 Kehitystiimiä tukevat teknologiaa kohti olevat testit

Lohkon yksi kehitystiimiä tukevat teknologiaa kohti olevat testit koostuvat yksikkötesteistä ja komponenttitesteistä, jotka parantavat ohjelmakoodin laatua auttamalla sovelluskehittäjää ymmärtämään mitä ohjelmakoodin pitää tehdä, ja tarjoamalla ohjausta oikeaan designiin. Lohkon yksi testit varmistavat järjestelmän sisäisen laadun, maksimoivat kehitystiimin tuottavuuden ja minimoivat teknisen velan syntymisen, eli ne auttavat tekemään tuotteen oikein.

Yksikkötestit varmistavat järjestelmän pienen palasen toiminnallisuuden, kuten yhden olion tai yhden funktion käyttäytymisen. Yksikkötestit tarkistavat reunatapauksia, kuten esimerkiksi tyhjiä merkkijonoja, vääränlaisia formaatteja ja erilaisia syötteiden yhdistelmiä. Yksikkötestien ei tulisi verifioida laajoja työnkulkuja, olla yhteydessä tietokantaan, vaatia konfigurointia tai tarvita jotain ulkopuolista järjestelmää. Komponent-

titestit puolestaan varmistavat järjestelmän jonkin hieman isomman kokonaisuuden käyttäytymisen, kuten eri olioiden välisen interaktion. [5; 7, katso 8]

Lohkon yksi testit luodaan testivetoisen kehityksen (test driven development, TDD) mukaisesti. Testivetoisessa kehityksessä ohjelmointi etenee kolmessa vaiheessa. Ensimmäisessä vaiheessa sovelluskehittäjä kirjoittaa kehityksen kohteena olevalle toiminnallisuudelle ensin epäonnistuvan yksikkötestin. Seuraavana kirjoitetaan vain riittävä määrä tuotantokoodia, jotta aiemmin kirjoitettu yksikkötesti voidaan suorittaa onnistuneesti. Kolmannessa vaiheessa sovelluskehittäjä refaktoroi tuotantokoodiaan siistimmäksi ja kierros alkaa alusta. Kolmas vaihe voidaan myös mieltää suunnitteluksi, sillä siinä pyritään parempaan järjestelmän rakenteeseen. Yhdessä nämä vaiheet muodostavat testaa-ohjelmoi-refaktoroi-syklin, jonka erot perinteisempään kehitystapaan on esitetty kuvassa 3.2. Testaa-ohjelmoi-refaktoroi-sykliä kutsutaan myös vaihtoehtoisesti punainen-vihreä-refaktorointi-nimellä, jossa punainen viittaa epäonnistuvaan testiin ja vihreä viittaa läpi menevään testiin. [2] Edellä mainitut testivetoisen kehityksen perusperiaatteet voisikin tiivistää yhteen yksinkertaiseen sääntöön: ”kirjoita tuotantokoodia vain korjataksesi epäonnistuvan testin”.



Kuva 3.2: Perinteisen kehitystavan ja testivetoisen kehitystavan eroavaisuudet

Testivetoinen kehitysmenetelmä vähentää vikojen selvittelyyn kuluvaa aikaa, koska siinä edetään toteutustyössä pienin askelein, jolloin tiedetään helposti, mitkä ohjelmakoodirivit aiheuttivat testien hajoamisen. Näin ollen virheet voidaan korjata aiemmin, joka vähentää järjestelmän kehityskustannuksia. [2] Virheiden löytäminen ja korjaaminen järjestelmän elinkaaren ylläpitovaiheessa on huomattavasti aktiivista toteutusvaihetta kalliimpaa [22]. Testivetoisen kehitysmenetelmän on myös todettu ohjaavan sovel-

luskehittäjiä kirjoittamaan pienempiä yksiköitä, jotka ovat vähemmän kompleksisia ja paremmin testattuja [33].

Testivetoista kehitysmenetelmää harjoittaessa tulee muistaa se, että kaikella ohjelmakoodilla on kehityskustannus ja ylläpitokustannus, myös testikoodilla. Tästä syystä sovelluskehittäjän tulee käyttää omaan kokemukseensa ja riskien arviointiin pohjautuvaa harkintaa mihin kaikkeen kannattaa yksikkötestejä kirjoittaa. Marickin mukaan sataprosenttiseen testikattavuuteen ei tule pyrkiä [37].

Yksikkötestien ja komponenttitestien suorittaminen tulisi olla nopeaa, koska testivetoisessa kehityksessä sovelluskehittäjät ajavat testit todella monia kertoja päivässä. Pitkä odottaminen palautteen saamiseksi testeistä saattaa myös hankaloittaa sovelluskehittäjien keskittymistä ohjelmoimiseen. Adzicin mukaan yksikkötestit ovat liian hitaita, kun yksikkötestiajo on kestoltaan lähellä kymmentä minuuttia [9, s. 163].

3.2 Kehitystiimiä tukevat liiketoimintaa kohti olevat testit

Kehitystiimiä tukevat liiketoimintaa kohti olevat testit kohdistuvat liiketoimintavaatimuksiin ja auttavat kehitystiimiä ymmärtämään järjestelmän kokonaiskuvaa. Tässä työssä käytetään lohkon kaksi korkean tason automatisoitavista testeistä termiä “liiketoimintatestit”. Lähdemateriaalissa näitä testejä kutsutaan monilla eri nimillä, kuten käyttäjätarinatestit, asiakastestit ja hyväksymistestit. Crispinin et al. mukaan termi “hyväksymistesti” on erityisen hämäävä, koska sillä voidaan myös tarkoittaa suorituskykytestausta ja tietoturvatestausta. Näitä liiketoimintatestejä ei pidä sekoittaa käyttäjien suorittamaan hyväksymistesteihin (user acceptance testing, UAT), jossa käyttäjät testaavat järjestelmän uudet ominaisuudet ennen niiden hyväksymistä.

Lohkon kaksi testien kirjoittaminen aloitetaan ennen varsinaisen ohjelmointityön aloittamista. Ohjelmistokehitystiimit toimivat parhaiten, kun sovelluskehittäjät ja asiakkaan edustajat kommunikoivat selkeästi keskenään. Tästä syystä kehitystiimi ja asiakas käyvät yhdessä työpajassa läpi toimintoja kuvaavia käyttäjätarinoita ja pyrkivät muodostamaan yhteisen käsityksen toiminnosta esimerkkien avulla. Näistä esimerkeistä muodostetaan automatisoitavia liiketoimintatestejä. Työpajojen toinen hyöty automaattisten liiketoimintatestien lisäksi on keskusteluissa asiakkaan ja kehittäjätiimin kesken jaettava toimialuetietämys. [16; 7]

Esimerkeistä ei pysty suoraan tekemään hyviä liiketoimintatestejä, vaan niistä pitää jalostaa esiin toiminnallisuuden kannalta olennainen tieto ja muuntaa se selkeäksi ja yksikäsitteiseksi määritelmäksi siitä, milloin toteutus on valmis. Näin muodostettujen liiketoimintatestien tulisi yksiselitteisesti määrittää vaadittu toiminnallisuus ottamatta kantaa toteutusyksityiskohtiin, jotta kehitystiimillä olisi mahdollisimman vapaat kädet löytää paras tekninen toteutustapa esitetylle toiminnallisuudelle [9].

Adzicin mukaan nykypäivänä on olemassa kaksi suosittua mallia liiketoimintatesteille: hyväksymistestauskeskeinen malli ja järjestelmän käyttäytymiskeskeinen malli [9]. Käyttäytymiskeskeisen testaamisen mallin ja hyväksymistestaamiskeskeisen mallin oleellisin ero on, että hyväksymistestaamiskeskeisessä mallin avulla pyritään muodos-

tamaan selkeämpiä tavoitteita sovelluskehittäjille ja käyttäytymiskeskeisessä mallissa pyritään rakentamaan jaettu ymmärrys toteutettavasta järjestelmästä sidosryhmien ja kehitystiimin välillä. Molempien mallien pitkäaikaisvaikutus on ehkäistä regressio-ongelmia. [9] Tässä työssä tarkastellaan tarkemmin vain käyttäytymiskeskeisen mallin, joka tunnetaan myös nimellä behaviour driven development (BDD), tapaa tehdä liiketoimintatestejä.

Korkean tason testit kattavat kerralla suuremman osan tuotantokoodista kuin yksittäiset yksikkötestit. Adzicin mukaan sovelluskehittäjillä on taipumus jättää yksikkötestejä toteuttamatta säästääkseen aikaa, koska korkean tason testit jo tarkistavat saman toiminnallisuuden. Epäonnistuneet liiketoimintatestit viestivät ongelmasta, mutta eivät osoita ongelman sijaintia yhtä tarkasti kuin yksikkötestit. Liiketoimintatestit eivät myöskään tarkista kaikkia puhtaasti teknisiä rajatapauksia. [5]

Automaattisilla liiketoimintatesteillä voidaan korvata V-mallin mukaiset perinteiset määrittely- ja suunnitteludokumentit. Ne kuitenkin poikkeavat perinteisistä dokumenteista siten, että ne eivät ole koskaan valmiita, koska ohjelmistotkaan eivät ole koskaan valmiita ja niitä jatkuvasti päivitetään muuttuvien yritystoiminnan vaatimusten mukaan. [5] Liiketoimintatestit kuitenkin ovat hyvä elävä dokumentaatio, sillä niiden oikeellisuus on aina varmistettavissa ajamalla testit. Jos testejä ei päivitetä järjestelmän muuttuessa, testien suorittaminen epäonnistuu. Automaattisilla liiketoimintatesteillä ei myöskään ole tarkoitus verifioida koko järjestelmää. Niillä on esimerkiksi hankala havaita käyttöliittymässä esiintyviä pieniä ongelmia. [5]

Automaattisten liiketoimintatestien päärooli on parantaa kommunikaatiota järjestelmän nykyisistä tai suunnitelluista ominaisuuksista, joka on yksi ketterien menetelmien kulmakivistä, sekä auttaa rakentamaan yhteinen käsitys siitä, miten järjestelmän tulisi toimia [5; 6]. Liiketoimintatestien tavoite on myös tunnistaa järjestelmästä korkean riskin osioita ja varmistaa, että niille tehdään ongelmakohtia vahvistava toteutus.

Vaikka liiketoimintatestit vähentävät riskejä ja regressio-ongelmia, niin myös muun tyyppiset testit ovat tarpeen. Riittävä suorituskyky, tietoturva, luotettavuus ja käytettävyys ovat myös riskejä, joita varten tulee kehittää testejä. Nämä testit kuuluvat lohkoihin kolme ja neljä.

3.3 Tuotetta arvioivat testit

Kuvan 3.1 nelikentän oikealla olevat puolella liiketoimintaa ja teknologiaa kohti olevat testit arvioivat tuotetta. Näiden testien avulla löydetään tuotteelle uusia vaatimuksia ja esimerkkejä, joita voidaan syöttää kohti prosessia, joka tukee kehitystiimiä ja ohjaa tuotteen kehitystyötä. Vaikka järjestelmää olisi kehitetty esimerkeistä luotujen liiketoimintatestien perusteella, se ei välttämättä siltikään vastaa asiakkaan toivomuksia tai jokin toiminnallisuus voi olla väärin määritelty. Asiakkaan antamat esimerkkikäyttötapaukset ovat voineet olla virheellisiä tai niiden määrittämät toiminnot tarpeettomia. Tuotetta arvioivat testit auttavat siis tekemään oikean tuotteen tunnistamalla vääriä toimintoja, virheellisiä konseptitason oletuksia ja puutteita molemmissa.

Lohkon kolme liiketoimintaa kohti olevat tuotetta arvioivat testit ovat manuaalisia testejä, joissa pyritään simuloimaan tapaa, jolla aito käyttäjä käyttäisi järjestelmää. Lohkoon kolme kuuluu tutkiva testaus, skenaariotestaaminen, käytettävyystestaus, käyttäjien suorittamat hyväksymistestit ja alfa/beta-testaus. Ne ovat manuaalista testaamista, jonka vain ihminen voi suorittaa. Testaajan apuna saattaa olla jotain apuskriptejä testidatan alustamiseksi, mutta varsinaiseen testaamiseen täytyy käyttää aivoja ja intuitiota, jotta selviää onko kehitystiimi toimittanut asiakkaan tarvitsemaa arvoa. Erilaisten liiketoimintatapahtumien ja oikeiden työnkulkujen tunteminen auttaa tekemään realistisempia testejä.

Käyttäjien suorittamat hyväksymistestit kuuluvat myös lohkoon kolme. Niissä käyttäjät ja asiakkaat testaavat tuotetta mahdollisimman hyvin tuotantokäyttöä vastaavissa olosuhteissa, eli esimerkiksi käyttäjän omalla työasemalla tuotantokäytöstä kopioidulla sisällöllä. Käyttäjän suorittamat hyväksymistestit antavat käyttäjille mahdollisuuden antaa palautetta uusista ominaisuuksista ja mahdollisesti esittää muutoksia niihin. [42]

Käytettävyystestaamisessa keskitytään eri tavoin löytämään testattavan järjestelmän käyttöliittymästä ongelmia tai parannuskohteita. Ennen järjestelmän toiminnon toteuttamista tai valmistumista voidaan käyttöliittymälle tehdä paperiprototypointia, jonka avulla voidaan kustannustehokkaasti ja nopeasti kokeilla käyttäjien kanssa erilaisia käyttöliittymäratkaisuja. Pidemmälle suunnitellusta käyttöliittymästä voidaan tehdä esimerkiksi staattisia HTML-sivuja, joita on mahdollista käyttää esittelytarkoituksessa. Käyttöliittymän toteuttamisen jälkeen asiantuntija-arvioinneilla voidaan löytää karkeimpia käytettävyysongelmia käyttöliittymästä. Käyttäjän toimintaa käyttöliittymän parissa voidaan havainnoida käytettävyystesteissä, joissa suoritetaan hallituissa olosuhteissa jotain tehtävää, tai käyttäjän todellisessa käyttöympäristössä.

Tutkiva testaus on lohkon kolme olennainen osa. Tutkivassa testauksessa testaaja yhtäaikaisesti suunnittelee ja toteuttaa testejä käyttäen kriittistä ajattelua tulosten arvioinnissa. Tämä tarjoaa paljon paremman mahdollisuuden oppia järjestelmän toiminnasta kuin skriptatut testit. Tutkiva testaus yleensä paljastaa järjestelmän kriittisimmät virheet.

Kaikkea testaamista ei kannata tehdä järjestelmän käyttöliittymän kautta. Järjestelmän testaamiseen voi esimerkiksi hyödyntää sen tarjoamia ohjelmointirajapintoja, jolloin joidenkin toiminnallisten testitapausten suorittaminen on tehokkaampaa. Esimerkiksi verkkopalveluissa yleisesti käytössä olevaan HTTP JSON -ohjelmointirajapintaan on helppo lähettää pyyntöjä.

Lohkon neljä teknologiaa kohti olevat testit arvioivat tuotteen ei-toiminnallisia ominaisuuksia kuten suorituskykyä, robustiutta, ylläpidettävyyttä ja tietoturvallisuutta. Jotkut näistä ominaisuuksista saattavat olla jopa tärkeämpiä kuin tuotteen oikeat toiminnot. Esimerkiksi jos Internetissä toimivan verkkopalvelun sivunlatauksen vasteaika on yksi minuutti, niin käyttökokemus on täysin kelvoton ja verkkopalvelun tarjoamat ominaisuudet ovat asiakkaan kannalta merkityksettömiä. Myös suorituskykytestit tulisi suorittaa ympäristössä, joka on mahdollisimman lähellä tuotantokäyttöä.

Lohkon neljä testejä varten voidaan kirjoittaa teknisiä käyttäjätarinoita, jotka voivat määritellä esimerkiksi missä ajassa järjestelmän tulee palauttaa käyttäjälle hakutulokset.

Toteutus näille käyttäjätarinoille voidaan tehdä samaan aikaan testattavan osuuden kanssa tai myöhäisemmässä vaiheessa.

4 TESTAUKSEN AUTOMATISOINTI

Ketteriä menetelmiä käyttävissä projekteissa pyritään tuottamaan asiakkaalle arvoa jatkuvasti, jolloin testiautomaatioon laitettavan panoksen täytyy myös olla tuottoisa. Hüttermannin mukaan testien automatisointi on edellytys lyhyelle julkaisusyklille, aikaisen palautteen saamiselle ja korkealaatuisen ohjelmistotuotteen luomiselle [35]. Näiden ominaisuuksien perusteella voidaan nähdä testiautomaatioon sijoitetun panoksen tuottavan merkittävää arvoa asiakkaalle.

Luvussa 3 testausmenetelmät jaettiin neljään lohkoon menetelmän tarkoituksen ja testauksen kohteen perusteella. Näistä esitellyistä lohkoista molemmat tiimiä tukevat lohkot, eli lohkot yksi ja kaksi, käyttävät automaattisia testejä. Lohkot auttavat tunnistamaan mitä testausmenetelmiä voidaan käyttää, mutta tarkempi testaamisen automatisoinnin strategia on välttämätön, koska testiautomaation toteuttamiselle on niin paljon erilaisia vaihtoehtoja.

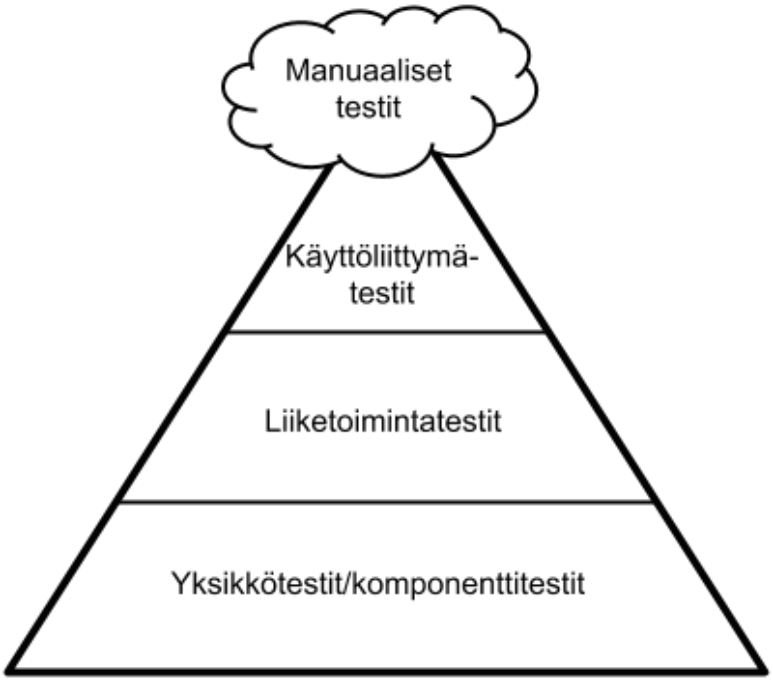
Jos projektissa ei ole vielä minkäänlaista testiautomaatiota, voi alkuun pääseminen olla haastavaa. Ensin täytyy ymmärtää, minkä ongelman haluaa ratkaista ja mitä yrittää automatisoida. Crispin et al. ehdottaa, että ensimmäinen panostus testiautomaation tulisi kohdistua suurimpaan kipukohtaan [7].

Testien automatisointiin kannattaa käyttää samanlaista lähestymistapaa kuin ohjelmakoodin kirjoittamiseenkin. Ensimmäisenä kannattaa pyrkiä siihen, että saa yhden kapean kaistaleen toimimaan kunnolla ja sitten siirtyä eteenpäin. Testien automatisoinnissa kannattaa pyrkiä hyödyntämään koko kehitystiimin osaamista, jolloin koko kehitystiimi sitoutuu varhaisessa vaiheessa testiautomaation käyttämiseen.

Tässä luvussa käydään läpi testaamiseen automatisointiin liittyviä menetelmiä ja prosesseja. Tämän luvun pääasiallisena on lähteenä käytetty Crispin et al. teosta Agile Testing: A Practical Guide for Testers and Agile Teams [7].

4.1 Testauksen automatisoinnin strategia

Automatisoitavat testit voidaan kategorisoida aiemmin esitellyn lohkojaon lisäksi myös seuraavalla sivulla kuvassa 4.1 olevan ”testiautomaatiopyramidin” tavoin. Tätä jakoa voidaan käyttää ohjaamaan kehitystiimiä kohti tehokasta ja lisäarvoa tuottavaa automatisoitua testausta. Testiautomaatiopyramidi sisältää kolme erilaista automatisoitujen testien kerrosta.



Kuva 4.1: Testiautomaatiopyramidi [7] soveltaen

Alimmaisella kerroksella on valtaosa automaattisista testeistä ja se on kaikkien automaattitestien perusta, joka tukee kaikkia muita kerroksia. Se koostuu enimmäkseen vakaista eristyksissä ajettavista nopeista yksikkötesteistä ja komponenttitesteistä, eli kehitystiimiä tukevista teknologiaa vasten olevista testeistä. Nämä testit auttavat tuottamaan korkealaatuista koodia, tarjoavat nopeimman palautteen ja ovat halvimpia kirjoittaa. Näin ollen niillä on myös selvästi suurin panostuksesta saatava hyöty. Myös muun tyyppiset teknologiaa vasten olevat testit, kuten suorituskykytestit kuuluvat tälle kerrokselle.

Keskimmäinen kerros sisältää suurimman osan automatisoiduista liiketoimintaa vasten olevista testeistä. Nämä testit ovat toiminnallisia testejä ja varmistavat, että rakennetaan oikeaa tuotetta. Tälle kerrokselle kuuluvat liiketoimintatestit ja yleisesti ne testit, jotka kattavat enemmän toiminnallisuutta kerralla kuin yksikkötestit. Nämä testit pyritään kirjoittamaan toimialuekohtaisella kielellä, joten ne vaativat hieman enemmän työtä kuin yksikkötason testit. Verkkopalvelun tapauksessa hyväksymistestejä on helppo ajaa esimerkiksi HTTP JSON -ohjelmointirajapintaa vasten.

Testiautomaatiopyramidin ylin kerros koostuu käyttöliittymää vasten ajettavista testeistä. Nämä testit yleensä tarjoavat pienimmän panostukseen suhteutetun hyödyn. Käyttöliittymätesteissä kannattaa keskittyä kattamaan vain ohjelmiston kriittisimmät suorituspolut. Järjestelmän jokaista suorituspolkua ei kannata käyttöliittymän kautta yrittää testata.

Käyttöliittymätestit ovat herkkiä hajoamaan. Esimerkiksi yhdenkin HTML-elementin ulkoasuun liittyvän class-attribuutin muuttaminen saattaa rikkoa jonkin käyttöliittymätestin. Käyttöliittymän kautta suoritettavat testit ovat myös huomattavasti hitaampia, kuin esimerkiksi alemmilla kerroksilla suoraan tuotantokoodia vasten suoritet-

tavat yksikkötestit. Automaattisiin käyttöliittymätesteihin liittyvistä haasteista huolimatta käyttöliittymätestejä ei tulisi sivuuttaa, sillä niitä tarvitaan jos halutaan välttyä käyttöliittymän regressioilta.

Esimerkiksi käyttöliittymän ulkoasun toimivuutta on vaikea tai jopa mahdotonta testata tehokkaasti automaattisilla testeillä. Automaattisten testien lukumäärästä riippumatta aina tarvitaan myös manuaalisia testejä, kuten tutkivaa testausta ja käyttäjien suorittamaa hyväksymistestaamista. Nämä testit on esitetty kuvassa 4.1 pienenä pilvenä pyramidin yläpuolella. Kuitenkin suurin osa regressiotestaamisesta tulee olla automatisoitua, sillä muuten testaajilla menee kaikki aika regressioiden etsimiseen ja he eivät pysty tekemään tehokasta uusien ominaisuuksien manuaalista testaamista.

4.2 Jatkuva integrointi

Jatkuva integrointi (continuous integration, CI) merkitsee ohjelmakoodin integroinnin suorittamista vähintään kerran päivässä. Tässä kontekstissa sana *jatkuva* tarkoittaa toistettavaa prosessia, joka tapahtuu säännöllisesti ja usein. Sana *integrointi* tarkoittaa, että erikseen kehitetyt ohjelmiston moduulit yhdistetään kokonaisuudeksi, käännetään, paketoidaan, testataan ja tarkistetaan. [35]

Kehitystiimillä tulisi olla käytössään jatkuva integrointi -palvelin. Duvall et al. mukaan jatkuva integrointi pienentää riskejä, vähentää toistuvia manuaalisia prosesseja, tuottaa käyttöönotettavan sovelluksen joka paikassa ja koska tahansa, parantaa näkyvyyttä projektiin ja vahvistaa kehitystiimin luottamusta kehitettävään tuotteeseen [35, katso 36].

Jatkuva integrointi -palvelimella voi olla ohjelmiston testaamisen ja paketoinnin lisäksi useita muitakin tehtäviä. Ohjelmistoa vasten voidaan suorittaa automaattisesti esimerkiksi staattisia analyyseja, jotka etsivät ohjelmakoodista virheitä suorittamatta sitä. Jatkuva integrointi -palvelin voi auttaa myös tiedon jakamisessa esittämällä ohjelmiston viimeisimmän integrointiajon tilan erillisellä radiaattorinäytöllä, joka on kehitystiimin ja muiden nähtävillä.

Yksi syy jatkuvan integroinnin tarjoamiin hyötyihin on, että se mahdollistaa virheiden havaitsemisen nopeammin. Mahdollisia syitä aiheutuneelle virheelle on vähemmän, kun kehitettävän järjestelmän komponenttien integrointi tehdään usein. Virheiden korjaaminen on helpompaa, kun tehty muutos on vielä sovelluskehittäjän tuoreessa muistissa. Hüttermannin mukaan integrointiin vaadittava vaivannäkö kasvaa eksponentiaalisesti suhteessa siihen kuinka paljon aikaa edelliseen integrointiin on kulunut. [35] Jatkuva integrointi myös varmistaa, että järjestelmä ei ainoastaan toimi oikein vain sovelluskehittäjän omassa paikallisessa ympäristössä [9].

Järjestelmän automaattisen rakennus- ja testausprosessin tulisi mahdollisimman nopea, jotta sovelluskehittäjien ei tarvitse kauaa odottaa palautetta muutoksien vaikutuksista. Crispinin ja Gregoryn mukaan useimpien tiimien mielestä yli 8-10 minuuttia kestävä prosessi on liian hidas, koska silloin palautteen saaminen kestää kauan. Jos testien suorittaminen kestää kauan, alkaa jatkuva integrointi -palvelimen työjonoon kasaantua

odottavia pyyntöjä järjestelmän testien ajamisesta ja sovelluskehittäjät joutuvat odottamaan palautetta entistä pidempään.

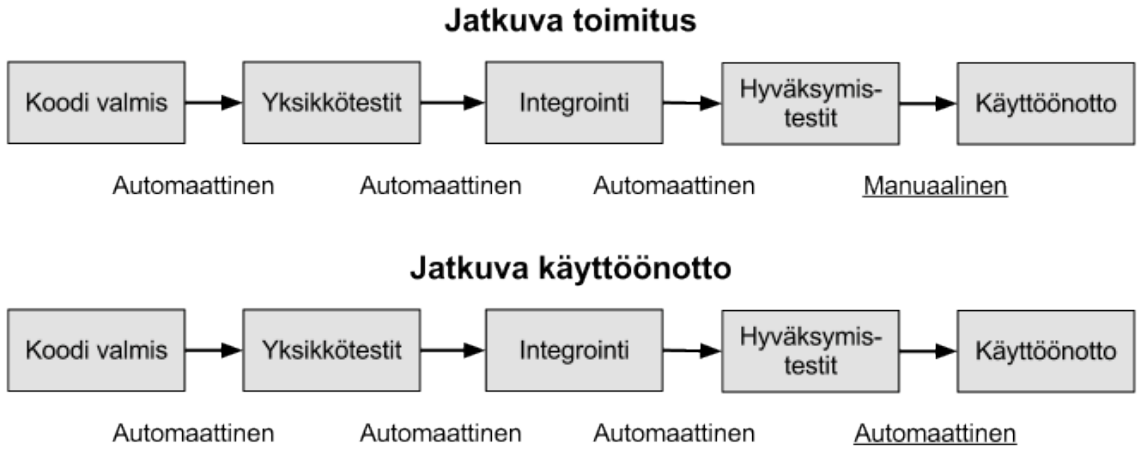
Usein hidas hitaat testiajot johtuvat siitä, että testit käsittelevät tietokantaa tai testaavat järjestelmää käyttöliittymän kautta. Jatkuva integrointi -prosessia tulisi nopeuttaa yksi pieni askel kerrallaan, jolloin pystytään mittaamaan mentiinkö muutoksella suorituskyvyssä oikeaan suuntaan.

Kuten yllä mainittiin, testattavaa järjestelmää vasten ajetaan useita eri tyyppisiä testejä. Monet näistä testeistä on mahdollista suorittaa rinnakkain ja kaikkia testejä ei ole pakko suorittaa tiettyjä järjestelmän versioita vasten. Esimerkiksi jos testaajat haluavat suorittaa tutkivaa testaamista, heille voi riittää järjestelmän versio, joka on läpäissyt automaattiset hyväksymistestit, mutta suorituskykytestejä ei ole ajettu. Jakamalla testit useisiin eri testikokoelmiin voidaan tarjota nopeampi tapa saada palautetta, kuin ajamalla joka kerta kaikki mahdolliset testit. [21]

4.3 Jatkuva toimitus ja jatkuva käyttöönotto

Järjestelmän uuden version käyttöönotto tulisi olla automatisoitu prosessi virheiden välttämiseksi. Järjestelmän automatisoituun käyttöönottoon pyrkivät menetelmät, kuten jatkuva toimitus (continuous delivery) ja jatkuva käyttöönotto (continuous deployment), perustuvat prosessiin, jossa hyväksytysti suoritettujen automaattisten testien jälkeen järjestelmä on valmis käyttöönottoa varten. Eli nämä prosessit ovat molemmat menetelmiä, joilla pyritään toimittamaan uusi versio nopeasti käyttäjien saataville. Loppukäyttäjän näkökulmasta ohjelmisto on saatavilla vain, jos se on otettu käyttöön oikeassa ympäristössä ja on täysin toimiva [35]. Molempien prosessien voidaan katsoa olevan laajennoksia jatkuvalle integroinnille.

Jatkuvan toimituksen ja jatkuvan käyttöönoton perustava ero on, että jatkuvassa toimituksessa prosessin lopussa järjestelmän uuden version käyttöönotto tuotantoon on manuaalinen ja jatkuvassa käyttöönotossa se tapahtuu aina automaattisesti prosessin päätteeksi [20]. Molemmat prosessit ovat esillä kuvassa 4.2.



Kuva 4.2: Jatkuvan toimittamisen ja jatkuvan käyttöönoton erovaisuudet [20]

Ohjelmiston testaukseen käytettävän ympäristön, eli esimerkiksi laitteiston ja käyttöjärjestelmän, tulisi olla mahdollisimman samankaltainen lopullisen tuotantoympäristön kanssa, jotta ympäristöstä aiheutuvat ongelmat paljastuvat aikaisemmin. On suositeltavaa käyttää samaa käyttöönottoskriptiä kaikkien ympäristöjen kanssa. Käyttöönottoskriptin tulisi tarkistaa oma tuotoksensa, eli esimerkiksi verkkopalvelun tapauksessa sen tulisi varmistaa, että käyttöönotettu ohjelmiston uusi versio todella käynnistyi ja on saatavilla. Käyttöönotto täytyy olla mahdollista ajamalla vain yksi skripti usean manuaalisen askeleen suorittamisen sijaan. [35]

Verkkopalveluiden ohjelmistopäivitykset näkyvät käyttäjille käytännössä heti, koska käyttäjien verkkoselaimeen tai laitteeseen ei tarvitse asentaa mitään vaan ohjelmisto on heti käytettävissä, kun käyttäjä siirtyy verkkopalveluun. Ohjelmistopäivitysten jakelutavan yksinkertaisuus helpottaa ketterissä menetelmissä käytössä olevan iteratiivisen ja inkrementaalisen ohjelmistokehityksen tavan käyttämisen verkkopalvelun toteutusprojektissa.

Manuaalinen testaaminen on edelleen erittäin tärkeässä roolissa, vaikka ohjelmiston käyttöönottoputki olisi täysin automatisoitu ja se sisältäisi kattavat automaattitestit. Manuaalisella testaamisella pyritään löytämään ne viat, jotka sovelluskehittäjältä, automaattitesteiltä ja muilta tarkastuksilta jäi huomaamatta. Automaattitestit pyrkivät osoittamaan, että ohjelmisto toimii edelleen odotetusti, mutta manuaalisessa testaamisessa pyritään löytämään uusia tilanteita, joissa ohjelmisto ei toimi oikein.

5 KÄYTTÄYTYMISKESKEISET LIIKETOIMINTATESTIT

Tässä työssä käydään läpi käyttäytymiskeskeisiä liiketoimintatestejä behaviour driven development (BDD) -menetelmän kautta. Menetelmän alunperin kehitti Dan North vakiintuneista ketterien menetelmien käytännöistä korjatakseen testivetoisessa kehityksessä (TDD) ilmenneitä ongelmia ja julkaisi sen artikkelissaan Introducing BDD [1]. Artikkelissa BDD kuvataan testivetoisen kehityksen muunnelmana, jossa pyritään määrittämään sovelluksen käytöstä sitä hyvin kuvaavasti nimettyjen testitapausten avulla.

Sovelluskehittäjät huomasivat olevansa hämmentyneitä käyttäessään testivetoista kehitystä projekteissaan, “ohjelmoijat halusivat tietää, mistä aloittaa, mitä testata ja mitä ei testata, miten paljon testata yhdellä kertaa, miksi kutsua heidän testejään, ja ymmärtää miksi testi epäonnistuu”, kertoo North. Tähän tarpeeseen North esittelee artikkelissaan, miten käyttäjätarinoista johdetaan automatisoitavia korkean abstraktiotason liiketoimintatestejä. Käyttäjätarinoilla ei yksinään ole liiketoiminta-arvoa, mutta ne tarjoavat yleiskäsityksen siitä, minkä toiminnon ominaisuus toteuttaa. Yleensä käyttäjätarinat kuvataan noudattaen seuraavaa muotoa: [1]

```
As a role
I want a feature
So that benefit
```

jossa Feature kuvaa jonkin halutun ominaisuuden, Benefit kuvaa ominaisuudesta koituvan hyödyn ja Role kuvaa ominaisuudesta hyötyvän käyttäjän. North paloittelee käyttäjätarinan sen perusosiin, jotka hän nimeää skenaarioiksi. Skenaariot noudattavat seuraavaa muotoa: [1]

```
Given some initial context,
when an event occurs,
then ensure some outcomes.
```

Skenaarioiden kieli ei ole sidottu englantiin, vaan sama voidaan esittää esimerkiksi suomen kielellä seuraavasti:

```
Annettuna jokin alkutilanne,
kun jotain tapahtuu,
niin varmista jotkut lopputulemat.
```

Skenaariossa ensimmäisenä (given/annettuna) esitellään alkutilanne tai konteksti, johon testattava järjestelmä alustetaan. Toisena määritellään jokin tapahtuma (when/kun), jonka vaikutuksia halutaan testata. Viimeisenä varmistetaan (then/niin), että testattavassa järjestelmässä tapahtuvat halutut asiat. Lyhyt esimerkkiskenaario [15]:

```
Scenario: Refunded items should be returned to stock
Given a customer previously bought a black sweater
And I currently have 3 black sweaters left in stock
When he returns the sweater for a refund
Then I should have 4 black sweaters in stock
```

Skenaarioiden kirjoittamisessa tulisi käyttää yhtenäistä kieltä asiakkaan edustajien käyttämien termien ja verbien kanssa. Yhtenäisen kielen käyttäminen skenaarioissa helpottaa asiakkaan edustajia kuvittelemaan millaisia ominaisuuksia he haluavat saada [7]. Ihmisluettavat ja yhtenäisen kielen skenaariot saattavat vähentää käyttäjien kouluttamisen tarvetta, koska käsitteet ovat jo tuttuja [19]. Väärinymmärrysten mahdollisuus pienenee ja eri osapuolten sanastomurteiden kääntämisestä aiheutuva kitka keskusteluissa poistuu, kun kehitystiimi käyttää tätä kieltä johdonmukaisesti heidän keskusteluissaan, dokumentaatiossa ja koodissa [13].

BDD:n määritelmän mukaan ohjelmakoodin tulisi olla osa järjestelmän dokumentaatiota ja järjestelmän määrittelyn tulisi olla osa ohjelmakoodia [3]. Edellä esitetyn tavan mukaisesti käyttäjätarinoista saadaan muodostettua automaattisesti todennettava ja testattava järjestelmän määrittely eli elävä dokumentaatio.

5.1 Työkalut

Käyttäytymiskeskeistä testaustapaa tukevia työkaluja löytyy yli 15 ohjelmointikielelle [14]. Solís et al. mukaan yleisimmin käytössä olevia BDD-työkaluja ovat Cucumber (Ruby), JBehave (Java), NBehave (C#), RSpec (Ruby), ja Specflow (C#) [3].

Skenaariot kirjoitetaan tavallisena tekstinä omiin tiedostoihinsa. Kuvaussäännöt tarjoavat tavan, jolla skenaariot yhdistetään testikoodiin. Yhdistämistavat vaihtelevat työkalujen kesken. Esimerkiksi JBehavessa yksi käyttäjätarina on yksi tiedosto, joka voi sisältää useita skenaarioita ja sen nimi viittaa käyttäjätarinaluokkaan. Jokainen skenaarion askel kuvataan käyttäjätarinaluokan testimetodissa, joka paikannetaan askeleen kuvaavan annotaation avulla. Ohjelmassa 5.1 on yksinkertaistettu esimerkki askeleen yhdistämisestä testikoodiin JBehavea käyttäen. [3]

```
public class RefundSteps {
  private Stock stock = new Stock();
  private Customer customer;

  @Given("a customer previously bought a black sweater")
  public void customerBoughBlackSweater() {
    customer = new Customer();
    customer.addShoppingHistory(new ShopItem("black sweater"));
  }

  @Given("I currently have $amount black sweaters left in
stock")
  public void setStockBlackSweaterAmount(int amount) {
    stock.setItemAmount("black sweater", amount);
  }

  @When("he returns the sweater for a refund")
  public void returnBoughtSweater() {
    customer.returnItem("black swater");
  }

  @Then("I should have $amount black sweaters in stock")
  public void blackSweaterStockAmountShouldBe(int amount) {
    ensureThat(stock.getItemAmount("black sweater"),
equalTo(amount));
  }
}
```

Ohjelma 5.1: Yksinkertaistettu esimerkki askeleiden yhdistämisestä testikoodiin

Yleisesti käytössä oleva testaustyökalu Selenium ei yksinään ole BDD-työkalu, koska siinä ei ole skenaarioita eikä sen avulla ajettavista skripteistä muodostu elävää dokumentaatiota. Verkkopalvelujen testaamisessa Seleniumia voidaan käyttää jonkin BDD-työkalun kanssa automatisoimaan verkkoselaimen ohjaamista. [18]

Tällä hetkellä tarjolla olevien BDD-työkalujen käyttäminen yhteistyövälineenä asiakkaan ei-teknisten edustajien kanssa on vielä perustavaa laatua olevia ongelmia. Työkalujen tulee elää ohjelmakoodin lähellä ja kehittyä samaan tahtia ohjelmakoodin kanssa. Muuten BDD-skenaariot altistuvat samanlaisille järjestelmän evoluutioon liittyville ongelmille, joita esiintyy esimerkiksi kun UML-kaavioita yritetään synkronisoida ohjelmakoodin kanssa. Skenaarioiden tulee olla myös mukana versionhallinnassa, joka tuo skenaarioiden kirjoittamiseen mukaan konsepteja, joita ei-teknisten henkilöiden saattaa olla hankala hallita. Skenaarioita kuten kaikkea muutakin ohjelmakoodia, tulee jatkuvasti refaktoroida ja tämä on lähempänä sovelluskehittäjän työtä kuin tuotepäällikön. [35]

5.2 Skenaarioiden muodostaminen

Ominaisuuksien toteuttaminen ja testien kirjoittaminen kannattaa aloittaa sopivan kokoisesta kokonaisuudesta tai skenaariosta, jonka avulla voidaan testata järjestelmän yleisarkkitehtuurin toimivuus. Hyvä ensimmäinen testi kattaa ohuen siivun toiminnallisuudesta ja seuraa onnellista polkua päästä päähän. Onnellisella polulla tarkoitetaan skenaariota, jossa kaikki työvaiheet onnistuvat ilman virheitä. Poikkeustapauksia ei siis tarvitse tässä vaiheessa vielä ottaa huomioon. Tällaista ohuen sivun testiä voidaan kutsua nimellä "terässäie" tai "valojuovaluoti", koska se kulkee kokonaisen toiminnallisuuden alusta loppuun. Se varmistaa kaikkien järjestelmän komponenttien toimivuuden kytkemällä ne yhteen, jolloin järjestelmään voidaan luoda lisää toiminnallisuutta inkrementaalisesti. [7]

Onnellisen polun testin jälkeen kannattaa määritellä korkean riskin skenaariot. Niissä on epäsuotuisa lopputulos, jolla on myös suuri todennäköisyys tapahtua. Näitä testejä tilanteita voidaan etsiä kysymällä asiakkaan edustajalta kysymyksiä kuten "Mikä on pahin asia, joka voisi tapahtua?" tai kysymällä sovelluskehittäjiltä "Mitä jälkiehtoja tällä koodilla on? Mitä tietokantaan tulisi tallentaa?" Näillä testeillä pyritään minimoimaan tunnistettuja riskejä. [7]

Jotkut skenaariot saattavat vaatia monimutkaisia testattavan järjestelmän tilan konfigurointeja, sillä kaikkia olioita harvoin voi alustaa eristyksessä toisistaan. Kuitenkaan kaikkea alustamista ei kannata eksplisiittisesti tehdä osana skenaarion alustamista, sillä se saattaa tehdä skenaariosta vaikealukuisen ja hankaloittaa sen ymmärtämistä. Riippuvuudet, jotka eivät suoraan liity skenaarion tavoitteeseen, tulisi siirtää alemmille kerroksille. [9]

On olemassa kaksi toisistaan poikkeavaa tapaa ilmaista tietokoneelle käskyjä: imperatiivinen tyyli ja deklaratiivinen tyyli. Imperatiivisessa tyyli tarkoittaa sarjaa käskyistä tietokoneelle tietyssä järjestyksessä. Deklaratiivinen ohjelma kertoo tietokoneelle, mitä sen täytyisi tehdä kertomatta tarkemmin, miten se tulee tehdä. [16]

Imperatiivisella tyylillä kirjoitettujen skenaarioiden kieli näyttää geneeriseltä eikä siinä muodostu aihealueen omaa sanastoa tai elävän dokumentaation yhtenäistä kieltä. Skenaarioista tulee usein pitkiä, niissä on paljon kohinaa ja ne hajoavat helposti, koska ne sisältävät liikaa toteutusyksityiskohtia. [16]

Skenaarioiden kirjoittamiseen tulisi käyttää deklaratiivista tyyliä, koska skenaarioiden abstraktiotason noustessa tekniset termit vähenevät ja eri sidosryhmien jäsenet ymmärtävät paremmin käytettyä sanastoa. Samalla myös skenaarioiden kytkentä järjestelmän toteutusyksityiskohtiin vähenee ja ylläpidettävyys kehittyy. [16]

Skenaarioita kirjoittaessa tulisi välttää skenaarion tavoitteen kannalta turhien oheisyksityiskohtien lisäämistä. Esimerkiksi skenaario jossa kuvataan sähköpostiasiakasohjelmassa saapuneiden viestien tarkastaminen:

```
Given a user “Dave” With password “password”
And a user “Sue”
And an email to “Dave” from “Sue”
When I sign in as “Dave” with password “password”
Then I should see 1 email from “Sue” in my inbox
```

Käyttäjänimet helpottavat skenaarion tarinan kertomista, mutta salasanat ovat pelkästään kohinaa, sillä niillä ei ole mitään tekemistä varsinaisen testattavan asian kanssa. Nämä epäolennaiset yksityiskohdat vaikeuttavat skenaarion lukemista ja se puolestaan voi aiheuttaa sidosryhmien mielenkiinnon heikkenemisen skenaarioiden lukemiseen. [16]

6 KÄYTÄNNÖN SOVELLUS

Tässä luvussa paneudutaan diplomityön käytännön osuuteen. Ensin kerrotaan tuotekehitysprojektin kohteena olevasta järjestelmästä. Seuraavana perehdytään miten Kanban-menetelmää käytetään tuotekehitysprojektissa. Lopuksi käydään läpi miten tehdään automaattisia koko järjestelmän läpileikkaavia käyttäytymiskeskeisiä liiketoimintatestejä laajalle yhden sivun sovellus -tyyppisen verkkopalvelun käyttöliittymälle ja mitä hyötyjä haasteita siitä on.

6.1 Tuotteen kuvaus

Tämä työ tehtiin osana laajempaa tuotekehitysprojektia, jossa kehitetään uutta Master Data Management -järjestelmää korvaamaan asiakkaan vanhentunut järjestelmä. Ensimmäisessä vaiheessa molemmat järjestelmät toimivat rinnakkain, mutta projektin edetessä vanha järjestelmä poistetaan käytöstä suurimmalta osalta.

Master Datalla tarkoitetaan yrityksen liiketoiminnan kannalta avainasemassa olevaa tietoa, joka voi olla esimerkiksi tietoa tuotteista, asiakkaista tai toimittajista. Master Data tarjoaa koko organisaatiolle sovitunlaisen näkymän itsenäisiin liiketoiminnallisiin kokonaisuuksiin.

Master Data Management -järjestelmä puolestaan muodostaa joukon prosesseja, hallintoa, käytäntöjä, standardeja ja työkaluja, jotka yhdenmukaisesti määrittelevät ja hallinnoivat Master Dataa. Master Data Management -järjestelmän tehtävänä on tarjota prosesseja datan keräämiseen, koostamiseen, sovittamiseen, yhdistämiseen, laadunvarmistamiseen, tallentamiseen ja jakeluun koko organisaatiolle varmistaakseen datan yhtenäisyyden ja hallinnan tämän tiedon sovelluskohteissa. [24]

Järjestelmän sovellusympäristö koostuu yhdestä tai useammasta samanlaisesta palvelinsovellusinstanssista ja yhdestä jaetusta tietokantapalvelimesta. Palvelinsovellus tarjoaa HTTP-protokollan yli REST-tyyppisiä luku- ja kirjoitusrajapintoja sen sisältämään Master Dataan. REST-tyyppisellä ohjelmointirajapinnalla tarkoitetaan hajautettujen järjestelmien tilatonta ja yksinkertaista arkkitehtuurityyliä. Järjestelmä on integroitu lukuisiin muihin asiakasorganisaation käyttämiin järjestelmiin käyttäen edellä mainittuja ohjelmointirajapintoja, sekä joitain muita integrointimekanismeja. Palvelinsovellus on toteutettu käyttäen Scala-ohjelmointikieltä [43].

Asiakassovellus eli järjestelmän graafinen käyttöliittymä päätettiin toteuttaa yhden sivun sovelluksena (single-page application, SPA). Usein niiden tarkoitus on tarjota työpöytäsovelluksen kaltainen käyttökokemus ja suurin osa käyttöliittymän toimintalogiikasta suoritetaan verkkoselaimessa. Yhden sivun sovelluksissa verkkoselain hakee palvelimelta kerran tarvittavan määrän HTML-merkkausta sisältävän verkkosivun ja sen

mukana JavaScript-ohjelmointikielellä toteutetun verkkoselaimessa toimivan sovelluksen. Tämä sovellus noutaa taustalla tarvittaessa lisää resursseja palvelimelta käyttäjän toimien niin vaatiessa.

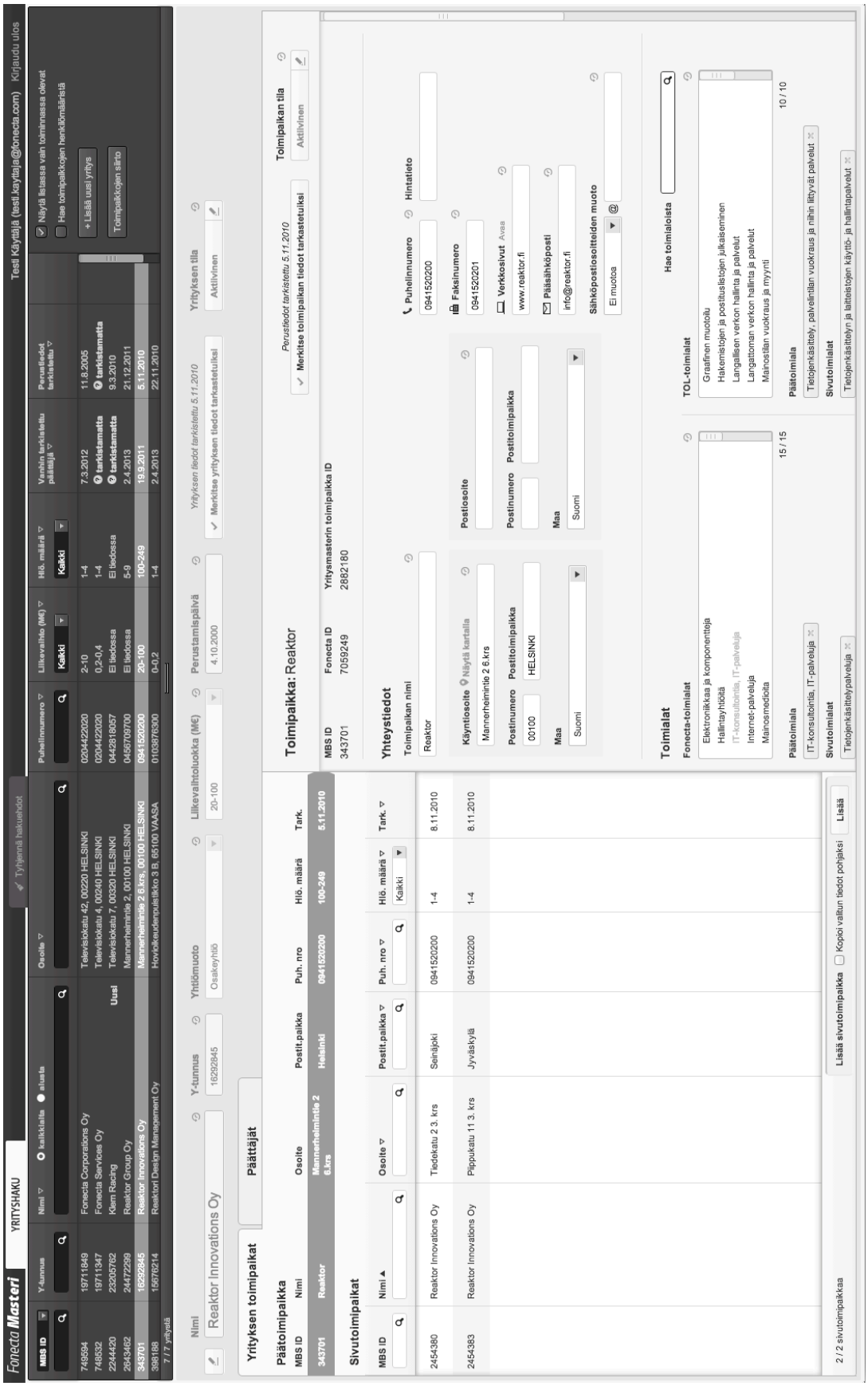
Esimerkiksi käyttäjän suorittaessa haun koko verkkosivua ei siis ladata uudelleen, vaan yhden sivun sovellus lähettää hakusanan käyttäen Ajax-tekniikkaa palvelimelle, joka palauttaa tarvittavat resurssit. Sovellus muuntaa tuloksena saadun datan haluttuun HTML-esitysmuotoon ja päivittää siihen liittyvät verkkosivun osat. Usein palvelimet tarjoavat REST-tyyppisen rajapinnan, joka palauttaa pyydetyn resurssin XML-formaatissa tai nykyään yhä useammin JSON-formaatissa. JSON-formaatti on tiedonsiirtoon tarkoitettu ihmisluettava avoin standardiformaatti, joka käyttää avain-arvopareja dataolioiden siirtämiseen.

Yhden sivun sovellukset muistuttava enemmän raskas asiakaspääte -mallia kuin kevyt asiakaspääte -mallia. Yhden sivun sovelluksissa on enemmän verkkoselaimessa suoritettavaa toimintalogiikkaa kuin perinteisissä verkkopalveluissa, joissa verkkosivujen HTML-merkkaus muodostetaan valmiiksi palvelimella ja verkkoselaimen tehtävänä on vain esittää haluttu sisältö [25].

Usein yhden sivun sovelluksia saatetaan kutsua myös nimellä HTML5-sovellus. HTML5-standardin verkkoselaintuen yleistymisen myötä yhden sivun sovelluksissa on ollut mahdollista hyödyntää HTML5n mukana tulleita uusia rajapintoja ja ominaisuuksia kuten esimerkiksi yhteydettömän tilan tiedontallentamista (offline storage), paikannusta ja multimediatukea. Aikaisemmin verkkoselaimiin piti asentaa liitännäisiä, kun esimerkiksi haluttiin toistaa videokuvaa verkkoselaimessa, mutta nykyään tuetut HTML5-teknologiat mahdollistavat videokuvan toistamisen ilman verkkoselaimeen asennettavia liitännäisiä.

Asiakassovelluksen toteutuskieleksi valittiin CoffeeScript [39]. CoffeeScriptillä kirjoitettua ohjelmakoodia ei suoraan suoriteta verkkoselaimessa vaan ensin se käännetään JavaScript-koodiksi, jota on mahdollista suorittaa verkkoselaimessa. CoffeeScript piilottaa ohjelmoijalta joitain JavaScriptin ”rumia” ominaisuuksia ja monisanaisuutta ja siten voi helpottaa ohjelmoijan työtä.

Seuraavalla sivulla kuvassa 6.1 on esillä kuvankaappaus asiakassovelluksen tämän työn kirjoitushetkisestä käyttöliittymästä. Asiakassovellusta kehitetään edelleen. Kuvasta voi havaita käyttöliittymän monimutkaisuuden ja laajuuden. Tämän työn kirjoitushetkellä asiakassovelluksen käyttöliittymässä on yli 100 toimintoa, kuten esimerkiksi muokatun datan automaattinen tallentaminen palvelimelle. Asiakassovelluksen toteutuksen monimutkaisuutta lisää toimintojen määrän lisäksi myös se, että useat käyttöliittymän toiminnot aiheuttavat muutoksia käyttöliittymässä useassa paikassa näkyvään dataan, joka täytyy pysyä ajan tasalla.



Kuva 6.1 Kuvankaappaus asiakassovelluksen käyttöliittymästä

6.2 Kehitysprosessi

Asiakassovelluksen käyttöliittymän suunnittelutyö oli aloitettu jo ennen varsinaista käyttöliittymän toteuttamisen aloittamista. Käyttöliittymän suunnittelutyön toteutti kaksi käyttöliittymäsuunnittelijaa yhteistyössä toteutettavan järjestelmän tuoteomistajan ja vanhan järjestelmän käyttäjien kanssa. Suunnittelutyössä käyttöliittymäsuunnittelijat testasivat käytettävyyttä käyttäen työvälineinään haastatteluja, havainnointia, paperiprototypointia sekä simuloituja käyttötapauksia. Tällä tavoin ennen toteutustyön aloittamista ja yhdenkään ohjelmakoodirivin kirjoittamista luotiin oikeiden käyttötapausten perusteella käyttöliittymän oikeat toiminnot kuvaavat rautalankamalli. Toteutustyön alkaessa käyttöliittymän rautalankamallien perusteella oli useimmista ominaisuuksista piirretty myös taittokuvat. Toteutustyön käynnistyttyä useiden lisäominaisuuksien suunnittelua jatkettiin samaan aikaan käyttöliittymän toteutuksen kanssa.

Järjestelmän käyttöliittymän toiminnallisuutta koskevissa kysymyksissä käyttöliittymäsuunnittelijat toimivat toimialueasiantuntijoina, koska heillä on erittäin laaja tietämys toimialueesta käyttöliittymäsuunnittelua varten tehtyjen haastattelujen takia ja ammattitaitoa luoda käytettävyydeltään hyvällä tasolla olevia käyttöliittymäratkaisuita. Osa järjestelmän käyttäjistä työskentelee lähellä kehitystiimiä ja on siten käytettävissä mahdollisten kysymysten noustessa esiin.

Järjestelmän kehitystiimin koko on vaihdellut projektin aikana 2-5 sovelluskehittäjässä. Kehitystiimin ei kuulu varsinaisia päätoimisia testaajia. Järjestelmän kehitystyöhön on lisäksi osallistunut edellä mainitut käyttöliittymäsuunnittelijat sekä yksi graafinen suunnittelija. Kehitystiimi käyttää kehitystyön ohjaamisprosessissa ketterää Kanban-menetelmää. Kehitystiimin soveltamassa Kanban-mallissa ominaisuudet ovat jaettu isolle ja pienelle taululle. Isolla taululla on näkyvillä priorisoituna tulevina viikkoina kehitettävät järjestelmän ominaisuudet. Pienellä taululla on tällä hetkellä tekeillä olevat ominaisuudet jaettuna pienempiin tehtäviin. Ison taulun tarkoitus on tarjota näkyvyyttä järjestelmän sidosryhmille tulevaisuudessa toteutettavista ominaisuuksista ja muutoksista. Pieni taulu auttaa kehitystiimiä seuraamaan jonkin ominaisuuden toteutuksen etenemistä ja kertoo minkä tehtävän parissa kukin kehitystiimin jäsen työskentelee.

Kun isolta taululta otetaan jokin ominaisuus työn alle se suunnitellaan tarkemmin ja jaetaan pienempiin tehtäviin. Ominaisuuteen liittyvistä tehtävistä kirjoitetaan tehtävälaput ja ne lisätään pienelle taululle. Tehtävälappujen suunnitteluvaiheessa käydään läpi toimialueasiantuntijoiden kanssa esimerkiksi käyttöliittymässä olevat työnkulut ja niiden reunatapaukset.

Kehitystiimin pieni taulu on esillä seuraavalla sivulla kuvassa 6.2. Pienellä taululla on tehtävälapuille viisi saraketta: ei aloitettu, työn alla, dev-ympäristössä, katselmoitu ja QA-ympäristössä. Työn alla -sarakkeesta tehtävälaput siirretään dev-ympäristössä-sarakkeeseen, kun tehtävään liittyvä ohjelmakoodi on tallennettu versionhallintaan. Dev-ympäristö on järjestelmän testausympäristö, johon järjestelmän uusin versio asennetaan automaattisesti jokaisen versiohallintaan saapuneen lisäyksen jälkeen. Kehitystiimin jonkun toisen sovelluskehittäjän suorittaman vapaamuotoisen katselmoinnin jäl-

keen tehtävälaput siirretään katselmoitusarakkeeseen. Ennen järjestelmän uuden version asentamista tuotantoympäristöön järjestelmän uusin versio asennetaan QA-ympäristöön, jossa käyttäjät testaavat järjestelmän uusia ominaisuuksia. QA-ympäristössä käytössä oleva tietokanta sisältää tuotantokannasta tuotua aitoa tuotantodataa, jotta testattaessa järjestelmän käyttäytyminen on mahdollisimman lähellä tuotantoympäristöä. Korjauksia ja uusia ominaisuuksia pyritään viemään tuotantoon viikoittain tai useammin, jos kriittiselle korjaukselle on tarvetta. Kuvassa 6.2 sarakkeiden vasemmalla puolella on pieniä ei-kiireellisiä tehtävälappuja ja viankorjaustehtävälappuja, joita voidaan ottaa työn alle, jos esimerkiksi uusien ominaisuuksien kehittäminen on jostain syystä estynyt väliaikaisesti.

Kuva 6.2. Kehitystiimin Kanban-taulu

6.3 Käyttöliittymätestien toteuttamisen lähtötilanne

Asiakassovelluksen suunnitellun käyttöliittymän laajuuden ja kompleksisuuden takia haluttiin, että käyttöliittymälle tehdään kattavat automaattitestit, jotta voidaan välttyä regressioilta ja jotta käyttöliittymän toiminnallisuudet olisivat dokumentoituja. Asiakassovellusta päätettiin testata päästä-päähän-tyyppisesti. Käyttöliittymätesteissä ei korvata asiakassovelluksesta palvelimelle lähteviä Ajax-kutsuja korvata tyngillä (stub) eikä jäljitelmillä (mock). Tällä tavoin koko järjestelmää testaamalla voidaan varmistua siitä, että asiakassovellus on aina yhteensopiva palvelinsovelluksen tarjoamien rajapintojen kanssa. Käyttöliittymätesteillä ei pyritä testaamaan palvelinsovelluksen rajapintojen kaikkia

reunatapauksia, koska niille on olemassa palvelinsovelluksen omassa testijoukossa aliluvussa 4.1 esitellyn testiautomaatiopyramidin liiketoimintatestitasolle kuuluvat rajapintaintegraatiotestit. Käyttöliittymätesteissä keskitytään enemmän esimerkiksi siihen, että käsiteltävän tiedon tallentamisen jälkeen käyttöliittymä päivittyy oikein vastaamaan nykyistä tilannetta.

Asiakassovelluksen käyttöliittymätesteille ei koettu tarpeen asettaa vaatimukseksi mahdollisuutta kirjoittaa testejä pelkistettynä tekstinä, koska asiakassovelluksen toiminnallinen määrittely oli pääasiallisesti jo tehty paperiprototyyppeinä, joiden muuttaminen pelkistetyksi tekstiksi olisi enimmäkseen päällekkäistä työtä. Lisäksi järjestelmän käyttäjiä ei aikaisemmin palvelinsovelluksen toteutuksessa saatujen kokemusten mukaan pystyttäisi sitouttamaan riittävästi käyttöliittymätestien toteuttamiseen.

Asiakassovelluksen toteutusprojektin alussa käyttöliittymätestejä kokeiltiin kirjoittaa käyttäen palvelinsovelluksen testaamiseen käytettävää ScalaTest-ohjelmistotestauskehystä ja Selenium-ajuria, joka ajaa testit oikeassa verkkoselaimessa [44]. Tällä tavoin käyttöliittymätestien kirjoittaminen ja suorittaminen osoittautui liian hitaaksi. Testien suorittamiseksi Scala-kielellä kirjoitetut testit täytyi ensin kääntää osana palvelinsovellusta ja palvelinsovellus täytyi käynnistää uudelleen, mikä oli myös erittäin hidasta. ScalaTestin ja Seleniumin yhdistelmän verkkosivulle tarjoama ohjelmointirajapinta myös tuntui liian kankealta ja suppealta yhden sivun sovelluksen testaamiseen. Käyttöliittymätestejä toteutettaessa jouduttiin useasti testien sisään lisäämään JavaScript-ohjelmakoodia, jotta saatiin toteutettua tarvittava toiminnallisuus käyttöliittymätestiä varten. Nämä syyt johtivat siihen, että haluttiin etsiä vaihtoehtoisia testausmenetelmiä korvaamaan ensimmäisenä kokeiltu erittäin hitaaksi ja työlääksi osoittautunut tapa kirjoittaa käyttöliittymätestejä.

6.4 Yhden sivun sovellusten testaamisen erityispiirteet

Yhden sivun sovellukset poikkeavat toteutukseltaan ja toimintaperiaatteeltaan huomattavasti perinteisistä verkkopalveluista siltä osin, että yhden sivun sovelluksissa siirtyminen tilojen välillä saattaa sisältää useita asynkronisia HTTP-pyyntöjä ja käyttöliittymän muuttumisen vastausten perusteella. Perinteisissä verkkopalveluissa sovelluksessa siirtymä tilasta toiseen aiheuttaa yleensä kokonaisen sivunlatauksen. Näistä syistä myös yhden sivun sovellusten testaaminen on hyvin erilaista verrattuna perinteisiin verkkopalveluihin. Perinteisissä verkkopalveluissa voidaan sivunlatauksen valmistuttua synkronisesti tarkastaa, että sivulla on odotettu sisältö.

Yhden sivun sovelluksia testattaessa tilasiirtymisen alkamisen ja päättymisen tulkinta on kompleksisempaa kuin perinteisen verkkopalvelun tapauksessa. Esimerkiksi kun halutaan tietää jonkin verkkosivulla olevan sisällön muuttuneen napin painalluksesta aiheutuneen Ajax-kutsun jälkeen ilman kokonaista sivunlatausta, täytyy sen havaitsemiseksi olla jokin muu mekanismi kuin kokonaisen sivunlatauksen seuraaminen. Käytännössä vaihtoehtoina on kahden tyyppisiä odotusmekanismeja: kiinnostavan tilan tarkas-

taminen määräajoin, eli pollaaminen, ja johonkin tapahtumaan (event) sidottu odottaminen.

Pollaaminen voi tapahtua esimerkiksi tarkastamalla verkkosivulla olevan DOM-elementin sisältö määräajoin, kunnes jokin ohjelman aiheuttama odotettu muutos elementin sisällössä on tapahtunut. Tapahtumaan sidottu odottaminen voi tarkoittaa esimerkiksi verkkoselaimen jonkin Ajax-kutsun palaamiseen liitetyn takaisinkutsufunktion suorittamisen odottamista. Näistä vaihtoehdoista voidaan tapahtumaan sidottua odottamista pitää suositeltavampana vaihtoehtona, koska muutoksen pollaamisessa voi aiheutua viivettä varsinaisen muutoksen ja muutoksen havaitsemisen väliin, mutta tapahtumaan sidottu odottaminen saa tiedon käytännöllisesti katsoen välittömästi tapahtuman liipaisun jälkeen.

Kuten luvussa 5 esitettiin, käyttäytymiskeskeiset liiketoimintatestit sisältävät kolme vaihetta: *given*, *when* ja *then*. Käyttöliittymätestit sisältävät usein monia askeleita jokaisessa vaiheessa, jotka voivat olla synkronisia tai asynkronisia. Asynkronisen askelten toimintaperiaatteen vuoksi täytyy askeleet pystyä suorittamaan perätysten jollain mekanismilla. Perinteinen vaihtoehto on lisätä asynkronisen askeleen toteuttavan funktion parametriksi takaisinkutsufunktio, jota kutsutaan, kun askeleen asynkroninen toimenpide on suoritettu. Ohjelma 6.1 on yksinkertaistettu esimerkki siitä, miten takaisinkutsufunktioita ketjuttamismekanismina käyttävä asynkroninen käyttöliittymätesti voisi näyttää CoffeeScript-kielellä toteutettuna [39].

```
it "shows person name", (done) ->
  selectBusinessFromList "business1", ->
    selectPerson "john doe", ->
      openDetailsView ->
        personNameField().should.eq("john")
        done()
```

Ohjelma 6.1: Esimerkki takaisinkutsufunktioita käyttävästä asynkronisesta testistä

Yllä olevasta esimerkistä huomaa, että jo yksinkertaisenkin asynkronisen testin askeleiden ketjuttaminen takaisinkutsufunktioiden avulla vaatii useita sisäkkäisiä funktioita ja siten aiheuttaa ohjelmakoodille nopeasti syvenevän rakenteen. Ohjelmakoodin syvä hierarkia heikentää luettavuutta ja siten koodista tulee vaikeampi ylläpitää. Yksi tai kaksi sisäkkäistä funktioita ei vielä vaikuta häiritsevästi, mutta kolme sisäkkäistä funktiota tai enemmän alkaa heikentämään luettavuutta, varsinkin jos takaisinkutsufunktiot sisältävät paljon ohjelmakoodia. Yksi keino vähentää takaisinkutsuhierarkian syvyyttä ja parantaa ohjelmakoodin luettavuutta on käyttää askeleiden ketjuttamiseen jotain Promise-spesifikaation toteuttavaa kirjastoa.

Promise-spesifikaatio määrittelee nimensä mukaan lupauksen siitä, että yksittäinen operaatio lopulta palauttaa arvon sen valmistuttua. Promisella on kolme mahdollista tilaa: täyttämättä, täytetty ja epäonnistunut. Täytetyn Promisen jälkeen sen sisältämää arvoa ei saa enää muuttaa, odottamattomien sivuvaikutuksien välttämiseksi Promisea kuuntelevassa ohjelmakoodissa. Promise voi vaihtaa tilaansa vain täyttämättömästä täy-

tettyyn tai täyttämättömästä epäonnistuneeseen. Promise ei vain ole mekanismi takaisinkutsujen kokoamiseen vaan se tarjoaa suoran tavan synkronisten ja asynkronisten funktioiden väliseen sanomanvälitykseen. [12]

Promise-spesifikaatio ei määrittele miten Promiset luodaan tai miten Promise käyttäytyy sisäisesti. Se vain määrittelee tarpeellisen rajapinnan, jonka kautta Promisen käyttäjät ovat vuorovaikutuksessa sen kanssa. Spesifikaatiossa Promise määritellään olioksi, jolla on rajapintana yksi metodi ”then”. Then-metodin tulee ottaa vastaan argumentteinaan kolme takaisinkutsufunktiota: success, error ja progress. Jokainen näistä argumenteista on vapaaehtoinen ja muut kuin funktiotyyppiset argumentit hylätään. Success-argumentissa annettua funktiota kutsutaan, kun Promise siirtyy tilaan täytetty, ja error-argumentissa annettua funktiota kutsutaan, kun lupaus siirtyy tilaan epäonnistunut. Progress-argumentissa annettua funktiota kutsutaan, kun Promisen operaatiossa tapahtuu edistymistä. [12]

Aikaisempi ohjelman 6.1 testiesimerkki on toteutettu uudelleen ohjelmassa 6.2 käyttäen Promise-spesifikaation then-funktioita askeleiden ketjuttamiseen. Ohjelman toisella rivillä oleva selectBusinessFromList-funktio palauttaa olion, joka toteuttaa Promise-spesifikaation mukaisen then-rajapinnan. Myös jokaiselle sitä seuraavalle then-funktiolle annettu takaisinkutsufunktio palauttaa Promise-spesifikaation mukaisen olion.

```
it "shows person name", (done) ->
  selectBusinessFromList("business1")
  .then(-> selectPerson("john doe"))
  .then(-> openDetailsView())
  .then ->
    personNameField().should.eq("john")
    done()
```

Ohjelma 6.2: Esimerkki Promise-mekanismia käyttävästä asynkronisesta testistä

Yllä olevasta uudelleen kirjoitetusta esimerkistä voidaan havaita, että Promiseja hyödyntäen toteutettu testin asynkronisten askeleiden ketjuttaminen yksinkertaisessakin ohjelmassa selkeyttää ohjelmakoodin rakennetta. Then-funktiokutsut muodostavat ohjelmakoodiin helposti seurattavan askeleiden ketjun. Uusien asynkronisten lisäaskeleiden lisääminen ketjuun ei muuta ohjelmakoodin rakennetta syvemmäksi, sillä niissä lisätään ketjuun vain yksi then-funktiokutsu, jolle annetaan parametrina askeleen toteuttava funktio.

6.5 Käyttöliittymätestien työkaluvalinnat

Asiakas antoi kehitystiimille vapaat kädet käyttöliittymän työkalu- ja teknologiavalinnoissa. Eri vaihtoehtoja testien toteutustavalle tutkittaessani päätettiin, että asiakassovelluksen koko järjestelmän läpi leikkaavat käyttöliittymätestit tulisi toteuttaa siten, että ne on mahdollista suorittaa oikeassa verkkoselaimessa ja ”päätteettömästi” (headless), eli ilman graafista käyttöliittymää. Käyttöliittymätestien suorittaminen ilman graafista

käyttöliittymää mahdollistaa niiden suorittamisen jatkuva integrointi -ympäristössä. Haluttiin myös, että ne olisivat erillinen kokonaisuus palvelinsovelluksesta, jolloin palvelinsovellusta ei tarvitse kääntää tai käynnistää uudelleen aina kun asiakassovelluksen testikoodiin tehdään muutoksia.

Mahdollisuus suorittaa käyttöliittymätestit oikeassa verkkoselaimessa helpottaa testikoodin ja tuotantokoodin virheiden selvittämistä, koska siten on mahdollista käyttää verkkoselaimen tarjoamia kehittäjätyökaluja testiajon yhteydessä. Kompleksisten ongelmatapausten selvittämiseen Google Chrome -verkkoselaimesta löytyy työkaluja esimerkiksi verkkosivun DOM-puun elementtien tutkimiseen, HTTP-pyyntöjen tarkasteluun ja verkkosivulla olevan JavaScript-ohjelmakoodin debuggaamiseen [26]. Vastaavia työkaluja tulee oletuksena mukana myös muissa moderneissa verkkoselaimissa.

Asiakassovelluksen ei-toiminnallisiin vaatimuksiin oli määritelty, että asiakassovelluksen täytyy tukea vain Google Chrome -verkkoselainta. Näin ollen ei ollut tarvetta etsiä ratkaisua automaattisten käyttöliittymätestien ajamiseen automaattisesti useissa eri verkkoselaimissa.

JSter-sivustolla olevan JavaScript-kirjastokatalogin [45] avulla selvitettiin ohjelmistotestauskehysvaihtoehtoja, joista päädyttiin valitsemaan projektin käyttöön Mocha, joka oli yksi suosituimmista kirjastoista [11]. Mocha on JavaScript-ohjelmointikielellä toteutettu ohjelmistotestauskehys ja se tarjoaa BDD-syntaksin mukaisen rajapinnan testien kirjoittamiseen. Mocha valittiin, koska se tukee tukee synkronisia testejä ja asynkronisia testejä. Koska Mocha on toteutettu JavaScript-ohjelmointikielellä, sitä käyttävät testit on mahdollista suorittaa oikeassa verkkoselaimessa ilman verkkoselaimeen asennettavaa liitännäistä, jonka esimerkiksi Selenium vaatii.

Vaihtoehtoina Mochalle harkittiin suunnilleen samat toiminnot tarjoavaa Jasmine-ohjelmistotestauskehystä, mutta Jasminessa asynkronisten testien kirjoittaminen on hankalampaa kuin Mochassa [30]. Asiakassovelluksen toteutusprojektin alussa testaustyökaluvaihtoehtoja läpikäydessäni Gherkin-syntaksia tukevan Cucumber.jsn [31] kehitystyö oli vielä niin alussa, että sen valitseminen testaustyökaluksi ei ollut mahdollista. Gherkin on Cucumberin käyttämä ohjelman toiminnallisuuden määrittelyyn tarkoitettu kieli, jota noudattaa luvussa 5 esiteltyä given-when-then-tapaa.

Mochan lisäksi käyttöliittymätesteihin valittiin käyttöön myös Chai-apukirjasto [46], joka laajentaa Mochan assertointirajapintaa. Chai-apukirjasto helpottaa helppolukuisten testien kirjoittamista lisäämällä useita deklaratiivisia funktioita käytettäväksi testien tulosten tarkistamiseen.

Testien kirjoittaminen Mochan BDD-tyyppistä rajapintaa käyttäen tapahtuu käyttäen describe ja it -funktioita. Mocha tarjoaa myös testien ennen ja jälkeen suoritettavat alustus- ja siivousfunktiot before, after, beforeEach ja afterEach. Describe-funktion tarkoitus on luoda varsinaisille testeille ympäröivä konteksti, johon voi yhdistää edellä mainittuja alustus- ja siivousfunktioita. Varsinaisen testin vastaanottavat it-funktiot sijoitetaan yhden tai useamman describe-funktion sisään. Molemmat funktiot ottavat sisään ensimmäisenä parametrinaan merkkijonon, joka kuvailee describe-funktiossa kontekstin ja it-funktiossa testattavan asian. Toisena parametrina describe-funktiolle annetaan ta-

kaisinkutsufunktio, jonka sisään voi lisätä sisäkkäisia describe-konteksteja tai it-funktiokutsuja. Myös it-funktio ottaa toisena parametrina takaisinkutsufunktion ja se sisältää testin itse toteutuksen.

Kirjoittamalla describe- ja it-funktiokutsuja sisäkkäin edellä mainitulla tavalla muodostuu testiajon tulosteeksi hiearkkinen näkymä, jonka voi lukea testattavan järjestelmän ominaisuuksia kuvaavina lauseina. Ohjelmassa 6.3 on esimerkki Mochan describe- ja it-funktioiden käyttämisestä.

```
describe "Calculator", ->
   describe "division", ->
      it "divides two numbers", ->
         ...
      it "raises error when dividing with 0", ->
         ...
   describe "sum", ->
      it "calculates two numbers together", ->
         ...
```

Ohjelma 6.3: Esimerkki Mochan describe- ja it -funktioiden käytöstä

Listaus 6.1 esittää ohjelman 6.3 Mocha-testiajon tulosteet, josta voidaan havaita, miten describe- ja it-funktiot tuottavat helposti luettavat tulokset.

```
Calculator
   division
      ✔ divides two numbers
      ✔ raises error when dividing with 0
   sum
      ✔ calculates two numbers together
```

Listaus 6.1: Esimerkkitulostus Mochan testiajosta

Asynkronisten testien kirjoittaminen eroaa synkronisten testien kirjoittamisesta siten, että asynkronisen testin lopussa täytyy kutsua testifunktiolle parametrina annettavaa takaisinkutsufunktiota (usein nimetty done), jolloin Mocha jää odottamaan takaisinkutsufunktion kutsumista testiajon päättymisen merkiksi. Mocha odottaa takaisinkutsufunktion kutsumista, kunnes aikakatkaisu kertoo testin epäonnistuneen. Samaa mekanismia käytetään myös asynkronisissa Mochan alustus- ja siivousfunktioissa.

Mocha tai muutkaan verkkoselaimessa toimivat testausohjelmistokehykset eivät tarkasteluhetkellä tarjonneet täyttä ratkaisua koko järjestelmän läpi leikkaaviin käyttöliittymätesteihin, mutta Mocha vaikutti tähän tarpeeseen parhaiten soveltuvimmalta ratkaisulta. Mochaan on tarjolla lukuisia valmiita liitännäisiä joiden avulla Mochan päälle oli mahdollista rakentaa oma toteutus koko järjestelmän läpileikkaavista käyttöliittymätesteistä.

6.6 Mocha-ohjelmistotestauskehyksen käyttöönotto

Mocha-ohjelmistotestauskehyksen soveltuvuutta järjestelmän kaikkien kerrosten läpileikkaavien käyttöliittymätestien toteuttamisissa kokeiltiin muuntamalla aikaisemmin käytössä olleelle ohjelmistotestauskehykselle kirjoitetut käyttöliittymätestit yhteensopiviksi Mochan kanssa. Käyttöliittymätestien toteuttaminen Mochalle vaikutti tehokkaalta, joten se päätettiin ottaa käyttöön pysyvästi. Käyttöliittymätestit käynnistettiin avaamalla asiakassovellus verkkoselaimessa käyttäen osoitteen mukana tiettyä parametria, jolloin myös käyttöliittymätestit, Mocha ja muut testauskirjastot ladattiin asiakassovelluksen mukana.

Asiakassovelluksen toteutusprojektin alkuvaiheissa asiakassovelluksen käyttöliittymän kautta oli mahdollista vain lukea tietoa, joten tietokannan sisältöä ei voitu muuttaa asiakassovelluksen kautta. Tietosisällön muuttumattomuus mahdollisti sen, että järjestelmän kaikkien kerrosten läpileikkaavia käyttöliittymätestejä voitiin kirjoittaa siten, että testien välissä asiakassovelluksen tila alustettiin käyttöliittymän kautta. Esimerkiksi hakua käyttävän testin jälkeen tila alustettiin tyhjentämällä kaikkien hakukenttien syötteet.

Käyttöliittymätestien määrän kasvaessa huomattiin nopeasti, että testiajon kesto kasvoi paljon odotettua nopeammin. Hidastuminen ilmeni esimerkiksi siten, että testiajon loppuvaiheilla suoritettavat testit olivat niin hitaita, että ne aiheuttivat usein aikakatkaisun ja siten testin epäonnistumisen. Verkkoselaimen muistinkäytön havaittiin kasvavan jatkuvasti testiajon aikana. Mahdollisia syitä tähän olivat asiakassovelluksen käyttöliittymän tai käyttöliittymätestien aiheuttamat muistivuodot tai verkkoselaimen huonosti toimiva roskienkeruu.

Asiakassovelluksen toteutuksen edetessä käyttöliittymään toteutettiin myös tallennusominaisuuksia, jotka muuttivat tietosisältöä pysyvästi. Tietosisältöä muuttavien käyttöliittymätestien myötä riippuvuuksien poistaminen testien väliltä, eli asiakassovelluksen tilan alustaminen käyttöliittymän kautta testin alussa, osoittautui lähes mahdottomaksi.

6.7 Asiakassovelluksen kapselointi

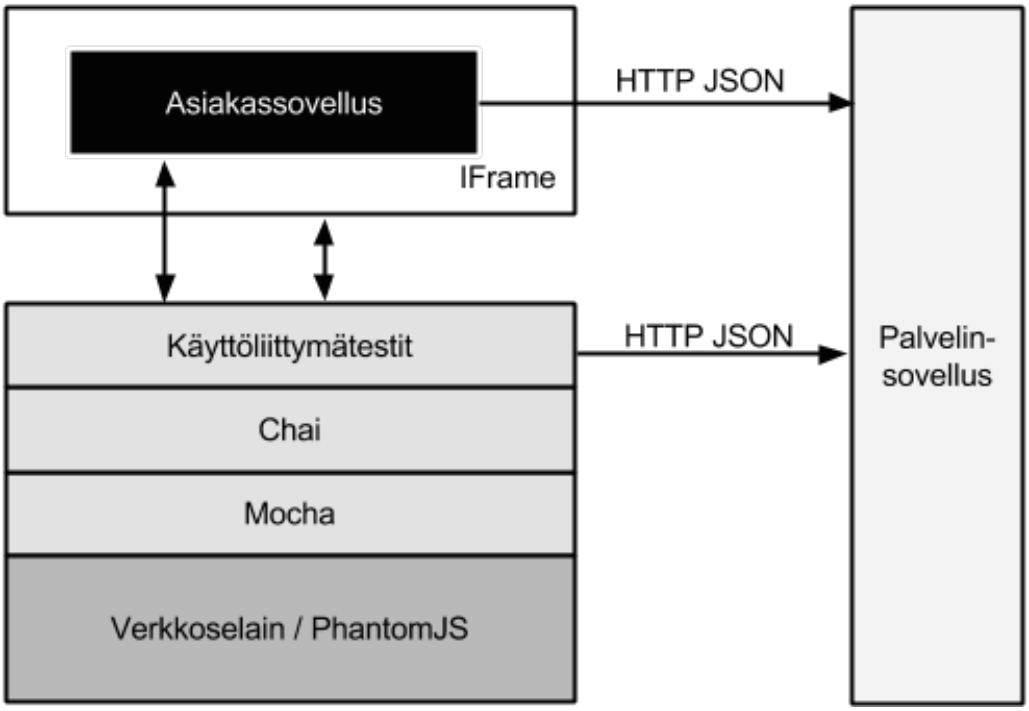
Edellisessä luvussa mainittujen suorituskyky- ja alustusongelmien ratkaisemiseksi haluttiin kehittää jokin mekanismi, jolla asiakassovelluksen tila voidaan käyttöliittymätesteissä alustaa luotettavammin. Käytännössä se tarkoittaa sitä, että asiakassovellus täytyy pystyä lataamaan kokonaan uudelleen, jotta edellisestä käyttöliittymätestistä jäänyt käyttöliittymän tila voidaan alustaa lähtötilaan ennen jokaista testiä.

Yksinkertaisin tapa olisi ladata koko verkkosivu uudestaan jokaisen käyttöliittymätestin jälkeen kutsumalla verkkoselaimessa window.location.reload-funktiota JavaScriptistä, mutta näin ei kuitenkaan voi tehdä, koska testiajuri on osa myös asiakassovelluksen sisältävää verkkosivua. Näin ollen verkkosivun

uudelleenlataaminen kadottaisi testiajon nykyisen tilan, eli mitkä testit on tämänhetkisessä testiajossa jo suoritettu, ja koko testiajo alkaisi alusta.

Tarvittiin siis lisäkerros testiajurin ja asiakassovelluksen väliin, joka eristää asiakassovelluksen uudelleenlataamisen testiajurista. Ratkaisuksi muodostui käyttää HTML IFrame -tekniikkaa asiakassovelluksen käärimiseen verkkosivun sisällä eri kontekstiin, jotta se voidaan ladata uudelleen käyttöliittymätestien välissä. HTML iframe -elementti lisää verkkosivulle sisäkkäisen kontekstin, joka sisältää toisen kokonaisen HTML-dokumentin [28].

Käyttöliittymätestien suorittamista varten luotiin erillinen testiverkkosivu, joka sisältää Mocha-testiajurin sekä tarvittavat apukirjastot. Kaikki testeihin liittyvä ohjelmakoodi siirrettiin pois asiakassovelluksen sisältä ladattavaksi erillisen testiverkkosivun kautta. Kuvassa 6.3 on havainnollistettu käyttöliittymätestien eri komponenttien yhteyksiä.



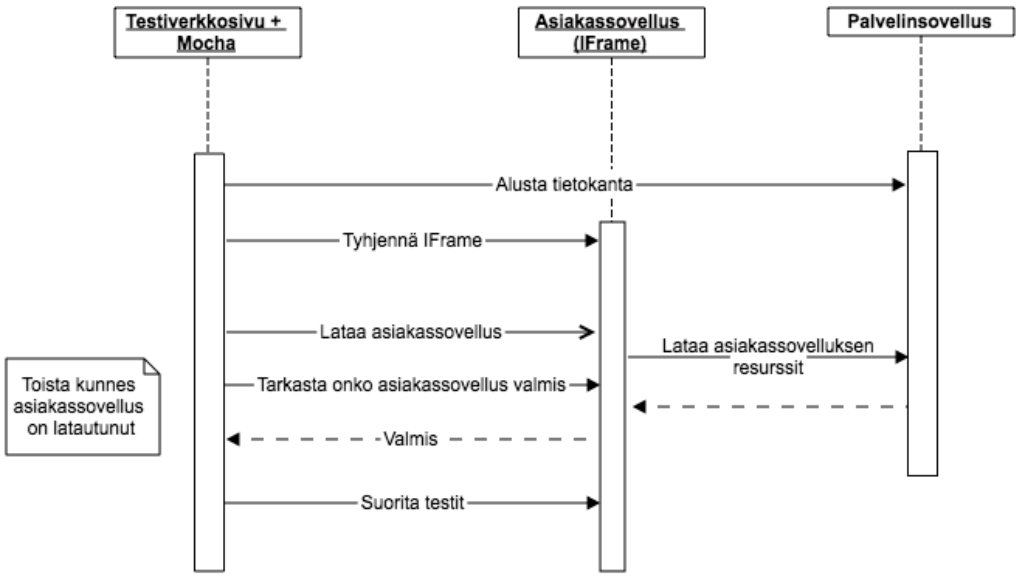
Kuva 6.3. Käyttöliittymätestien komponenttien yhteydet

Ennen jokaista koko järjestelmän läpileikkaavaa käyttöliittymätestiä testiverkkosivulla oleva HTML iframe -elementti poistetaan automaattisesti ja tilalle lisätään uusi HTML iframe -elementti, joka lataa asiakassovelluksen. Tällöin asiakassovellus latautuu kokonaan uudelleen testiverkkosivun sisäiseen kontekstiin vaikuttamatta testiajurin tilaan. Tuotantokäytössä asiakassovellusta ei sijoiteta HTML iframe -elementin sisään.

Viite sisäisen kontekstin window-objektiin löytyy isäntäikkunan window.frames-taulukosta. Sisäisessä kontekstissa sijaitsevan verkkosivun DOM-puuhun pystyy vaikuttamaan sisäisen kontekstin window.document-objektin kautta tai käyttämällä sisäiseen kontekstiin globaalisti ladattua jQuery-kirjaston $-objektia.

Asiakassovelluksen tilan alustamisen lisäksi täytyi muuttunut tietokannan tietosisältö pystyä alustamaan käyttöliittymätestien välissä. Tietokannan alustamista varten luotiin palvelinsovellukseen HTTP-rajapinta, jota kutsumalla tietokantaan ajetaan oletustestidata. Palvelinsovellus tarjoaa testikäyttöön myös muutamia muitakin HTTP-rajapintoja, kuten esimerkiksi hakurajapinnan, joka palauttaa käyttöliittymätesteissä tarvittavaa dataa JSON-muodossa. Nämä rajapinnat ovat käytössä vain kun palvelinsovellus käynnistetään testien ajamiseen tarkoitetusta Scala-sovellusobjektista, joka on eri kuin tuotantokäytössä käytettävä.

Asiakassovelluksen käynnistymisen tilaa ei pysty luotettavasti havaitsemaan ainoastaan odottamalla jotain tapahtumaa, joten asiakassovelluksen latauksen tilaa täytyy kysellä, kunnes se on valmis. Asiakassovelluksessa asetetaan kuvaamaan sovelluksen käynnistymisen tilaa globaali totuusarvomuuttuja, jota käyttöliittymätestit tarkastelevat. Edellä kuvatut yhden käyttöliittymätestin suoritusprosessin vaiheet ovat esitetty sekvenssikaaviona Kuva 6.4.



Kuva 6.4 Sekvenssikaavio käyttöliittymätestien suorittamisesta

Käyttämällä HTML iframe -elementtiä testiverkkosivulle sisäisen kontekstin luomiseen asiakassovellusta varten ja lataamalla asiakassovellus tarvittaessa käyttöliittymätestien välissä uudelleen pysyi verkkoselaimen muistinkäyttö hyväksyttävissä rajoissa eikä aiemman kaltaisia käyttöliittymätestien ajossa esiintyneitä aikakatkaisuongelmia enää ilmennyt.

Useimmat testitapaukset sisältävät monia askeleita, joiden toteutus on yleensä kytköksissä asiakassovelluksen DOM-puuhun. Käyttöliittymätestikoodin haluttiin pysyvän mahdollisimman luettavana, jotta siitä pystytään helposti ymmärtämään, mitä testissä tapahtuu. Luettavuuden säilyttämiseksi testikoodin abstraktiotaso pyrittiin pitämään korkeana toteuttamalla askeleet erilliseen moduuleihin. Erottamalla askeleet testikoodista moduuleihin on testikoodia mahdollista kirjoittaa deklaratiivisesti, asiakassovelluksen

käyttöliittymään sidotut yksityiskohdat eivät häiritse testikoodin lukemista ja askeleita on mahdollista käyttää uudelleen muissa käyttöliittymätesteissä.

Käyttöliittymätesteissä on keskitytty testaamaan laajoja työnkulkuja, jotka vaikuttavat asiakassovelluksen useisiin komponentteihin. Suurimmassa osassa käyttöliittymätestejä käyttöliittymäkomponentteja ei testata yksikkötestimäisesti.

6.8 Jatkuva integraatio

Käyttöliittymätestin ajamiseen päätteettömästi, eli ilman graafista käyttöliittymää, valittiin PhantomJS, joka perustuu avoimen lähdekoodin WebKit-selainmoottoriin ja tarjoaa ohjelmoitavan JavaScript-rajapinnan [27]. PhantomJS itsessään ei ole ohjelmistotestauskehys, vaan sitä käytetään oikean verkkoselaimen korvikkeena halutulle testiajurille, jota ajetaan PhantomJSn sisällä. Mocha-ohjelmistotestauskehyksen käyttämiseksi PhantomJSn kanssa täytyi ottaa käyttöön mocha-phantomjs-laajennos, joka tulkitsee Mochan tuottaman testituloksen [29]. Tällöin PhantomJS-prosessi loppuessaan palauttaa standardin POSIX-muotoisen tilakoodin, jotta sitä voidaan käyttää jatkuva integraatio -palvelimen kanssa.

Asiakassovelluksessa on käytössä Grunt-tehtäväajuri [47]. Grunt-tehtäväajuri on aputyökalu erilaisten toistuvien tehtävien suorittamisen automatisointiin. Valmiiden liitännäisten avulla voidaan helposti ottaa käyttöön Grunt-tehtäviä, jotka esimerkiksi kääntävät CoffeeScript-koodit verkkoselaimessa ajettavaksi JavaScript-koodiksi tai tuotantokäyttöä varten yhdistävät ja minimoivat ohjelmakoodin ja tyylitiedostot.

Kehitystiimillä oli jo järjestelmän palvelinsovelluksen kehityksessä käytössä jatkuvan integroinnin palvelin Jenkins CI, joten asiakassovelluksen jatkuvaa integraatiota varten vain luotiin uusi Jenkins-projekti samaan jatkuva integrointi -ympäristöön. Asiakassovelluksen Jenkins-projekti kytkettiin ajettavaksi jokaisen onnistuneen palvelinsovelluksen testiajon jälkeen, joka suoritetaan jokaisen versiohallintaan tulleen muutoksen laukaisemana. Suorittamalla palvelinsovelluksen testit ensin voidaan varmistua siitä, että palvelinsovellus on kunnossa käyttöliittymätestien ajamista varten.

Asiakassovelluksen Jenkins-projektista kutsutaan ensin testitietokannan alustusrutiinia, joka tyhjentää koko tietokannan, luo tietokantaan uudelleen sovelluksen tietomallin ja lisää palvelinsovelluksen ajamiseen tarvittavat rivit. Jatkuva integrointi -ympäristössä asiakassovelluksen käyttöliittymätesteissä käytettävä tietokantapalvelin on jaetussa käytössä muiden Jenkins-projektien kanssa. Palvelinsovelluksen testit ja asiakassovelluksen käyttöliittymätestit saatetaan suorittaa samanaikaisesti, joten konfliktien välttämiseksi ne käyttävät eri kontekstissa sijaitsevia tietokantatauluja.

Tietokannan alustamisen jälkeen Jenkins kutsuu käyttöliittymätestien Grunt-tehtävää. Grunt-tehtävässä ensin käännetään ja käynnistetään palvelinsovellus testimoodissa ja avataan testit sisältävä verkkosivu PhantomJSssä, jossa testien suoritus tapahtuu samalla tavalla kuin oikeassa verkkoselaimessakin. Testiajon päätyttyä PhantomJS palauttaa Grunt-tehtävälle tilakoodin, jonka Grunt palauttaa Jenkinsille. Gruntin palauttaman tilakoodin perusteella Jenkins merkitsee suorituksen onnistuneeksi tai epäonnistu-

neeksi. Jenkinsin suorittamat käyttöliittymätestit on konfiguroitu tuottamaan xUnit-formaatin mukaista dataa sisältävän XML-tiedoston, josta Jenkins tulkitsee tiedot epäonnistuneista testeistä.

Jokaisen onnistuneen testiajon jälkeen järjestelmän uusi versio otetaan automaattisesti käyttöön kehityspalvelimella, jotta kaikki kehittäjät ja sidosryhmät pääsevät tarkastelemaan versionhallinnassa olevia uusia muutoksia. Järjestelmän tuotantoympäristössä ei ole käytössä jatkuvaa käyttöönottoa.

Jatkuva integrointi -ympäristössä ajettavissa käyttöliittymätesteissä on esiintynyt huomattava määrä epävakausongelmia, joita ei esiinny, kun käyttöliittymätestit ajetaan sovelluskehittäjän omalla työasemalla. Käyttöliittymätestit epäonnistuvat jatkuva integrointi -ympäristössä näennäisen satunnaisesti ja se on heikentänyt kehitystiimin luottamusta käyttöliittymätesteihin. Testien epäonnistumiseen ei enää reagoida samalla tavalla kuin jos epäonnistuminen aina oikeasti tarkoittaisi sitä, että järjestelmässä olisi jokin vika. Epävakauden takia käyttöliittymätestien Jenkins-projektiin on lisätty uudelleenyritysmekanismi, joka yrittää suorittaa käyttöliittymätestit uudelleen niiden epäonnistuessa. Näihin käyttöliittymätestien ongelmien selvittelyyn on kulunut huomattavasti aikaa ja vaivaa. Usein ongelman syyksi on paljastunut väärin toimiva asynkronisen tapahtuman odottaminen, joka on käynyt ilmi vain jatkuva integrointi -ympäristössä. Jotkut palvelinsovelluksessa asynkronisesti suoritettavat toimenpiteet ovat aiheuttaneet käyttöliittymätesteissä odottamattomia virheitä, jotka eivät käy ilmi normaalissa käyttötilanteessa, koska testeissä ei toimintojen suorittamisen välillä ole samanlaista viivettä kuin normaalissa käytössä.

Työn kirjoitushetkellä noin 260 käyttöliittymätestin suorittaminen jatkuva integraatio -ympäristössä kokonaisuudessaan kestää noin 20 minuuttia. Käyttöliittymätestien suorittamiseen kuluva aika on yli kaksi kertaa pidempi kuin aliluvussa 4.2 suositeltu 8–10 minuutin testien ajamisen maksimikesto. Kehitystiimi on kokenut tämän suoritusajan liian pitkänä, mutta ratkaisua tähän ongelmaan ei vielä ole haettu. Kehitystiimin pienen koon vuoksi jatkuva integrointi -palvelimelle testiajojen aikana ei kuitenkaan yleensä ehdi kertymään merkittävästi työjonoa.

Käyttöliittymätestien suorituksessa iso osa ajasta kuluu jokaisen läpileikkaavan testin alussa suoritettavaan asiakassovellukseen alustamiseen. Asiakassovelluksen alustamisen kesto vaihtelee 1100–1300 ms välillä.

6.9 Jatkokehitysideat

Diplomityön kirjoittamisen aikana on noussut esiin muutamia jatkokehitysideoita työn käytännön osuuden kohteena olleiden käyttöliittymätestien parantamisen suhteen. Ensimmäinen ja myös tärkein kehityskohde on käyttöliittymätestien epävakauden poistaminen tai vähentäminen jatkuva integrointi -ympäristössä, jotta kehitystiimi voisi luottaa siihen, että epäonnistuvat testit kertovat aina oikeasta viasta järjestelmässä. Se lyhentäisi virheiden löytämiseen kuluvaa aikaa ja samalla helpottaisi löytyneiden virheiden korjaamista, koska tehdyt muutokset olisivat vielä tuoreessa muistissa.

Käyttöliittymätestien suorittaminen jatkuva integrointi -ympäristössä on tällä hetkellä melko hidasta, joten käyttöliittymätestien suorituskyvyn parantaminen vähentäisi testeistä palautteen saamiseen kuluvaa aikaa. Testiajoa on mahdollista nopeuttaa korvaamalla nopeilla yksikkö- tai komponenttitesteillä nykyisistä koko järjestelmän läpileikkaavista käyttöliittymätesteistä sellaisia reunatapauksia, jotka ovat mahdollista eristää helposti. Eristyksissä testattavien käyttöliittymäkomponenttien eriyttäminen asiakassovelluksesta saattaa vaatia laajaa ohjelmakoodin refaktorointia.

Tällä hetkellä testikoodissa käytettävien testiaskeleiden ja apufunktioiden arkkitehtuuria pitäisi kehittää siihen suuntaan, että niiden uudelleenkäyttö eri käyttöliittymätesteissä olisi helpompaa ja testikoodin monistamiselle ei olisi tarvetta. Usein testiaskeleet ja apufunktiot ovat samoissa moduuleissa. Testiaskeleiden ja apufunktioiden jakaminen pienempiin yleiskäyttöisempiin moduuleihin mahdollistaisi niiden laajemman uudelleenkäyttämisen.

Käyttöliittymätestien luettavuutta olisi mahdollista parantaa ottamalla käyttöön Mocha as Promised -kirjasto, joka helpottaa asynkronisten testien kirjoittamista Mocha-ohjelmistotestauskehyksen kanssa [40]. Mocha as Promised -kirjasto vähentää promisemekanismin käyttämisestä aiheutuvaa moneen paikkaan samanlaisena tai pienin muutoksin sisällytettävän ohjelmakoodin eli boilerplate-koodin määrää.

7 YHTEENVETO

Diplomityön alussa kuvattujen ketterien menetelmien ajatus nopeasta muutoksiin reagoimisesta osoittautui tärkeäksi ominaisuudeksi. Tuotekehitysprojektissa, jonka osana diplomityön käytännön osuus tehtiin, laajasti käytetyt ketterien menetelmien kehitys- ja testauskäytännöt auttoivat kehitystiimiä ja asiakasta kehittämään yhdessä järjestelmään tarvittavat ominaisuudet. Järjestelmän käyttöönoton jälkeen kehitystiimi on pystynyt toimittamaan uusia ominaisuuksia ja havaittujen vikojen korjauksia tasaisin väliajoin.

Diplomityössä kuvattujen läpileikkaavien käyttöliittymätestien käyttäminen pääasiallisena kompleksisen käyttöliittymän testaamisen menetelmänä osoittautui yllättävän haastavaksi toteuttaa ja raskaaksi ylläpitää. Syynä tähän on käyttöliittymätestien suorittamisen hitaus ja näennäisen satunnaiset epävakausongelmat eri ympäristöissä. Muiden kehitystiimin jäsenten mukaan nämä havainnot ovat toistuneet usein muissakin projekteissa, joissa on tehty paljon automaattisia käyttöliittymätestejä.

Tällä hetkellä tuotekehitysprojekti on siirtymässä ylläpitovaiheeseen, jossa uusia ominaisuuksia ei enää kehitetä yhtä intensiivisesti kuin aktiivisessa kehitysvaiheessa. Jatkossa järjestelmän alkuperäinen kehitystiimi luovuttanee järjestelmän ylläpitovastuun jollekin toiselle taholle, joka ei tunne järjestelmän toimintaa yhtä hyvin. Tällöin kaikkien testien merkitys dokumentaationa kasvaa ja testien luotettavuus korostuu, koska muuten uusi ylläpitotiimi ei voi tietää toimiiko järjestelmä oikein, kun siihen on tehty muutoksia.

Tulevissa vastaavanlaisissa projekteissa tulen painottamaan enemmän liiketoimintalogiikan tarkempaa erottamista käyttöliittymäkoodista, jotta liiketoimintalogiikan reunatapaukset olisi helpompi testata. Jatkossa pyrin tekemään enemmän käyttöliittymäkomponenttien yksikkötestausta, joka vaatii myös hieman erilaista arkkitehtuuria käyttöliittymäsovellukselle.

Läpileikkaavilla käyttöliittymätesteillä on tehtävänsä varmistamassa laajoja työnkulkuja ja yhteensopivuutta taustajärjestelmien kanssa, mutta niiden tueksi kannattaa muodostaa vahva pohja matalamman abstraktiotason testausmenetelmistä. Käyttöliittymän automaattisen testaamisen määrässä tulee ottaa huomioon myös kehitettävän käyttöliittymän laajuus ja esimerkiksi regressioista aiheutuvien ongelmien riskit. Yksinkertaisen sovelluksen tapauksessa kattavien automaattisten käyttöliittymätestien tekeminen ei välttämättä kannata.

Käyttöliittymän uusien ominaisuuksien kehityksessä myös tutkivan testauksen tarve korostui. Automaattisilla käyttöliittymätesteillä voitiin varmistua yleisten työkulkujen toimivuudesta, mutta ihmisen suorittamaa tutkivaa testausta tarvittiin selvittämään eri-

koisempien reunatapausten toimivuus. Järjestelmän kehitystiimissä ei ollut osoitettua testaajaa ja sen puutteen huomasi usein perusteellisen testaamisen puutteena.

LÄHTEET

[1] North, D. Introducing BDD [WWW]. [viitattu 21.10.2012]. Saatavissa: http://dannorth.net/introducing-bdd/

[2] Koskela, L. Test Driven. 2008, Manning Publications Co. 513 s.

[3] Solís, C. & Wang, X. A Study of the Characteristics of Behaviour Driven Development. 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications. pp. 383-387. [viitattu 22.10.2012]. Saatavissa: http://ieeexplore.ieee.org/

[4] Rook, P. Controlling software projects. Software Engineering Journal (1986)1, pp. 7-16

[5] Adzic, G. Bridging the Communication Gap. Lontoo 2009, Neuri Limited. 258 s.

[6] Beck, K., Grenning, J., Martin R. C., Beedle M., Highsmith J., Mellor S., van Bennekum A., Hunt A., Schwaber K., Cuckburn A., Jeffries R., Sutherland J., Cunningham W., Kern J., Thomas D., Fowler M. & Marick B. Manifesto for Agile Software Development [WWW]. 2001. [viitattu 27.10.2012]. Saatavissa: http://agilemanifesto.org/

[7] Crispin, L. & Gregory, J. Agile Testing: A Practical Guide for Testers and Agile Teams. 1. painos. Boston 2009, Pearson Education Inc. 533 s.

[8] Meszaros, G. XUnit Test Patterns: Refactoring Test Code. 2007, Addison-Wesley

[9] Adzic, G. Specification by Example. Toinen painos. New York 2012, Manning Publications Co. 270 s.

[10] V Model | Waterfall Model [WWW]. [viitattu 11.11.2012]. Saatavissa: http://www.waterfall-model.com/v-model-waterfall-model/

[11] Mocha [WWW]. [viitattu 14.9.2013]. Saatavissa: http://visionmedia.github.io/mocha/

[12] Promises/A - CommonJS Spec Wiki [WWW]. [viitattu 14.9.2013]. Saatavissa: http://wiki.commonjs.org/wiki/Promises/A

[13] Cucumber - Making BDD fun [WWW]. [viitattu 12.11.2012]. Saatavissa: http://cukes.info/

[14] Implementations - Behaviour-Driven Development [WWW]. [viitattu 25.11.2012]. Saatavissa: http://behaviour-driven.org/Implementations

[15] Behavior-driven development [WWW]. [viitattu 2.12.2012]. Saatavissa: http://en.wikipedia.org/wiki/Behavior-driven_development

[16] Wynne, M. & Hellesøy, A. The Cucumber Book. 2012. Pragmatic Bookshelf. 313 s.

[17] jnicklas/capybara · GitHub [WWW]. [Viitattu 22.12.2012]. Saatavissa: https://github.com/jnicklas/capybara

[18] Selenium - Web Browser Automation [WWW]. [Viitattu 25.12.2012]. Saatavissa: http://seleniumhq.org/

[19] Graham, D. & Fewster, M. Experiences of Test Automation. 1. painos. Crawfordsville 2012, Addison Wesley. 617 s.

[20] Crisp's Blog » Continuous Delivery vs Continuous Deployment [WWW]. [viitattu 3.3.2013]. Saatavissa: http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment

[21] Humble, J., Read, C. & North, D. The Deployment Production Line. AGILE '06 Proceedings of the conference on AGILE 2006. pp. 113-118. [viitattu 3.3.2013]. Saatavissa: http://doi.ieeecomputersociety.org/10.1109/AGILE.2006.53

[22] Haikala, I. & Märijärvi, J. Ohjelmistotuotanto. 11. painos. Helsinki 2006, Talentum Media Oy. 440 s.

[23] What Is Kanban? [WWW]. [viitattu 18.5.2013]. Saatavissa: http://www.kanbanblog.com/explained/

[24] Master data management – Wikipedia, the free encyclopedia [WWW]. [viitattu 21.5.2013]. Saatavissa: http://en.wikipedia.org/wiki/Master_data_management

[25] Thin client [WWW]. [Viitattu 25.8.2013]. Saatavissa: http://en.wikipedia.org/wiki/Thin_client

[26] Chrome DevTools - Google Developers [WWW]. [viitattu 8.9.2013]. Saatavissa: https://developers.google.com/chrome-developer-tools/

[27] PhantomJS: Headless WebKit with JavaScript API [WWW]. [viitattu 8.9.2013]. Saatavissa: http://phantomjs.org/

[28] HTML/Elements/iframe -W3C Wiki [WWW]. [viitattu 21.9.2013]. Saavissa: http://www.w3.org/wiki/HTML/Elements/iframe

[29] metaskills/mocha-phantomjs [WWW]. [viitattu 19.10.2013]. Saatavissa: https://github.com/metaskills/mocha-phantomjs

[30] Jasmine [WWW]. [viitattu 22.1.2014]. Saatavissa: http://pivotal.github.io/jasmine/

[31] cucumber/cucumber-js · GitHub [WWW]. [viitattu 21.4.2014]. Saatavilla: https://github.com/cucumber/cucumber-js

[32] Schwaber, K. & Sutherland, J. 2011. The Scrum Guide [WWW]. [viitattu 24.1.2014]. Saatavissa: https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum%20Guide%20-%20FI.pdf

[33] Janzen D. S. & Saiedian H. 2008. Does Test-Driven Development Really Improve Software Design Quality?. IEEE Software [verkkolehti]. 25, 2, pp. 77-84 [viitattu 27.1.2014]. Saatavissa: http://www.computer.org/csdl/mags/so/2008/02/index.html

[34] A Sprint is Not a Mini Waterfall - Scrun Alliance [WWW]. 2012. [viitattu 30.1.2014]. Saatavissa: http://www.scrumalliance.org/community/articles/2012/january/a-sprint-is-not-a-mini-waterfall

[35] Hüttermann M. Agile ALM. 1. painos. Shelter Island 2012, Manning Publications Co. 332 s.
[36] Duvall P. M., Matyas, S. & Glover A. Continuous Integration: Improving Software Quality and Reducing Risk. New Jersey 2007, Pearson Education. 336 s.
[37] Marick B. How to Misuse Code Coverage [WWW]. 1999. Saatavilla: http://www.exampler.com/testing-com/writings/coverage.pdf
[38] Taivalsaari A., Mikkonen T., Anttonen M. & Salminen A. The Death of Binary Software: End User Software Moves to the Web. 2011 Ninth International Confe- rence on Creating, Connecting and Collaborating through Computing. pp 17–23.
[39] CoffeeScript [WWW]. [viitattu 29.3.2014]. Saatavilla: http://coffeescript.org/
[40] Domenic/mocha-as-promised [WWW]. [viitattu 6.2.2014]. Saatavilla: https://github.com/domenic/mocha-as-promised
[41] Anderson, D. J. Kanban. Sequim 2010, Blue Hole Press. 261 s.
[42] Vuori, M. 2010. Menestyksekäs hyväksymistestaus [WWW]. [viitattu 29.3.2014]. Saatavilla: http://www.mattivuori.net/julkaisuluettelo/liitteet/menestyksekas_hyvaksymistestaus.pdf
[43] The Scala Programming Language [WWW]. [viitattu 29.3.2014]. Saatavilla: http://www.scala-lang.org/
[44] ScalaTest [WWW]. [viitattu 29.3.2014]. Saatavilla: http://www.scalatest.org/
[45] Javascript Testing Frameworks - JSter Javascript Catalog [WWW]. [viitatttu 29.3.2014]. Saatavilla: http://jster.net/category/testing-frameworks
[46] Home – Chai [WWW]. [viitattu 21.4.2014]. Saatavilla: http://chaijs.com/
[47] Grunt: The JavaScript Task Runner [WWW]. [viitattu 21.4.2014]. Saatavilla: http://gruntjs.com/