Linda Talve

TESTAAMINEN KIINTEÄKSI OSAKSI OHJELMISTOPROJEKTEJA

Testausviitekehyksen laatiminen paikkatietoyritykselle

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittelyn koulutus

2023

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Tradenomi (AMK)
Tekijä	Linda Talve
Työn nimi	Testaaminen kiinteäksi osaksi ohjelmistoprojekteja – Testausviitekehyksen laatiminen paikkatietoyritykselle
Toimeksiantaja	Gispo Oy
Vuosi	2023
Sivut	46 sivua
Työn ohjaaja	Marjo Puikkonen

TIIVISTELMÄ

Ohjelmistotestaus on ohjelmistokehityksen keskeinen osa ja olennainen laatutekijä. Ohjelmistojen järjestelmällinen testaus kehitystyön aikana tehostaa ohjelmointia, minkä lisäksi testaus tekee ohjelmistoista helpommin ylläpidettäviä. Testauksen toteuttaminen mahdollisimman tehokkaalla ja hyödyllisellä tavalla edellyttää testauskokonaisuuden järjestelmällistä suunnittelua.

Tässä opinnäytetyössä laaditaan viitekehys ohjelmistotestauksen suunnitteluun, jota toimeksiantajayritys Gispo Oy voi hyödyntää paikkatietoalan ohjelmistoprojekteissaan. Lisäksi työssä käytetään tehtyä viitekehystä laatimalla sen pohjalta testaussuunnitelma yrityksessä parhaillaan käynnissä olevalle asiakasprojektille. Viitekehyksen ja sen pohjalta tehdyn projektikohtaisen testaussuunnitelman yhteisenä tavoitteena on varmistaa, että Gispon kehittämät sovellukset toimivat vakaasti ja halutulla tavalla sekä niiden kehitys- että tuotantovaiheissa.

Työn teoreettinen viitekehys koostuu testauksen roolin ja merkityksen tarkastelusta erilaisissa ohjelmistokehitysprojekteissa sekä siitä, mitä tekijöitä tulee ottaa huomioon toimivaa testikokonaisuutta suunniteltaessa. Näiden pohjalta tarkastellaan testausta Gispolle tyypillisten paikkatietosovellusten viitekehyksessä ja esitetään mahdollisia työkaluja, joita yritys voisi hyödyntää projektiensa testien toteutuksessa. Lopuksi esitetään testaussuunnitelma Lahden seudun jätehuoltoviranomaiselle tehtävälle jätehuollon seurantajärjestelmäprojektille. Suunnitelma kattaa projektin kuvauksen, ehdotuksen järjestelmän eri tasoille tehtävistä testeistä sekä niiden yleisen aikataulut ja vastuunjaon.

Opinnäytetyö rajautuu testausviitekehyksen ja -suunnitelman laatimiseen eikä sen osana kirjoiteta tai toteuteta varsinaisia testejä. Viitekehys tarjoaa pohjan Gispon tulevien ohjelmistoprojektien testaussuunnitelmille. Projektikohtainen testaussuunnitelma on ehdotus, ja lopulliset testit valitaan ja määritellään tarkemmin projektin edetessä Gispo Oy:n ja asiakkaan yhdessä sopimalla tavalla.

Asiasanat: ohjelmistokehitys, ohjelmistotestaus, testaussuunnitelma

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Business Administration
Author	Linda Talve
Thesis title	Making software testing a standard practice – Creating a testing framework for a GIS company
Commissioned by	Gispo Oy
Time	2023
Pages	46 pages
Supervisor	Marjo Puikkonen

ABSTRACT

Software testing is an important tool for increasing the quality of software products. Testing is done to ensure that a product meets its objectives, works as intended and that any critical errors are found. Systematic testing practices during programming make the work faster and more effective. Additionally, software with good test coverage is notably easier to maintain and develop further. To build an efficient, useful, and realistic test suite for a software project requires knowledgeable and careful planning.

The objective of the thesis was to create a test framework for Gispo Ltd, and to demonstrate how the framework could be used by applying it to an ongoing customer project by creating a test plan for the project. The framework and the test plan both aimed at increasing the quality of Gispo's software projects by offering a solid framework for planning and building test suites in the industry context Gispo operated in.

The theoretical background was built by discussing the role and importance of testing in different types of software projects, as well as reflecting on the variety of aspects to be considered when building a comprehensive test plan for a project. Following this, the work focused on Gispo and the methods and tools in the typical software projects of the company. These were most often GIS-related solutions created using open-source tools like QGIS and PostgreSQL databases. Finally, the presented framework was tested in practice by using it to create a test plan for a waste management monitoring system Gispo was building for the Lahti Region Waste Management Authority. The test plan covered general descriptions of tests on the different levels of the system, a schedule, and a division of responsibilities.

Building reliable and well-functioning software that meets the users' needs always requires testing. The size of Gispo's software projects has been growing in the recent years resulting in the need for a more solid approach to testing. The framework created in this thesis provides a tool for holistic and efficient testing efforts in Gispo's current and future projects.

Keywords: software testing, software development, test plan

SISÄLLYS

1 JOHDANTO

Ohjelmistotestaus on ohjelmistokehityksen keskeinen osa ja olennainen laatutekijä. Ohjelmistojen järjestelmällinen testaus kehitystyön aikana tehostaa ohjelmointia, minkä lisäksi testaus tekee ohjelmistoista helpommin ylläpidettäviä. Testauksen toteuttaminen mahdollisimman tehokkaalla ja hyödyllisellä tavalla edellyttää testauskokonaisuuden järjestelmällistä suunnittelua.

Ohjelmistotestaus on oma erikoisalansa ja toimivan testauskokonaisuuden luominen vaatii asiantuntemusta. Työtä helpottaa, jos käytettävissä on viitekehys, jonka avulla voidaan valita testaukseen liittyvistä monipuolista vaihtoehdoista ne, jotka ovat toimialakohtaisesti sekä tekeillä olevan ohjelmiston näkökulmasta varteenotettavia. Tässä opinnäytetyössä koostetaan toimeksiantajayritys Gispo Oy:lle (myöhemmin Gispo) viitekehys, jonka avulla se voi hyödyntää testauksen tarjoamia hyötyjä osana ohjelmistokehitysprojektejaan.

Opinnäytetyössä keskitytään seuraaviin kysymyksiin:

1. Miksi ohjelmistotestaus on tärkeää ja millainen rooli testauksella on nykyaikaisessa ohjelmistokehityksessä?
2. Miten Gispo voi suunnitella ohjelmistotestausta kokonaisvaltaisesti ja kiinteänä osana ohjelmistoprojektejaan?
3. Miten laadittua testauksen viitekehystä voidaan soveltaa ohjelmistoprojektissa?

Työn teoreettinen viitekehys koostuu testauksen roolin ja merkityksen tarkastelusta erilaisissa ohjelmistokehitysprojekteissa sekä siitä, mitä tekijöitä tulee ottaa huomioon toimivaa testikokonaisuutta suunniteltaessa. Näiden pohjalta laaditaan Gispolle viitekehys, jota yritys voi hyödyntää suunnitellessaan testausta paikkatietosovellusprojekteissaan. Lopuksi viitekehyksen käyttöä havainnollistetaan laatimalla sen pohjalta testaussuunnitelma Lahden seudun jätehuoltoviranomaiselle tehtävälle jätehuollon seurantajärjestelmäprojektille.

Opinnäytetyö rajautuu testausviitekehyksen ja -suunnitelman laatimiseen eikä sen osana kirjoiteta tai toteuteta varsinaisia testejä. Viitekehys tarjoaa pohjan Gispon tulevien ohjelmistoprojektien testaussuunnitelmille. Työssä esitetty projektikohtainen testaussuunnitelma on ehdotus, ja lopulliset testit valitaan ja

määritellään tarkemmin projektin edetessä Gispon ja asiakkaan yhdessä sopimalla tavalla.

2 KESKEISIÄ KÄSITTEITÄ

2.1 Ohjelmistotestaus

Ohjelmistotestauksella tarkoitetaan yleisesti niitä prosesseja, jotka liittyvät ohjelmiston suunnitteluun, kehitykseen ja arviointiin ja joilla varmistetaan, että ohjelmisto täyttää sille asetetut vaatimukset, toimii suunnitellusti ja että sen kriittiset virheet löydetään (Kasurinen 2013, 9). Ohjelmistotestauksella ei pyritä eikä pystytä takaamaan ohjelmiston täydellistä virheettömyyttä. Jo yksinkertaisessa ohjelmistossa mahdollisten tilojen ja käyttöpolkujen määrä kasvaa liian suureksi, jotta niistä jokainen voitaisiin testata erikseen. Näin ollen testaamisella pyritään varmistamaan, että ohjelmisto on *riittävän* hyvä käytettäväksi. (Myers ym. 2012, 7–12; Luukkainen 2022; Kasurinen 2013, 14; Bertolino 2001, 88.)

Järjestelmällinen ohjelmistotestaus alkaa testausprosessin kokonaissuunnittelulla, jossa suunnitellaan yleisellä tasolla testauksen tavoitteet, menetelmät, työkalut, resurssit, aikataulu ja työnjako. Yksityiskohtaisempaa testien suunnittelua ja toteutusta tehdään kehitysprojektin projektimallista riippuen sitä mukaa kun testien määrittelyä varten tarvittava tieto on saatavilla ja koko projektin ajan. (Kennett ym. 2018, 4–6.)

Tyypillisesti testausta kohdennetaan ohjelmiston eri tasoille, jotta voidaan varmistua ohjelmiston toimivuudesta kokonaisuutena. **Yksikkötestien** avulla varmistetaan ohjelman yksittäisten komponenttien toiminta. **Integraatiotestit** testaavat komponenttien toimintaa yhdessä toistensa kanssa sekä tehtyjä arkkitehtuuriratkaisuja. Toiminnallisuuksien valmistuessa suoritetaan **järjestelmätestausta**, jonka tarkoituksena on testata ohjelmiston toimintaa kokonaisuutena valituissa tilanteissa. Usein testauksen viimeistä tasoa kutsutaan **hyväksymistestaukseksi**, jossa ohjelmistoa käytetään todellisessa tai todellista simuloivassa ympäristössä ja varmistetaan sen olevan alkuperäisten käyttäjävaatimusten mukainen. (Kasurinen 2013, 10–11.)

Yksikkö-, integraatio- ja järjestelmätestien avulla **verifioidaan** eli todennetaan ohjelmiston osien ja niille asetettujen vaatimusten keskinäinen vastaavuus. Hyväksymistestit puolestaan **validoivat** läpimenollaan sen, että kehitetty ohjelmisto vastaa sille suunniteltua käyttötarkoitusta. (Adam 2021.) On kuitenkin huomattava, että eritasoisten testien määritelmät ovat varsin häilyviä ja esimerkiksi ”yksikkötestillä” voidaan tarkoittaa hyvin erilaisia asioita kontekstista ja määrittelystä riippuen (Fowler 2021).

Kullakin testitasolla voidaan suorittaa testejä erilaisin menetelmin. Yksittäisten testien tasolla **testitapaus** määrittää, miten jonkin yksittäisen toiminnallisuuden toimivuuden testaus suoritetaan ja miten järjestelmän tulisi tähän reagoida. Testitapaus voi esimerkiksi olla sarja arvoja, jotka syötetään ohjelmalle ohjeiden mukaisesti, sekä kullekin arvolle oletettu palautusarvo, jonka ohjelman oletetaan palauttavan. Lisäksi testitapauksessa määritetään ehdot sille, missä tilanteissa testi tulee keskeyttää ja milloin se puolestaan menee hyväksytysti läpi. (Kasurinen 2013, 77–78.)

Testausta tekevät sekä ohjelmiston toimittaja että asiakkaat. Asiakkaiden valmiudet osallistua testauksen suunnitteluun vaihtelevat, mutta asiakkaan osallistuminen testauksen toteutukseen on erityisen tärkeää, jotta voidaan varmistua siitä, että ohjelma tekee sitä mitä sen tulee tehdä. (Holcombe 2008, 183.) Toisaalta on myös tärkeää, että testaukseen osallistuvat henkilöt, jotka eivät ole ohjelmiston kehittäjiä tai muuten niin lähellä kehitystyötä, että objektiivisuuden voidaan nähdä vaarantuvan (Myers ym. 2012, 15).

2.2 Testaussuunnitelma

Testaussuunnitelma on ohjelmistoprojektin olennainen dokumentti, jossa määritellään, miten varmistetaan, että ohjelmisto vastaa sille asetettuja vaatimuksia: mitä testataan, millä menetelmällä ja missä vaiheessa. Hyvin laadittu testaussuunnitelma selkiyttää projektitiimin työskentelyä määrittelemällä testauksen kohteet, testien sisällöt ja testien aikataulutuksen mahdollisimman hyvin. Kun ohjelmiston ongelmat löydetään ajoissa ja niiden korjaamisen varataan aikaa, projektin aikataulu ja usein myös budjetti pysyvät todennäköisimmin suunnitellussa. (Passby 2021.)

Testaussuunnitelman laatii usein toimittajan projekti- tai suuremmissa hankkeissa testauspäällikkö, mutta suunnitelma voidaan tehdä myös yhteistyössä asiakkaan kanssa. Joskus asiakas vastaa hyväksymistestaussuunnitelman tekemisestä. Suunnitelman tyypilliset osat kattavat projektin kuvauksen, kuvauksen testattavasta tuotteesta ja sen mahdollisesti sisältämistä riskeistä, testien laajuuden ja toteutusmenetelmät, aikataulun ja työnjaon (Kasurinen 2013, 75–76).

2.3 Manuaalinen testaus ja automaatiotestaus

Testausta voidaan suorittaa monin eri tavoin, mutta selkein yksittäinen jaottelu menetelmissä lienee jako manuaaliseen ja automaatiotestaukseen. Manuaalinen testaus on ihmisen toimesta tehtävää testausta, kun taas automaatiotestauksessa tietokone ohjelmoidaan suorittamaan halutut testit ja tarkastamaan niiden avulla, että ohjelmisto toimii kuten sen toivotaan toimivan. Vaikka teoriassa lähes kaikki testaus voidaan automatisoida, automaation on todettu olevan erityisen hyödyllistä yksittäisten ohjelmistokomponenttien tai niiden yhdistelmien regressiotestauksessa, jossa automaatio tarkastaa jokaisen tehdyn muutoksen jälkeen, että aiemmin kehitetyt osat eivät ole menneet viimeksi tehdyn muutoksen takia rikki (Kasurinen 2013, 50; Kumar & Mishra 2016, 10).

Manuaalinen ja automatisoitu testaus eivät ole toisiaan poissulkevia menetelmiä, vaan niitä käytetään usein toisiaan täydentävinä. Automatisoidut testit havainnoivat virheitä nopealla syklillä ja vapauttavat ohjelmoijat muihin tehtäviin, mikä tehostaa ohjelmistoprojektien läpivientiä. (Kasurinen 2013, 49; Kumar & Mishra 2016, 9–10.) Manuaalisen testauksen vahvuudet näyttäytyvät puolestaan ihmisen arviointia korostavissa testaamisen muodoissa. Esimerkiksi **tutkivassa testaamisessa** järjestelmän toimintaa ja sisältöä hyvin tunteva testaaja testaa ohjelman toimintoja sen mukaan, missä hän asiantuntemuksensa pohjalta osaa nähdä mahdollisuuksia virheiden olemassaololle. (Kasurinen 2013, 47–48.)

Testiautomaation nopeasta kehityksestä huolimatta testiautomaatiotyökaluilta odotetaan edelleen merkittävää kypsymistä. Lisäksi testiautomaation käyttö hyödyllisesti ja kustannustehokkaasti edellyttää organisaatioilta osaamista.

(Wang ym. 2022, 3.) Valitettavan usein testiautomaatio koetaan työläänä aloittaa ja vaikka sen selkeät edut tunnistetaan teoriassa, käytännössä ne jäävät osittain tai jopa kokonaan hyödyntämättä (Palamarchuk 2015).

2.4 Muita keskeisiä käsitteitä

Testikattavuus tarkoittaa sitä osuutta ohjelmistosta tai järjestelmästä, jonka testit kattavat. Usein tätä mitataan prosenttiosuuksina kirjoitetun koodin osalta ja laskennan tekee testaukseen käytettävä ohjelmallinen työkalu. Jos testaus tehdään manuaalisesti, voidaan testikattavuutta mitata ohjelmiston toimintotasolla esimerkiksi vertaamalla ohjelmiston vaatimuslistaa toteutettuihin manuaalitesteihin ja niiden kattamiin toimintoihin. Toiminto voi olla esimerkiksi järjestelmään sisäänkirjautuminen käyttäjätunnuksella ja salasanalla. (ISO/IEC/IEEE 29119-1:2022(en) 2022.)

Regressiotestaus on tavallinen testauksen tapa, jossa ohjelmistoa testataan suhteessa omaan aiempaan ohjelmistoversioonsa. Käytännössä tämä tarkoittaa, että ohjelma esimerkiksi testataan jollakin syötteellä ja todetaan toimivaksi. Tämän jälkeen tehdään jokin muutos ohjelmaan ja toistetaan testi sen varmistamiseksi, että ohjelma toimii muutoksesta huolimatta edelleen toivotulla tavalla. (Kasurinen 2013, 43.)

3 TESTAUS OHJELMISTOKEHITYKSEN OSANA

Ohjelmistokehitysprojektien koko ja kompleksisuus ovat kasvaneet viime vuosikymmeninä jatkuvasti. Samaan aikaan vaatimukset tuottaa ohjelmistotuotteita markkinoille yhä nopeammin ovat lisääntyneet. (Dominguez 2009; Kumar & Mishra 2016, 9–10.) Ohjelmistoliiketoiminnan pitkään jatkunut voimakas kasvu on pitänyt esillä ohjelmistoprojekteihin usein liittyviä ongelmia, kuten aikataulujen ja budjettien ylityksiä sekä toteutettujen ohjelmistojen heikkoa laatua. Ohjelmistotoimittajien kyvystä toimittaa markkinoille hyvälaatuisia ohjelmistoja suunnitellussa ajassa ja annetun budjetin puitteissa on tullut kilpailuvaltti. (Slaughter ym. 1998, 67.)

Testauksen avulla etsitään ohjelmiston virheitä ja pyritään varmistamaan, että ohjelmisto täyttää sille asetetut vaatimukset. Testaus on työkalu, jonka avulla ohjelmiston laatua voidaan parantaa monin eri tavoin ja varmistua siitä, että ohjelmisto kattaa sille annetut odotukset. Tässä luvussa tarkastellaan ohjelmistojen yleisiä laadun määritelmiä sekä sitä, miten testaus tukee näiden saavuttamista perinteisessä vesiputousmallissa sekä nykyaikaisemmissa ketterissä projektimalleissa.

3.1 Ohjelmiston laatutekijät

Sekä ohjelmistojen että ohjelmistoprojektien laatua voidaan arvioida erilaisin kriteeristöin. Näistä tunnetuimpia lienee ohjelmistojen osalta kansainvälinen standardi ISO/IEC 25010:2011(en) (2011), joka jakaa ohjelmistojen laatuvaatimukset käytettävyyden laatuun (*quality in use*) ja tuotteen laatuun (*product quality*). Käytettävyyden laatutekijöinä standardi listaa vaikuttavuuden, tehokkuuden, tyytyväisyyden ja riskittömyyden. Tuotteen laatutekijöinä listataan toiminnallinen sopivuus, suorituskyvyn tehokkuus, yhteensopivuus, käytettävyys, luotettavuus, turvallisuus, ylläpidettävyys sekä siirrettävyys. Standardin luokituksen lisäksi on monia muitakin listoja laadun kriteereistä, mutta suuria eroavaisuuksia on vaikea löytää esim. Kasurisen (2013, 88–89) esityksestä. Useimmiten luokitukset myös pohjautuvat edellä esitettyihin standardeihin (esim. Vala, s.a.).

Ohjelmistoprojektien onnistumista mittaa puolestaan ohjelmistoalalla tunnettu, ohjelmistokehitysprojektien onnistumista arvioiva CHAOS-raportti (The Standish Group 2015, 1–2). Sen mukaan ohjelmistoprojektien onnistumisen pääelementit ovat aikataulussa ja budjetissa pysyminen. Kolmas onnistumisen arviointikriteeri vuosittaisissa arviointiraporteissa oli aiemmin se, miten hyvin ohjelmistot vastaavat niille määritettyjä vaatimuksia. Viimeksi mainitulle kriteerille esitetyn kritiikin (mm. Dominguez 2009) myötä The Standish Group (2015) on tarkastanut kriteeristöään. Sen sijaan, että ohjelmistoa tarkasteltaisiin suhteessa sen alkuperäisiin vaatimusmäärittelyihin, onnistumista arvioidaan sen mukaan, miten hyvin lopullinen ohjelmisto tuottaa käyttäjille lisäarvoa ja vastaa heidän tarpeisiinsa.

Sekä ohjelmistoja että ohjelmistoprojekteja arvioitaessa on muistettava, että jokaisen niistä kohdalla on tärkeä määrittää mitä mikin mittari tarkoittaa juuri kyseisen ohjelmiston tai projektin osalta (Wallace & Reeker 2001, 183). Mitä varhaisemmassa vaiheessa määrittelyt laaditaan, sitä todennäköisemmin laatuvaatimukset myös toteutuvat (Vala s.a., 3). On tärkeä määrittää selkeästi, mitä tarkalleen ottaen tarkoitetaan esimerkiksi ohjelmiston ”tehokkuudella” tai milloin projekti ”vastaa vaatimuksiaan”. Ilman näitä määrityksiä myös testaamisen suunnittelu ja toteutus on haastavaa.

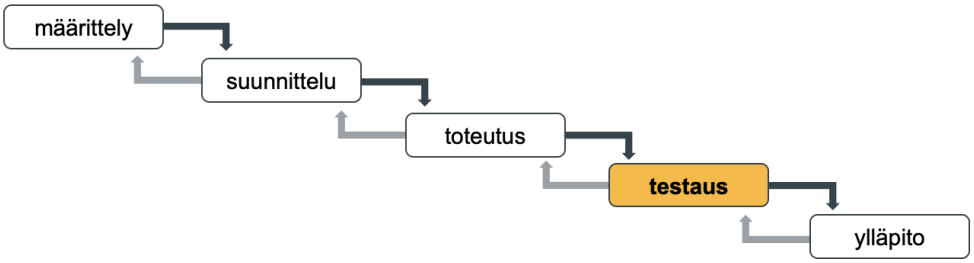
Jotta ohjelmisto voi saavuttaa sille asetetut laatuvaatimukset ja välttää ohjelmistoprojekteille tyypillisiä virheitä, virheet tulisi löytää ja korjata mahdollisimman varhaisessa vaiheessa (Wiegers 2021; Slaughter ym. 1998, 72; Ambler 2022; Myers ym. 2012, 9; Palamarchuk 2015). On arvioitu, että virhe, joka löytyy kehitysvaiheessa, on kustannuksiltaan vain 1–2 % niistä kustannuksista, jotka syntyvät, kun virhe löydetään julkaisun jälkeen (Kasurinen 2013, 13). Virheiden varhainen löytäminen nojaa pitkälti testaukseen, ja kilpailun myötä koventunut aikataulupaine on vauhdittanut etenkin automaatiotestauksen kehitystä merkittävästi (Wang ym. 2022, 2; Kumar & Mishra 2016, 9).

Testauksen toteutustavat riippuvat merkittävästi tekeillä olevasta ohjelmistosta sekä valitusta projektimallista. Ohjelmistoprojektit noudattavat joko perinteistä vesiputousmallia tai yhä useammin jotakin ketterää ohjelmistokehityksen mallia. Testauksen rooli on malleissa erilainen ja sen myötä myös testauksella saavutettavissa olevat hyödyt poikkeavat toisistaan.

3.2 Testaus perinteisissä ohjelmistoprojekteissa

Vesiputousmalli

Perinteisessä vesiputousmallissa (Kuva 1) koko projekti suunnitellaan alusta loppuun projektin alkaessa. Projekti etenee määrittelyvaiheesta suunnittelun kautta toteutusvaiheeseen, jonka valmistuttua aloitetaan järjestelmän testaus. Testauksen jälkeen järjestelmä otetaan käyttöön ja siirrytään ylläpitovaiheeseen. Mallin mukaan edellisiin vaiheisiin palaaminen on tarpeen tullen mahdollista, mutta tavoitteena on, että näin ei tarvitsisi tehdä. (Kasurinen 2013, 9–10.)



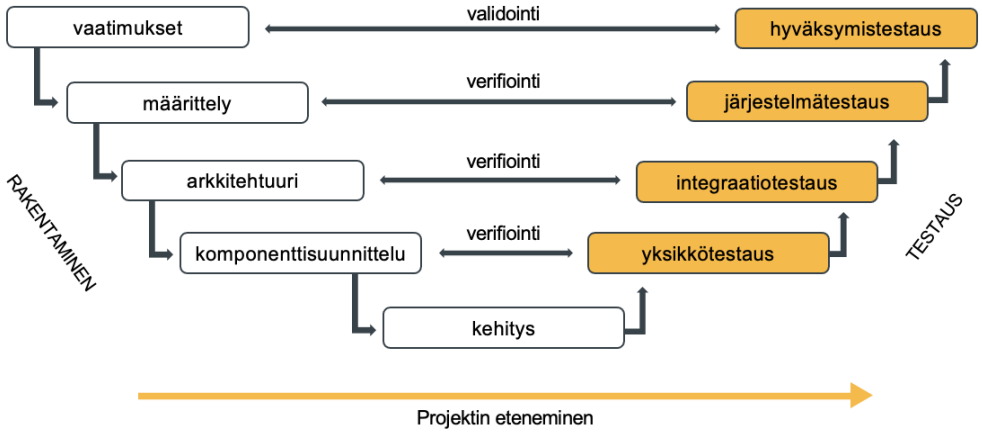
Kuva 1. Ohjelmistoprojektin eteneminen vesiputousmallissa (Kasurinen 2013, 10)

Ohjelmistoprojektin alun määrittelyvaiheessa kirjataan ylös, mitä kehitteillä olevan ohjelmiston tai järjestelmän pitäisi valmiina tehdä. Suunnitteluvaiheessa järjestelmä suunnitellaan tehtyjen määrittelyiden mukaan sekä arkkitehtuuritasolla että yksittäisten komponenttien osalta. Kun suunnitelmat on tehty, seuraa ohjelmiston kehitystyö. Sen valmistuttua aloitetaan testaaminen. Kun määritetyt testit menevät läpi, järjestelmä on valmis otettavaksi käyttöön ja siirtymään ylläpitovaiheeseen. (Adam 2021.)

Vesiputousmallin jo pitkään tunnettu ongelma on virheiden myöhäinen havaitseminen (Royce 1970). Kun ongelmat havaitaan projektin edettyä jo varsin pitkälle, on mahdollista, että virheet ovat niin kriittisiä, että niiden korjaamiseksi joudutaan palaamaan ei vain toteutusvaiheen ohjelmointiin, vaan suunnittelu- ja jopa määrittelyvaiheeseen. Jos projekti joudutaan aloittamaan osin tai jopa kokonaan lähes alusta, testauksessa löydettyjen virheiden aikataulu- ja kustannusvaikutukset kasvavat usein varsin merkittäviksi. (Wiegers 2021, Ambler 2022.) Vesiputousmalli on edelleen käytössä monissa ohjelmistokehitysprojekteissa, vaikka sen ongelmat on tunnistettu jo kauan sitten.

Testauksen V-malli

Testaaminen pitää sisällään ohjelmiston eri osakokonaisuuksiin kohdistuvia testejä. Nämä eri testauksen tasot esitetään usein nk. V-mallin (Kuva 2) avulla. Mallista on olemassa erilaisia variaatioita ja tulkintoja (esim. Adam 2021; Kasurinen 2013, 10), mutta niille on yhteistä, että projekti etenee vesiputousmallia noudattaen projektin alussa tehdyn suunnitelman mukaisesti.



Kuva 2. Testauksen V-malli (Adam 2021; Kasurinen 2013, 10)

Mallin vasemmalla puolella kuvataan projektin etenemistä alun käyttäjävaatimusten pohjalta tehdyistä määrittelyistä järjestelmän arkkitehtuuri- ja komponenttisuunnittelun kautta varsinaiseen kehitystyöhön. Mallin oikealla puolella esitetään testauksen eri tasot sen mukaan, mitä ohjelmistokokonaisuuden osaa niillä testataan. (Adam 2021; Kasurinen 2013, 10.)

V-mallin ansiona on pidetty sen selkeyttä ja yksinkertaisuutta sekä tapaa asettaa testaus kiinteäksi osaksi ohjelmiston kehitysprosessia (Adam 2021). Vesiputousmallia ja testauksen V-mallia on kuitenkin kritisoitu voimakkaasti niiden lähtöoletuksesta, jonka mukaan projektin kaikki vaiheet pystytään suunnittelemaan projektin alussa riittävällä tarkkuudella. Koska etenkin laajoissa ja pitkäkestoisissa ohjelmistokehitysprosesseissa näin ei käytännössä koskaan ole, projektin edetessä ilmeneviä tekijöitä on haastavaa ottaa huomioon ja mukauttaa projektia niiden edellyttämällä tavalla, kun etenemistä määrittää tiukka projektisuunnitelma. (Pries & Quigley 2010; Adam 2021, The Standish Group 2015.)

3.3 Testaus nykyaikaisissa ohjelmistoprojekteissa

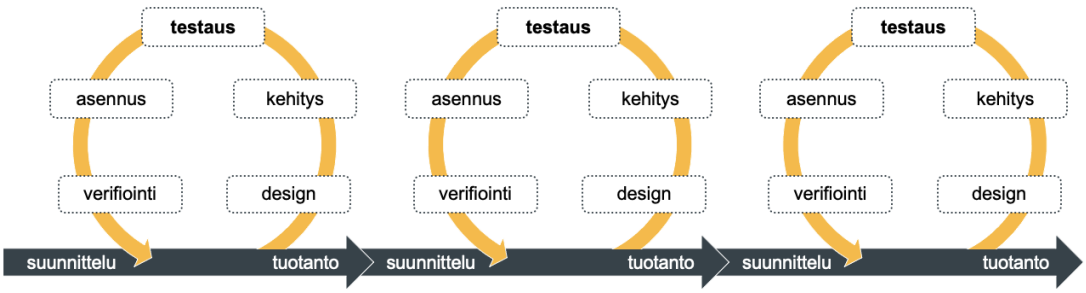
Nykyaikainen ohjelmistokehitysprosessi käyttää yhä useammin vesiputousmallin sijasta ketteriä ohjelmistokehityksen menetelmiä kuten Scrum. Toisin kuin perinteisissä malleissa, ketterissä ohjelmistokehitysmalleissa kehitystyö

pilkotaan pienempiin osiin, työn suunnittelua ja ohjelmiston määrittelyä tehdään iteratiivisesti, ja myös testausta suoritetaan kehitystyön osana alusta lähtien ja jatkuvasti. Testauksella on menetelmissä usein keskeinen portinvartija-rooli: työ etenee vaiheesta seuraavaan vasta, kun edellinen vaihe on testattu ja verifioitu. (Adam 2022.)

Yhtenä keskeisenä menestystekijänä ketterille menetelmille ovat olleet itse ketterien viitekehysten lisäksi projekteissa usein käytössä olevat lyhyeen palauteketjuun ja vahvaan testaukseen pohjautuvat käytänteet kuten testauslähtöinen kehittäminen (*test-driven development*) ja jatkuva koodin integrointi (*continuous integration*) (Ambler 2022, Luukkainen 2022). Seuraavassa esitetään Scrum-mallin perusperiaatteet sekä tässä mainitut hyvät käytänteet, joilla pyritään takamaan ketterillä menetelmillä kehitettyjen ohjelmistojen korkea laatu.

Scrum

Scrum-viitekehyksessä projektia edistetään 2–3 viikon mittaisissa kehitysiteraatioissa eli sprinteissä. Kehitysprojektia tehdään sprintti kerrallaan, ja jokaisessa sprintissä toteutetaan jokin ohjelmiston osakokonaisuus. Sprintti sisältää myös kyseisen ohjelmiston osan vaatimien testien suunnittelun ja toteutuksen (Kuva 3). Näin toimivan ohjelmiston edellyttämä testaus on sisäänrakennettu projektimalliin, ja eritasoiset testit tuloksineen ovat käytettävissä kumulatiivisesti projektin edetessä. (Pries & Quigley 2010, 103–104; Adam 2022.) Testauksen ja verifioinnin sisällyttäminen ohjelmiston komponenttien valmistamisprosessiin nähdään yleisesti keskeisenä ohjelmistokehityksen laatutekijänä (Wallace & Reeker 2001, 182).



Kuva 3. Iteratiiviset sprintit Scrum-mallissa (Adam 2022)

Sprintin aluksi Scrum-tiimi suunnittelee yhdessä sprintillä tehtävän työn suunnittelukokouksessa. Tämän jälkeen kehitystiimi kokoontuu päivittäin lyhyeen päivittäispalaveriin tarkastelemaan työn tilannetta ja etenemistä. Sprintin edetessä suunnitellut ohjelmiston osat kehitetään, testataan, asennetaan ja niiden toimivuus verifioidaan. Sprintin päättyessä järjestetään katselmointi, jossa sprintin tulokset käydään läpi yhdessä Scrum-tiimin ja projektin tärkeimpien sidosryhmien kanssa. Tämän lisäksi Scrum-tiimi pitää sisäisen retrospektiivin, jonka tavoitteena on sopia, miten toimintaa tai työkaluja tulisi kehittää, jotta tulevat sprintit sujuisivat paremmin. Seuraavan sprintin alussa työtä suunnitellaan eteenpäin aiempien sprinttien kerryttämän tiedon pohjalta. (Scrum-opas 2020, 8–10.)

Jatkuvan testauksen hyviä käytänteitä

Ketterien ohjelmistoprojektien yhteydessä on usein käytössä hyväksi havaittuja käytänteitä, joiden avulla edellä kuvatusta työskentelytavasta saadaan entistä tehokkaampaa, kun tuotettu koodi on lähes välittömästi niin luotettavaa, että se voidaan ottaa käyttöön.

Testauslähtöisessä kehittämisessä automaatiotestit kirjoitetaan rinta rinnan varsinaisen ohjelmakoodin kanssa siten, että ennen kutakin uutta ohjelmakoodin osaa toteutetaan sen testaava koodi. Kun ohjelmiston vaatimusten pohjalta laadittu testi on valmis ja on varmistettu, että se ei vielä mene läpi, kirjoitetaan varsinainen ohjelman osa, jonka tavoitteena on saada testi läpäistyä. Viimeisessä vaiheessa koodi refaktoroidaan, eli parannetaan ja siistitään sen rakennetta ilman, että varsinainen toiminnallisuus muuttuu. Refaktoroinnin tarkoituksena on tehdä seuraavien toiminnallisuuksien ohjelmoinnista helpompaa. Refaktoroinnin voi tehdä ilman pelkoa siitä, että toiminnallisuus menee huomaamatta rikki, koska testi paljastaa nämä tilanteet. (Steinfeld 2022.)

Jatkuva koodin integrointi on menetelmä, jossa tavoitteena on viedä jokainen tehty muutos mahdollisimman nopeasti osaksi tuotantokoodia. Käytännössä tätä menetelmää käytetään usein testauslähtöisen kehittämisen jatkona, eli kun testit menevät läpi ja koodi on refaktoroitu, uusi toiminnallisuus liitetään välittömästi ohjelmiston koodikokonaisuuteen. (Steinfeld 2022.)

Kaiken kaikkiaan ketterien ohjelmistoprojektien on todettu onnistuvan keskimäärin lähes neljä kertaa useammin kuin vesiputousmallilla toteutettujen projektien. Ketterät menetelmät ovat osoittautuneet sitä hyödyllisemmiksi mitä suuremmista ja monimutkaisemmista ohjelmistoprojekteista on kyse: suurissa projekteissa onnistuneiden ketterien hankkeiden määrä on jopa kuusinkertainen verrattuna vesiputoushankkeisiin. (The Standish Group 2015.)

4 TESTAUKSEN SUUNNITTELUN VIITEKEHYS

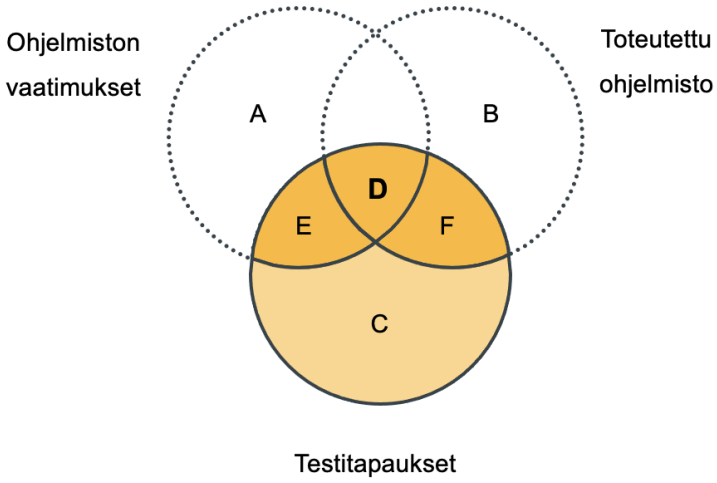
Jokainen ohjelmistoprojekti on erilainen ja vaatii tapauskohtaisen testaussuunnitelman. Suunnitelman laatimista varten on tunnettava ohjelmiston käyttötarkoitus, sille asetetut laatutavoitteet, valitut teknologiaratkaisut, projektimalli ja siinä käytettävissä olevat resurssit sekä erilaisten testausmenetelmien ja -työkalujen soveltuvuus erilaisiin testaustavoitteisiin. Nämä tiedot tarvitaan, jotta voidaan suunnitella juuri kyseessä olevalle ohjelmistokokonaisuudelle sopivin mahdollinen testauskokonaisuus.

4.1 Testauksen kohdentaminen

Ohjelmistolle asetetut vaatimukset ovat sekä ohjelmiston ohjelmoinnin että sen testaamisen osalta keskeisiä. Ideaalitilanteessa ohjelmisto vastaisi täydellisesti asetettuja vaatimuksia ja tehdyt testit kattaisivat jollakin tasolla kaikki vaaditut ominaisuudet. Priorisoinnin ja muiden kehityksen aikaisten tekijöiden seurauksena ohjelmistoissa on kuitenkin usein lopulta sekä vaadittuja ominaisuuksia että ei-vaadittuja ominaisuuksia, minkä lisäksi osa vaadituista ominaisuuksista voi puuttua. Testit voivat kattaa näitä ominaisuuksia eri tavoin, ja testauksen suunnittelussa on tärkeää tietää, mihin osaan ohjelmistoa suunniteltu testaus kohdentuu ja mitä se ei kata. (Jorgensen 2014, 5–10; Myers ym. 2012, 7–8.)

Kuva 4 ympyrä A kuvaa ohjelmistolle asetettuja vaatimuksia, ympyrä B toteutettua ohjelmistoa ja ympyrä C ohjelmistolle laadittuja testitapauksia. Kuvion keskellä oleva alue D kuvaa sitä osaa toteutetusta ohjelmistosta, joka vastaa

vaatimusmäärittelyä ja joka on katettu testeillä. Onnistuneessa ohjelmistoprojektissa tämä osuus on mahdollisimman suuri. (Jorgensen 2014, 5–6.)



Kuva 4. Ohjelmiston vaatimukset, toteutettu ohjelmisto sekä testikattavuus (Jorgensen 2014, 3–4)

Testauksen näkökulmasta kuviossa on paljon huomionarvoista. Alue E kuvaa vaatimusten mukaisia ominaisuuksia, joita ei ole toteutettu, mutta jonka testit kattavat. Testien avulla nämä puuttuvat ominaisuudet voidaan löytää, mutta alueella A sijaitsevat ominaisuudet, joille ei ole suunniteltu testejä, jäävät testituloksissa huomaamatta.

Alue F kuvaa niitä määritysvaatimusten ulkopuolisia ominaisuuksia, jotka on toteutettu ja joita testataan. Vaikka ominaisuuksia ei olisi vaatimusten mukaan tarvinnut toteuttaa, testien avulla niissä olevat ongelmat voidaan havaita ja niiden mahdolliset vaikutukset muihin ohjelman osiin ovat paremmin hallittavissa. Alueella B sijaitsevat ominaisuudet, jotka on toteutettu, vaikka ne eivät ole vaatimuksissa. Koska testit eivät kata näitä ominaisuuksia, ne voivat tarpeettomuutensa lisäksi aiheuttaa ohjelmistossa ongelmia esimerkiksi lisäämällä monimutkaisuutta ja siten vaikeuttamalla ohjelmiston ylläpitoa.

Testien valinnassa voidaan käyttää sekä vaatimusmäärittelyä että toteutettua ohjelmakoodia lähtökohtana. Kuitenkin molempia tarvitaan. Jos suunnitelma tehtäisiin yksinomaan ohjelmakoodin pohjalta, vain toteutetut ominaisuudet voivat päätyä testattaviksi, jolloin vaatimusmäärittelyn toteutumista ei tutkita. Toisaalta, jos testit laaditaan yksinomaan vaatimusmäärittelyiden pohjalta,

kaikki ohjelmassa niiden lisäksi oleva jäisi testien kattamattomiin. (Jorgensen 2014, 9–10.)

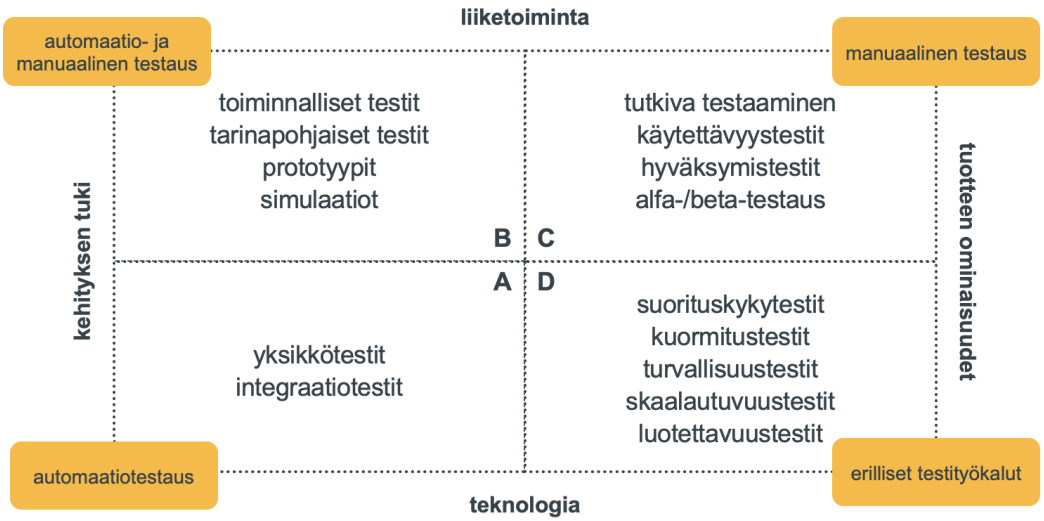
Muita tekijöitä, jotka kannattaa pitää mielessä suunniteltaessa mitä testataan, ovat esimerkiksi kaikki uusi koodi, uusien teknologioiden yhdistäminen ohjelmakokonaisuuteen, muualta tuotu koodi sekä uusien, ohjelmakokonaisuutta heikommin tuntevien työntekijöiden panos (Kasurinen 2013, 77). Lisäksi ohjelmistoa rakennettaessa löydettyjen ja vielä löytämättömien virheiden määrät korreloivat keskenään. Toisin sanoen ohjelmiston osa, josta on jo löytynyt runsaasti virheitä, pitää todennäköisemmin sisällään enemmän virheitä kuin sellainen osa, jota on testattu mutta josta virheitä ei ole löytynyt. (Myers ym. 2012, 17; Kasurinen 2013, 77.)

Seuraavissa luvuissa tarkastellaan testikokonaisuuden rakentamista niin, että se kattaisi ohjelmiston eri osat mahdollisimman monipuolisesti.

4.2 Testimenetelmien valinta

Edellä on pohdittu, *miksi* testaus on tärkeä osa ohjelmistokehitystyötä (luku 3) ja *mitä* testauksella pyritään tarkastelemaan (luku 4.1). Ohjelmiston mahdollisimman hyvän testikokonaisuuden laatiminen vaatii käsillä olevan ohjelmiston tuntemusta sekä taitoa arvioida eri testausvaihtoehtojen tarjoamia mahdollisuuksia ja riskejä. Seuraavassa esitetään katsaus siitä, *miten* testikokonaisuutta voidaan hahmottaa parhaiden testimenetelmien valitsemiseksi.

Kattavan testikokonaisuuden laatimiseksi ohjelmistoa voidaan tarkastella monista eri näkökulmista. Kuva 5 nelikentässä (Cruzes ym. 2017) esitetään erilaisia testauksen tapoja sen mukaan, onko testauksen tavoitteena enemmän kehitystyön vai tuotteen kehittäminen (vaaka-akseli) sekä sen mukaan, onko testien kohteena liiketoiminnan vai teknologioiden parantaminen (pystyakseli). Mallin ansiona on pidetty erityisesti sitä, että se nostaa esille testauksen roolin myös kehitystyön tukena (kentät A ja B) eikä ainoastaan tuotteen ominaisuuksien parantamisessa (C ja D).



Kuva 5. Testausvaihtoehtoja luokiteltuna sen mukaan, kohdentuvatko ne enemmän kehitystyön tukeen vai tuotteen ominaisuuksiin sekä sen mukaan, onko tavoitteena enemmän liiketoiminnan vai teknologian kehittäminen (Cruzes ym. 2017)

Kentän A yksikkö- ja integraatiotestit paljastavat ohjelmointivirheitä ja toimivat näin kehityksen tukena. Nämä testit toteutetaan usein automaatiota hyödyntäen. Myös erilaiset ohjelmiston toimintaa tarkastelevat testit (B) kuten erilaiset toiminnalliset testit (onnistuuko ohjelman käyttö oletetulla tavalla), käyttäjätarinoihin pohjautuva testaus (vastaako ohjelmisto käyttäjän määrittelemää käyttötarvetta), prototyyppien ja simulaatioiden avulla tapahtuva testaus toimivat kehitystiimin tukena. Näitä testejä voidaan toteuttaa sekä manuaalisesti että automaatiotesteinä.

Tuotteen ominaisuuksiin kohdentuvaa testausta voidaan liiketoiminnan näkökulmasta (C) kehittää parhaiten erilaisin manuaalisin, mm. tutkivan testaamisen menetelmin. Myös käytettävyys- ja hyväksymistestaus testaavat ohjelmiston ominaisuuksia liiketoimintahyötyjen näkökulmasta. Alfa- ja beta-testauksella tarkoitetaan eri ohjelmistoversioiden liiketoimintahyötyjen testausta.

Ohjelmiston suorituskykyä, kuormituksenkestoa, turvallisuutta, skaalautuvuutta ja luotettavuutta mittaavat testit (D) ovat hyvin teknologiakeskeisiä ja niitä varten käytetään useimmiten niihin erityisesti tarkoitettuja erillisiä työka-

luja. Kaikkia näistä elementeistä ei tarvita jokaisessa projektissa (esim. kuormitustestausta, jos ohjelmalla on vain vähän käyttäjiä), mutta ohjelmiston turvallisuuden ja luotettavuuden takaamiseen liittyvää laadunvarmistusta tulisi olla mukana jokaisen ohjelmistoprojektin testisalkussa etenkin, jos käytössä on avoimella lähdekoodilla tuotettuja ohjelmistoja tai kirjastoja (Contrast Security 2014).

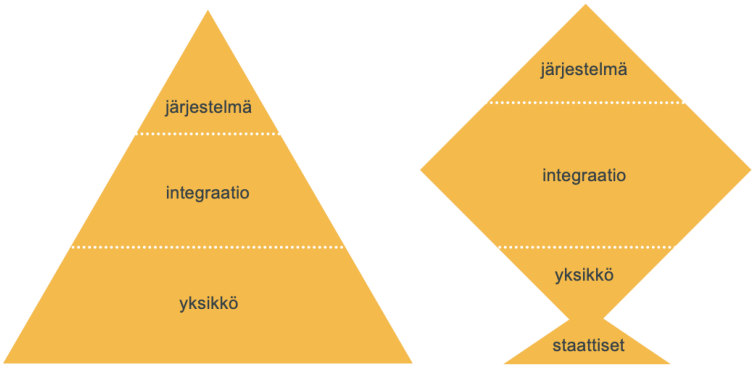
Nelikenttää voi käyttää minkä tahansa ohjelmistoprojektin testauskokonaisuuden suunnittelun tukena. Se auttaa menetelmien tasapainoisessa valitsemisessa kulloisenkin ohjelmistoprojektin tarpeisiin. Koska kaikkien mahdollisten virheiden varalta testaaminen on mahdotonta, olennaista testaustyyppien valinnassa on, että erilaiset testit toimivat erilaisissa tilanteissa ja niitä yhdistelemällä voidaan parantaa testauksen kattavuutta ja tehokkuutta virheiden löytämisessä (Myers ym. 2012, 11–12).

4.3 Testiautomaation optimointi

Testikokonaisuuden suunnittelussa on pohdittava, miten eri tasoilla tapahtuvaa testausta tulisi painottaa, jotta testauksesta saatava kokonaishyöty, eli käyttäjän kokema arvo ja ohjelmiston laatu, olisi mahdollisimman suuri. Koska testiautomaation nähdään olevan erityisesti ketterissä ohjelmistoprojekteissa onnistumisen kannalta keskeinen, mutta myös haastava työväline (Wang ym. 2022, 3; Palamarchuk 2015; Kumar & Mishra 2016, 10), on syytä tarkastella hieman tarkemmin, miten testiautomaatio kannattaa suunnitella, jotta se palvelee kokonaisuutta mahdollisimman hyvin.

Kuten aiemmin on kuvattu, testausta voidaan suorittaa ohjelmiston eri tasoilla ohjelman yksittäisistä osista (yksikkötestaus) tai niiden yhdistelmistä (integraatiotestaus) koko järjestelmän niin kutsuttuun end-to-end-testaukseen, jossa ohjelmiston toimintaa testataan toiminnallisena kokonaisuutena (järjestelmätestaus). Lisäksi yleensä ohjelmiston valmistuessa tehdään niin kutsuttua hyväksymistestausta, jossa joko ohjelmiston tilaaja tai tilaaja ja käyttäjät yhdessä testaavat sen toimivuutta todellisista käyttötilannetta vastaavissa olosuhteissa. Kullakin tasolla testausta voidaan suorittaa eri tavoin. (Vala 2022.)

Edellä ollut Kuva 5 (s. 19) nelikenttä esittää, miten automaatio- ja manuaalitestaus tyypillisesti sopivat erilaisiin testaustarpeisiin. Testiautomaation asiantuntijoilla on lisäksi erilaisia näkemyksiä siitä, missä automaatiotesteistä saatava hyöty on parhaimmillaan. Kaksi kilpailevaa päämallia ovat Kuva 6 esitetyt, niin kutsutut testipyramidi- ja testipokaalimalli. Vasemmalla esitetyssä testipyramidissa automaatiotestien pääpaino on yksikkötesteissä, kun taas oikealla olevassa pokaalimallissa painotus on integraatiotesteissä (Molina 2021, luku 1).



Kuva 6. Automaatiotestauksen painotukset testipyramidi- ja testipokaalimalleissa (Molina 2021, luku 1)

Pyramidimallin perusajatus on, että yksikkötestit ovat nopeita, edullisia ja kohtuullisen yksinkertaisia toteuttaa. Ne kohdentuvat usein yksittäisiin komponentteihin tai moduuleihin ja antavat siten tarkkaa tietoa havaittujen ongelmien laadusta. Kun ohjelmiston osat toimivat hyvin ja niiden toiminta varmistetaan kattavilla testitapauksilla, ylempien tasojen usein hitaampia ja kalliimpia testejä tarvitaan vähemmän. Tyypillisesti yksikkötestit toteutetaan testiautomaatiolla ja ne kohdistuvat johonkin tiettyyn ohjelmistokomponenttiin tai moduuliin. Testien kohteen monimutkaistuessa myös testien suunnittelu ja toteuttaminen on vaikeampaa, jolloin näiden testitasojen oletetusti vähäisempi tarve puolustaa paikkaansa. (Molina 2021, luku 1; Palamarchuk 2015.)

Pokaalimallissa automaatiotestien paino on integraatiotesteissä. Mallin mukaan integraatiotestien hyötysuhde nähdään parhaana, koska niitä tarvitaan lukumääräisesti vähemmän, mutta ovat edelleen suhteellisen nopeita toteuttaa ja tunnistavat virheitä kattavasti. Integraatiotestit kattavat välillisesti suuren osan järjestelmästä. (Molina 2021, luku 1.)

Pokaalimalli seisoo jalustalla, jonka muodostavat ns. staattiset testit. Näitä ovat erilaiset koodin kirjoittamista helpottavat työkalut, kuten tyypitystyökalut ja lintterit, jotka kertovat ohjelmoijalle syntaktisista virheistä automaattisesti heti koodia kirjoittaessa. (Molina 2021, luku 1.) Staattisten työkalujen hyöty riippuu käytetystä ohjelmointikielestä, ja ne ovat avuksi virheiden havaitsemisessa ns. dynaamisesti tyypitettyjen kielten kuten JavaScript tai Python kanssa (Dodds 2021).

Kaiken kaikkiaan testauksen tasojen määrittelyt ja niille annettava sopivin painoarvo ovat esitettyjen mallien ja niiden pohjalta käydyn keskustelun perusteella häilyviä. Määrittelyitä on lähes yhtä monia kuin määrittelijöitäkin. Esimerkiksi Fowler (2021) on pohtinut mikä on ”yksikön” määritelmä ja milloin voidaan perustellusti puhua yksiköiden välisestä ”integraatiosta”. Testiautomaation ja manuaalisen testauksen painotus riippuu pitkälti toteutettavasta ohjelmistosta ja projektin vaatimuksista.

4.4 Testaustyökalut

Testaukseen käytettävät työkalut ovat kehittyneet ohjelmistoliiketoiminnan kehityksen myötä. Testaustyökaluja on saatavilla runsaasti sekä kaupallisina tuotteina että avoimen lähdekoodin ratkaisuina. Koska työkalut kehittyvät nopeasti ja testauksen tarve vaihtelee projekteittain, työkalujen valinta on tehtävä aina projektikohtaisesti sen mukaan, mikä on kussakin projektissa tarkoituksenmukaista. Työkaluja voidaan luokitella muun muassa käyttötarkoituksen (esim. ketterä, automaatio- tai mobiilitestaus), testitasojen, lisenssiehtojen tai teknologioiden mukaan (Woke 2022).

Eri ohjelmistotasojen ja testityyppien lisäksi on olemassa myös runsas joukko erilaisia testien hallinnointiin suunniteltuja työkaluja, jotka auttavat suunnittelemaan ja aikatauluttamaan testejä sekä dokumentoimaan ja jakamaan niistä saatavaa tietoa. Yksinkertaisimmillaan ja etenkin pienissä ohjelmistoprojekteissa näitä tehtäviä voi hoitaa myös yksinkertaisen taulukko-ohjelman (esim. Google Sheets) avulla.

5 TESTAUSVIITEKEHYS JA PROJEKTIKOHTAINEN TESTAUSSUUNNITELMA

Gispon ohjelmistoprojektien koko on kasvanut viime vuosina, ja testaus on lisännyt merkitystään osana ohjelmistojen laadunvarmistusta. Yritys toivoi siksi saavansa testauksen tueksi viitekehyksen, joka helpottaa testauksen suunnittelua kokonaisvaltaisesti, kiinteänä osana ohjelmistoprojekteja.

Tässä luvussa esitetään viitekehys ohjelmistotestauksen suunnitteluun, jota Gispo voi hyödyntää paikkatietoalan ohjelmistoprojekteissaan testauskokonaisuuden suunnittelun tukena. Lisäksi esitetään viitekehyksen pohjalta koostettu alustava projektikohtainen testaussuunnitelma Lahden seudun jätehuoltoviranomaiselle toteutettavaa asiakasprojektia varten. Suunnitelma havainnollistaa testausvalintojen tekemistä ja testaussuunnitelman iteratiivista luonnetta ketterässä projektimallissa. Viitekehyksen ja sen pohjalta tehtävien suunnitelmien yhteisenä tavoitteena on varmistaa, että Gispon kehittämät sovellukset toimivat vakaasti ja halutulla tavalla sekä niiden kehitys- että tuotantovaiheissa.

5.1 Testauksen viitekehys Gispolle

Projektin testiviitekehys voidaan rakentaa, kun tunnetaan projektin tavoitteet, projektimalli ja rakenteilla olevan ohjelmiston yleiset ominaisuudet. Näiden tietojen pohjalta voidaan hahmotella ohjelmiston testaukseen vaadittavat testitasot ja lopulta niille sopivat testausmenetelmät sekä -työkalut.

5.1.1 Projektimallin vaikutus testaukseen

Jotta testaus muodostaisi kiinteän osan ohjelmistoprojektia, se tulee huomioida jo projektin varhaisessa vaiheessa. Tämä voidaan varmistaa pohtimalla projektin testauksen laajuutta heti, kun projektin tavoitteiden määrittely ja niiden pohjalta sopivimpien pääteknologioiden valinnat on tehty.

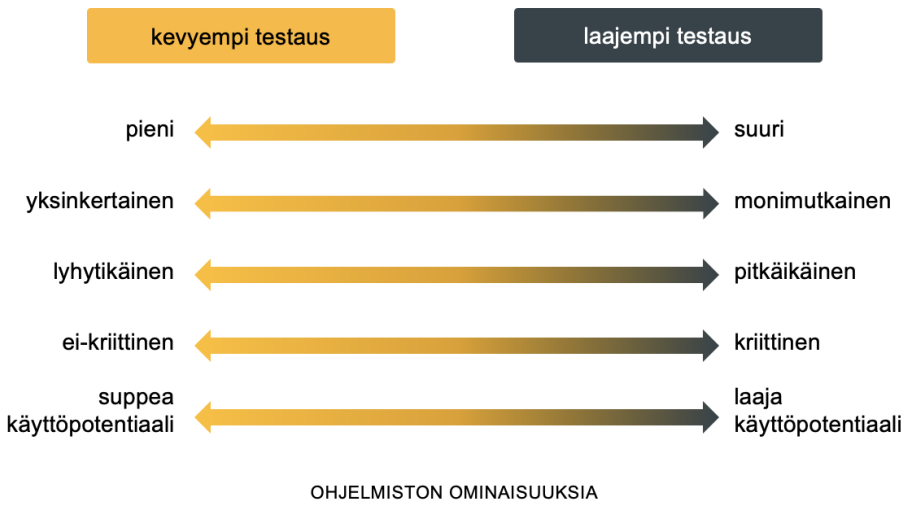
Projektia koskevan määrittelyn yksityiskohtien taso riippuu valitusta projektimallista (vesiputous vs. ketterä malli, ks. luvut 3.2 ja 3.3). Vesiputousmallissa tavoite, projektisuunnitelman vaiheet, käytettävät työkalut sekä validointimene-

telmät määritetään varsin tarkasti projektin alussa, kun taas ketterissä malleissa päätavoite sovitaan pääpiirteisesti, mutta prosessi ja työkalut sinne pääsemiseksi jätetään täsmennettäväksi projektin aikana tiedon lisääntyessä.

Gispon asiakasprojekteissa on ollut vaikutteita molemmista mainituista malleista. Vesiputousmalliin liittyvät haasteet on tunnistettu ja varsinkin isommissa projekteissa suositaan yhä useammin ketteriä lähestymistapoja, joista eniten vaikutteita on otettu Scrum-mallista. Scrumin lisäksi myös testauslähtöisen kehittämisen sekä jatkuvan integroinnin käytännöt voisivat tehostaa ohjelmointityötä ja hyödyttää etenkin yrityksen isompia projekteja. Koska Scrum-mallissa projektin määrittelyt täsmentyvät projektin edetessä, myös testauksen menetelmät voidaan projektin alussa suunnitella vain pääpiirteittäin, ja niitä tulee täsmentää projektin edetessä.

5.1.2 Testauksen laajuuteen vaikuttavat ohjelmiston ominaisuudet

Jokainen ohjelmistoprojekti on omanlaisensa ja vaatii tapauskohtaista harkintaa sen osalta, miten laajaa testausta ohjelmisto edellyttää. Kuvassa Kuva 7 esitetään ohjelmiston ominaisuuksia, joilla on vaikutusta siihen, miten laajaa ja kattavaa testausta ohjelmistoon on hyödyllistä suunnitella. Pääsääntönä voidaan pitää, että mitä pienemmästä ja yksinkertaisemmasta ohjelmistosta on kyse, sitä suppeammalla testauksella voidaan varmistaa, että ohjelmisto täyttää sille asetetut vaatimukset ja toimii suunnitellusti.



Kuva 7. Testauslähestymistavan laajuuteen vaikuttavia ohjelmiston ominaisuuksia

Laajemmissa ja kriittisemmissä projekteissa testaus on tärkeämpää kuin yksinkertaisissa ja pienissä projekteissa. Laajuudella voidaan tarkoittaa ohjelmiston käytön elinkaarta sekä käyttöpotentiaalia. Ohjelmisto, jolle kaavaillaan pitkää käyttöikää ja laajaa käyttäjäkuntaa edellyttää, että sitä voidaan käyttää, ylläpitää ja jatkokehittää mahdollisimman vaivattomasti. Tällöin on tärkeää, että testit kattavat ohjelmiston osat laajasti ja käytön tai jatkokehityksen aikana havaitut virheet saadaan helposti kiinni ja korjattua. Jos taas ohjelmisto on toteutettu pienelle käyttäjäkunnalle tai vain lyhytikäistä käyttöä varten, kevyempi käyttäjätestaus esimerkiksi tutkivan testauksen menetelmillä saattaa olla riittävä, eikä automaatiotestausta tarvita välttämättä lainkaan.

Gispon liiketoimintamallissa avoimuus on keskeinen tekijä, ja toteutetut ohjelmistot tai niiden osat jaetaan avoimena lähdekoodina yrityksen GitHub-tilillä aina, jos ei ole erityistä syytä olla toimimatta näin. Avoimuuden taustalla on ajatus siitä, että jo tehty työ voi hyödyttää jotakin toista, jolla on samoja tai samankaltaisia käyttötarpeita ohjelmistolle. Gispon työn voi siis yleisesti ottaen ottaa sellaisenaan käyttöön, tai sitä voi jatkojalostaa haluamallaan tavalla. Tästä näkökulmasta Gispon ohjelmointityö pitää aina sisällään laajan käyttöpotentiaalin, mikä tulisi ottaa huomioon ohjelmistojen testausta suunniteltaessa.

Ohjelmiston kriittisyydellä tarkoitetaan ohjelmistossa olevien virheiden mahdollisesti aiheuttaman haitan vakavuutta. Esimerkiksi viranomaisen työkaluksi toteutettava järjestelmä voidaan nähdä kriittisempänä kuin vaikka virkistyskäyttöön tarkoitettu retkeilyopas.

5.1.3 Testauksen menetelmiä ja työkaluja

Gispon kehittämissä ohjelmistoissa on usein taustalla paikkatietoa sisältävä PostgreSQL-tietokanta PostGIS-laajennoksella, ja ne käyttävät käyttöliittymänään avoimen lähdekoodin QGIS-paikkatieto-ohjelmistoa sekä hyödyntävät muutenkin avoimesti saatavilla olevia työkaluja. Toinen tyypillinen ohjelmistokokonaisuus, joita Gispo tuottaa, ovat QGIS-ohjelmiston lisäosat. Yleisimmin käytetty ohjelmointikieli on Python, ja selainpohjaisissa ratkaisuissa on käytetty muun muassa TypeScriptiä.

Testaustyökaluja on saatavilla runsaasti ja hyvin erilaisiin testaustarpeisiin. Taulukko 1 on esitetty valikoima testauksen työkaluja sen mukaan, minkälainen testaustarve niillä voitaisiin kattaa Gispon tyypillisissä ohjelmistoprojekteissa. Testauskokonaisuutta suunniteltaessa on hyvä pitää mielessä myös testauksen kohdentuminen ja testien kattavuus suhteessa koko ohjelmistoon ja sen määriteltyihin tavoitteisiin. Näitä on käsitelty kappaleessa 4.1.

Taulukossa olevat työkalut on jaoteltu kappaleessa 4.3 esitellyn pokaalimallin (Kuva 6, s. 21) mukaisille tasoille: staattinen, yksikkö, integraatio ja järjestelmä. Lisäksi mukaan on otettu työkaluja myös osalle kappaleessa 4.2 esitetyssä nelikentässä (Kuva 5, s. 19) mainituista testausmenetelmistä sen mukaan miten tarpeellisina ne nähdään Gispon testaustarpeiden osalta. Taulukon avulla voidaan pohtia erilaisten testausmenetelmien tarjoamia mahdollisuuksia ja valita kulloiseenkin projektiin sopivimmat työkalut.

Taulukko 1. Testauksen työkaluja, joita Gispon ohjelmistoprojektit voivat hyödyntää

Testauksen taso	**Avoimia työkaluvaihtoehtoja**	**Lisätiedot**
Staattinen	• mypy • SQL-rajoitteet (*constraints*)	Mypy auttaa tunnistamaan syntaktisia virheitä jo koodin kirjoitusvaiheessa ennen koodin ajamista (toisin kuin esim. Pythonin typing-kirjasto, joka tarkastaa näitä vasta ajon aikana (Python Software Foundation s.a. b). Gispon useissa projekteissa käyttämä Python-kieli on dynaamisesti tyypitetty. Mypy on työkalu, jonka tarkoitus on yhdistää staattisen ja dynaamisen tyypityksen hyödyt. (Mypy 2014.) Tietokantaprojekteissa SQL-rajoitteilla voidaan tarkastaa tietokantaan menevän datan oikeellisuus ja näin validoida se.
Yksikkö	• pytest • pytest-cov • unittest (eliPyUnit) • Jest	Pythonilla kirjoitettuja ohjelmia varten on olemassa laajalti käytettyjä testikirjastoja kuten Pytest ja sen kattavuuskirjasto pytest-cov (Pytest 2015) sekä esim. Pythonin mukana tuleva unittest (Python Software Foundation s.a. a). JavaScript- ja TypeScript-ohjelmia varten yleinen ja paljon käytetty testiviitekehys on Jest, jonka avulla voi kirjoittaa mm. yksikkötestejä (Jest 2022).

Integraatio	• Robot Framework • Jest	Robot Framework on laaja-alainen avoimen lähdekoodin automaatiotyökalu, jota voi käyttää testien automatisointiin monella tasolla ja useiden eri kielten ja testityökalujen kanssa. Robot Frameworkin yksi merkittävä etu on helppo luettavuus, mikä helpottaa mm. testien dokumentointia. (Robot Framework, s.a.) Integraatiotesteissä Robot Frameworkilla voi automatisoida esim. erilaisten rajapintojen välistä testausta. Robot Framework toimii yhdessä mm. kaikkien yksikkötestien osalta mainittujen Python-kirjastojen kanssa, mitä voidaan hyödyntää integraatiotestejä tehtäessä. (Robot Framework, s.a.) Jest puolestaan tarjoaa työkalun mm. JavaScript- ja TypeScript-kielillä kirjoitettujen sovellusten osalta myös integraatiotestaukseen (Jest 2022).
Järjestelmä	Selainpohjaiset: • Robot Framework + Selenium • Playwright • Puppeteer Mobiilisovellukset: • Robot Framework + Appium Työpöytäsovellukset: • Robot Framework + PyAutoGUI	Järjestelmätestityökalu valitaan sen perusteella, onko järjestelmä selainpohjainen, mobiilisovellus vai työpöytäsovellus. Robot Framework tarjoaa yhtenäisen sateenvarjon useiden erilaisten työkalujen käyttöön ja sen alta löytyy työkaluja testata kaikenlaisia sovelluksia. Esimerkiksi sillä voi testata selainpohjaisia sovelluksia käyttämällä Seleniumia (Selenium 2022), mobiilisovelluksia käyttämällä Appiumia (Appium s.a.) ja selainsovelluksia käyttämällä PyAutoGUI:ta (PyAutoGUI s.a.). Selainpohjaisia sovelluksia voi lisäksi testata esim. Playwrightin ja Puppeteerin avulla (Playwright 2022, Puppeteer 2022). PostgreSQL-tietokannan kanssa toimiessa testausta voidaan tehdä snapshotien avulla, jolloin tietokantaa verrataan kuvaan omasta aiemmasta tilastaan (esim. Bernier 2022).
Jatkuva integraatio	• GitHub Actions	GitHub Actions on jatkuvan integraation alusta, jolla voi automatisoida ohjelmistojen testausta ja asennusta. Työkalulla voi esim. automatisoida testejä, jotka ajetaan aina kun uutta koodia viedään GitHub-repositorioon. (GitHub Docs s.a.)
Suorituskyky Kuormitus	Suorituskyky: • Jmeter Kuormitus: • Locust • Jmeter	Suorituskyvyn testaamisella voidaan varmistaa, että rakennettu järjestelmä pystyy tekemään sille asetetut tehtävät riittävän tehokkaasti eli esimerkiksi suorittamaan tietokantahakuja riittävän nopeasti. Kuormitustesteillä varmistetaan järjestelmän kyky hallita tarpeellinen määrä samanaikaista käyttöä. Selain- ja työpöytäsovelluksille voi tehdä suorituskyky- ja kuormitustestausta esim. JMeter:illä (Apache s.a.). Kuormitustestausta selainpohjaisiin web-sovelluksiin voi tehdä Python-koodin avulla mm. Locust-työkalulla (Locust s.a.).

Turvallisuus Luotettavuus	• Dependabot • OWASP Dependency-Check	Turvallisuustyökalujen avulla voidaan tarkastaa, että käytetyt lisäosat ja liitännäiset ovat ajan tasalla, eikä niihin kohdistu päivitysten puutteesta johtuvia turvallisuusriskejä. Samalla ne takaavat myös ohjelmiston toimintakyvykkyyttä eli luotettavuutta. Työkaluja näihin tarkoituksiin ovat mm. GitHub-alustalla toimiva Dependabot (Mullans 2020) ja yleinen riippuvuuksia kartoittava OWASP Dependency-Check (OWASP s.a.) Gispon ohjelmistoprojekteissa tietokantapalvelimet hallinnoidaan usein joko asiakkaan tai toisen palveluntarjoajan puolelta, jolloin esim. autentikointiin tai käyttäjäoikeuksiin liittyviä turvallisuustestejä ei ole tarpeen suorittaa.
Hyväksymistestaus	• Robot Framework • Manuaaliset menetelmät	Selainpohjaisten ohjelmien hyväksymistestit voidaan tarvittaessa automatisoida esimerkiksi käyttäjätarinakohtaisesti Robot Frameworkin avulla. Usein hyväksymistestaus tehtäneen kuitenkin manuaalisesti asiakkaan toimesta joko asiakkaan, Gispon tai yhdessä tehtyjen testausohjeiden mukaisesti.
Käytettävyystestaus	Manuaaliset menetelmät kuten: • Heuristinen arviointi • Strukturoimaton etätestaus	Sovellusten käytettävyyden testausta voidaan suorittaa tarpeen mukaan erilaisin menetelmin. Käytettävyyttä arvioidaan usein Nielsenin (2010) laatukomponenttien mukaan. Nämä ovat: käytön opittavuus, tehokkuus, muistettavuus, virheisyys sekä koettu mielihyvä. Menetelminä voidaan käyttää mm. heuristista arviointia, jossa 2–3 asiantuntijaa arvioi sovelluksen käytettävyyttä itsenäisesti ja heidän havaintonsa kootaan yhteen (Niemelä 2020). Toinen mahdollinen menetelmä on antaa käyttäjien käyttää ohjelmistoa enemmän tai vähemmän vapaasti ja pyytää heitä raportoimaan havaitsemiaan asioita ohjelmiston toiminnasta.

Taulukossa esitetyt työkalut on valittu ohjelmistojen saatavilla olevien kuvausten perusteella niin, että ne sopisivat mahdollisimman hyvin Gispon nykyisenkaltaisiin ohjelmistotestauksen tarpeisiin. Osaa työkaluista on jo hyödynnetty yrityksen ohjelmistoprojekteissa. Jokaisen työkalun soveltuvuus uuteen yksittäiseen projektiin ja sille eri testitasoille erikseen määritettyihin testitapauksiin tulee kuitenkin tarkastella tapauskohtaisesti.

Seuraavassa kappaleessa havainnollistetaan edellä esitetyn viitekehyksen pohjalta alustavien projektikohtaisten testausvalintojen tekemistä ja projektin testaussuunnitelman iteratiivista luonnetta ketterässä projektimallissa.

5.2 Projektikohtainen testaussuunnitelma

Tässä kappaleessa esitetään Gispo Oy:n testausviitekehyksen pohjalta laadittu projektikohtainen testaussuunnitelma Lahden seudun jätehuoltoviranomaiselle toteutettavaa järjestelmää varten. Suunnitelma tarjoaa ohjelmistoprojektille raamin, jonka avulla testaus muodostaa systemaattisen osan kehitysprosessia ja toimii näin ohjelmiston laadunvarmistuksen työkaluna. Koska kyseessä on ketterällä mallilla toteutettu projekti, Gispo ja asiakas sopivat projektin edetessä, miten testaussuunnitelmaa hyödynnetään projektissa.

5.2.1 Projektin esittely

Gispo toteuttaa parhaillaan Lahden seudun jätehuoltoviranomaiselle jätehuollon seurantajärjestelmää, jonka avulla viranomainen tulee seuraamaan jätteenkuljetuksen järjestämisvelvollisuuden täyttymistä toimialueellaan. Järjestelmän tavoitteena on mahdollistaa vuosina 2021–2022 voimaan astuneiden jätelain uudistusten (Jätelaki 15.7.2021/714) mukaisen jätteenkäsittelyn tehokas seuranta ja raportointi. Ohjelmiston lähdekoodi tullaan julkaisemaan Gispon avoimessa GitHub-repositoriossa.

Projektimallina projektissa on soveltuvilta osin Scrum. Kehitystyötä tehdään 2–3 viikon mittaisissa kehityssprinteissä, jotka suunnitellaan edellisen sprintin tulosten pohjalta. Projektin alkuvaiheessa on noudatettu myös projektin alussa laadittua työjärjestystä. Projektin edetessä, tietämyksen kasvaessa ja käyttötapausmäärittelyn tarkentuessa työtapa tulee todennäköisesti muuttumaan yhä aidommin iteratiiviseksi.

Testauksen rooli on projektin alkuvaiheessa ollut hyvin pieni ja sitä on tehty lähinnä manuaalisin menetelmin. Yksikkötestejä on toteutettu jonkin verran, mutta ei kattavasti. Työn edetessä ja järjestelmän monipuolistuessa testauksen rooli kasvaa. Testausta tarvitaan varmistamaan sekä kehityksenaikaista että myöhemmin käyttöönoton jälkeistä toimivuutta sekä mahdollista jatkokehitystyötä, mikä edellyttää monipuolisen testauskokonaisuuden suunnittelua.

Järjestelmän käyttötarkoitus

Järjestelmän valmistuttua jätehuoltoviranomainen voi syöttää järjestelmään uutta tietoa ja tarkastella käyttöliittymän avulla jätehuoltoalueen kiinteistöjen jätteenkuljetuksen järjestämisvelvollisuuden täyttymistä karttanäkymässä muun muassa liikennevalomallin avulla (vihreä – velvoitteet täyttyvät täysin, keltainen – velvoitteet täyttyvät osittain, punainen – velvoitteet eivät täyty lainkaan) sekä tuottaa erilaisia raportteja työnsä tueksi.

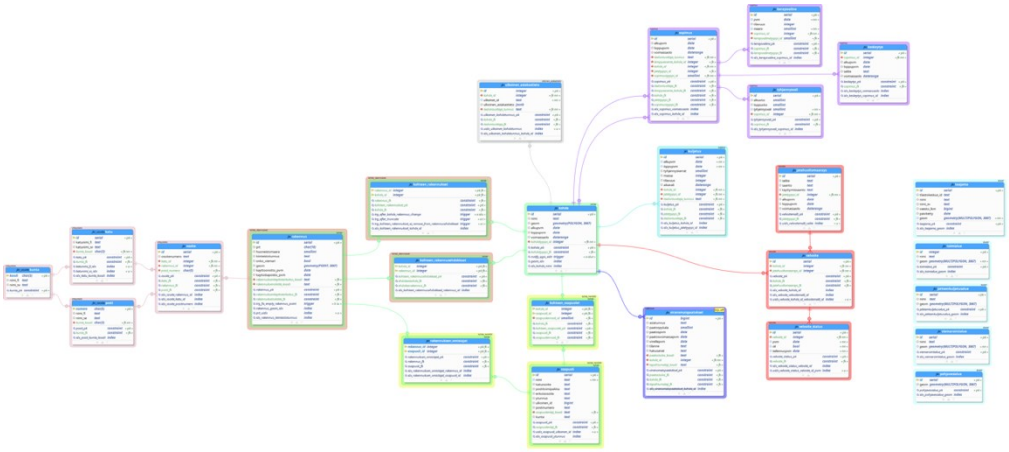
Järjestelmään kootaan tietoja eri lähteistä, kuten Digi- ja väestövirastosta, kaupungin asianhallintajärjestelmästä sekä jätteenkuljetusyrityksiltä. Tietoja yhdistelemällä järjestelmä muodostaa jätehuollolle seurantakohteita, joissa seuranta-alueen (Lahti ja kahdeksan ympäröivää kuntaa) rakennuksiin liitetään tietoja niitä koskevista jätevelvoitteista, rakennusten jätevelvollisista (esim. omistaja tai asukas) sekä rakennuksella toteutuneista jätteenkuljetuksista. Yhdistämällä tietoa siitä, millaisia jätevelvoitteita kullekin rakennukselle kohdistuu ja miten kyseisen rakennuksen jätehuolto on järjestetty, voidaan seurata täyttyvätkö voimassa olevat jätehuoltovaatimukset ja tarvittaessa ryhtyä toimenpiteisiin havaittujen ongelmien korjaamiseksi. Lisäksi kootun tiedon perusteella voidaan tarkastella esimerkiksi alueellisten kierrätystavoitteiden täyttymistä.

Jätehuoltoviranomainen päivittää jätteenkuljetusyrityksiltä saamansa kuljetustiedot sekä rakennusten omistaja- ja asukastiedot järjestelmään määräajoin. Lisäksi järjestelmään viedään säännöllisesti muita asiaankuuluvia tietoja, kuten päätöstietoja jätevelvollisuuksien päättymisestä, ilmoituksia jätteenkuljetuksen kimppasopimuksista tai kiinteistöllä tapahtuvasta kompostoinnista.

Järjestelmän tekniset ratkaisut

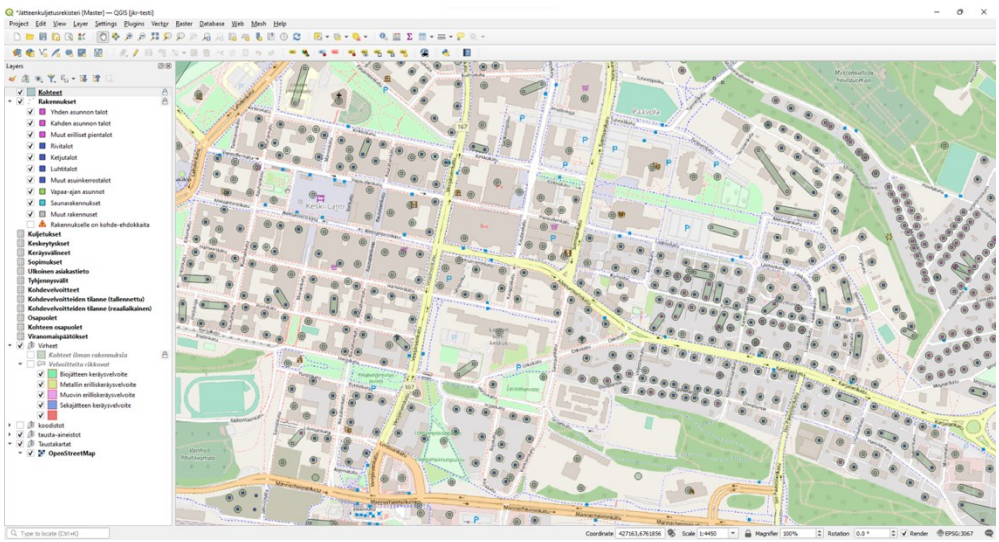
Järjestelmässä tiedot tallennetaan Azure-pilvipalvelussa sijaitsevaan PostgreSQL-tietokantaan, jonka käyttöliittymänä toimii karttapohjainen, QGIS-sovellukseen toteutettu työtila. Tietokannan hallinnointivastuu on asiakkaalla.

Tietomalli muodostuu tällä hetkellä 26 tietokantataulusta, joiden muodostamasta kokonaisuudesta saa yleiskuvan Kuva 8. Tietomalli tulee todennäköisesti vielä täydentymään kehitystyön edetessä. Tietokantaan vietävät tiedot luetaan Excel-tiedostoista ja muokataan SQL-syötteiksi Python-koodin avulla. Ylläpitovaiheen ratkaisu tietojen päivittämiselle viranomaisten toimesta on toistaiseksi suunnittelematta.

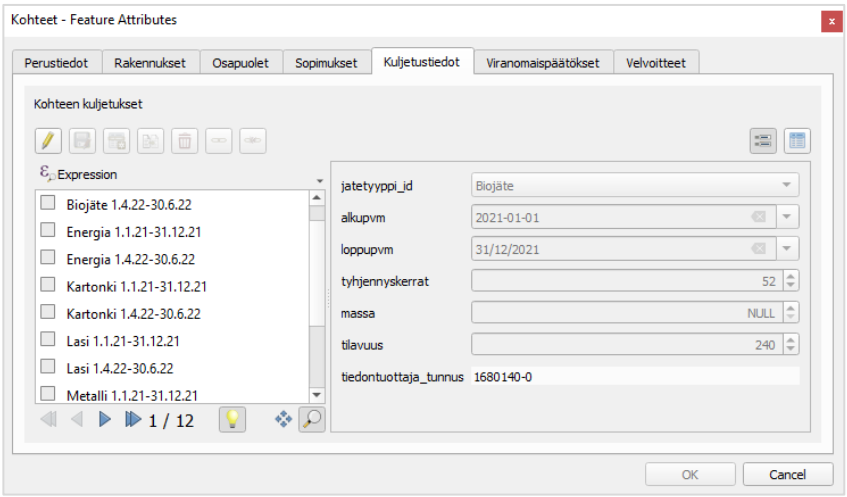


Kuva 8. Järjestelmän tietomallin yleiskuva alkuvuodesta 2023

Koska järjestelmän avulla halutaan tarkkailla jätehuollon seurantakohteita karttanäkymässä, käyttöliittymäksi on valikoitunut avoimen lähdekoodin paikkatieto-ohjelmisto QGIS. Ohjelmiston perustoimintoihin kuuluu esittää tietokannasta SQL-kyselyin haettua tietoa kartalla (Kuva 9), taulukkomuotoisena sekä lomakenäkymissä. QGIS-käyttöliittymässä voidaan tarkastella yksittäisen seurantakohteen tietoja (Kuva 10) sekä ryhmitellä ja suodattaa seurantakohteita eri tavoin.



Kuva 9. Seurantakohteita kartalla



Kuva 10. Yksittäisen seurantakohteen tietoja käyttöliittymän lomakkeella

Tarvittaessa käyttöliittymän lomakkeen kenttiä voidaan asettaa muokattavaksi, jolloin käyttöliittymän kautta voidaan tehdä muutoksia suoraan tietokantaan. Rajoituksia tiedon muokkaukselle voidaan määrittää paitsi tietokannassa (*SQL constraints*) myös QGIS-käyttöliittymässä. Nämä määritykset ovat toistaiseksi tekemättä.

5.2.2 Testauksen yleiset suuntalinjat

Jätehuollon seurantajärjestelmäprojektin ydin on monipuolisen, eri lähteistä tulevan datan saaminen tietokantaan niin, että tiedot yhdistyvät toisiinsa oikein ja pysyvät mahdollisimman ajantasaisina seurantatyötä varten. Järjestelmän koodipohja keskittyy datan tuomiseen tietokantaan oikealla tavalla, kun taas iso osa käyttöliittymässä näkyvistä toiminnoista pohjautuu QGISin tietokannasta hakeman datan visualisointiin. Näin ollen testien automaatiota on todennäköisimmin hyödyllistä toteuttaa lähinnä koodin yksikkötesteissä sekä tietokantahakuja tarkastelevien regressiotestien avulla. Muu järjestelmän testaaminen voidaan toteuttaa pitkälti manuaalisin menetelmin.

Testauksen tavoitteena on varmistaa ohjelmiston toimivuus ja tarkoituksenmukaisuus. Testaussuunnitelmaa ohjaavat seuraavat yleiset suuntalinjat:

1. Testauksen kattavuustavoitteena on testata ohjelmiston keskeiset toiminnot niin, että sekä sen kehityksen- että käytönaikainen toiminnallisuus on säilytettävissä sille määritetyllä vaatimustasolla.
2. Testit suunnitellaan niin, että sekä niiden toteuttaminen että tulosten analysointi ja hyödyntäminen on käytettävien resurssien puitteissa realistista.
3. Usein suoritettavat testit pyritään automatisoimaan. Vaikka automatisoitujen testien suunnittelu ja toteuttaminen vie aikaa, testiautomaation pitkäaikaiset hyödyt järjestelmän kehitys- ja/tai ylläpitovaiheessa maksavat itsensä takaisin varmistamalla, että järjestelmä säilyy eheänä siinä tai sen toimintaympäristössä tapahtuvista muutoksista huolimatta.
4. Testaus on toimittajan ja asiakkaan yhteinen tehtävä. Testien lopullinen valinta ja tarkat testimäärittelyt tullaan sopimaan asiakkaan kanssa projektin edetessä.

Testaussuunnitelma on elävä dokumentti, jonka avulla voidaan varmistaa, että järjestelmän eri osat toimivat suunnitellusti. Jotta suunnitelmasta saadaan projektissa mahdollisimman suuri hyöty, sitä tulee käyttää ja täsmentää, kun järjestelmään suunnitellaan ja toteutetaan uusia toimintoja.

5.2.3 Testit, aikataulu ja vastuunjako

Seuraavassa kuvaillaan jätehuollon seurantajärjestelmälle alustavasti suunnitellut testit testitasoittain. Jokaisella tasolla määritetään testeille kohde, tavoite ja aikataulu. Lisäksi esitellään testeissä käytettävät testimenetelmät, vastuunjako, eli kuka testit suunnittelee ja toteuttaa, sekä se, mitä asioita tulee ottaa huomioon kehitystyön edetessä, kun testejä valitaan toteutukseen ja suunnitellaan tarkemmin.

Järjestelmän toiminnallisuuksien perusteella järjestelmä katsotaan voitavan testata seuraavilla testitasoilla: savu-, yksikkö-, järjestelmä-, suorituskyky- ja hyväksymistestit. Integraatiotestejä ei nähdä tarpeellisena, koska järjestelmän kokonaisuudessa ei ole varsinaisia integraatioita erillisiin komponentteihin tai ulkoisiin järjestelmiin. Kaikki testit on suunniteltu alustavasti nykytiedon valossa, ja ne vaativat täsmennystä projektin edetessä tarkkojen testitapausten määrittelyä varten.

Savutestit

Savutestit (taulukko Taulukko 2) varmistavat, että järjestelmän perustoiminnot toimivat. Niiden keskeisenä tarkoituksena on todentaa järjestelmän käynnistyminen oikealla tavalla. Savutestejä on kaksi, joista ensimmäinen varmistaa tietokantayhteyden toimivuuden ja toinen tietokannan perustietojen näkymisen oikein QGIS-työtilassa.

Taulukko 2. Savutestit

	Testin kohde	Testi	Milloin ja miten tehdään
1a	Tietokanta	Tietokantayhteys muodostuu: kirjautuminen onnistuu	QGIS-työtilan avaamisen yhteydessä, manuaalinen
1b	QGIS-karttanäkymä	Tietojen luku tietokannasta onnistuu käyttöliittymässä: kohteet näkyvät kartalla	Tietokantayhteyden muodostumisen jälkeen, manuaalinen

Savutestit tulevat suoritetuksi jokaisen käyttöliittymää käyttävän henkilön toimesta aina, kun järjestelmän käyttö aloitetaan. Savutestit suoritetaan avaamalla QGIS-työtila ja muodostamalla tietokantayhteys järjestelmän käyttöohjeen mukaisesti. Jos sekä ohjelma että tietokanta toimivat odotetusti, käyttöliittymän karttanäkymässä näkyvät jätehuollon seurantakohteet. Tarkempi aloitusnäkymä tulee määrittää projektin käyttöönottovaiheessa.

Ongelmien ilmetessä Gispo korjaa mahdolliset QGISin toimintavirheisiin liittyvät ongelmat, mutta tietokannan VPN-yhteyteen tai mahdollisiin autentikointiongelmiin liittyvät virhetilanteet täytyy hoitaa asiakkaan toisen palveluntarjoajan kautta.

Yksikkötestit

Yksikkötestien (Taulukko 3) tavoitteena on varmistaa ohjelmakoodin toimivuus ohjelmaan tehtävistä muutoksista huolimatta. Python-koodin osalta tämä onnistuu pygis-testikirjaston avulla ja näitä testejä on koodissa jonkin verran jo olemassa. Projektissa tulisi tavoitella mahdollisimman laajaa testikattavuutta, mitä voidaan tarkastella testikattavuuden tarkastelua helpottavan pygis-cov-kirjaston avulla.

Taulukko 3. Yksikkötestit

	Testin kohde	Testi	Milloin ja miten tehdään
2a	Ohjelmakoodi	Jokainen luokka ja funktio toimii: automaatiotesti menee läpi vaikka koodi muuttuisi	Ohjelmoinnin aikana, optimaalisesti niin, että jokaiselle kirjoitetulle luokalle ja funktiolle kirjoitetaan samalla myös testi. Työkaluina pygis ja pygis-cov. Kehittäjä tekee.

Automaatiotestien kirjoittaminen on kehittäjien tehtävä. He myös seuraavat testien läpimenoa kehitystyön jatkuessa. Tavoiteltu testikattavuus tulee määritellä.

Järjestelmätestit

Järjestelmätestien tarkoitus on testata järjestelmän toimimista kokonaisuutena. Järjestelmätestejä suunnitellaan toteutettavan kuusi kappaletta ja ne on esitetty Taulukko 4. Kolme ensimmäistä testiä (3a–3c) ovat regressiotestejä, joiden avulla tarkastetaan tietokannassa olevien tietojen oikeellisuus tietokantaan tehtyjen muutosten jälkeen. Testien 3d ja 3e avulla tarkastetaan työtilassa ja Excel-raporteissa näkyvien tietojen oikeellisuus suhteessa tietokannan tietoihin, ja testi 3f testaa työkalua, jonka avulla jätehuoltoviranomainen voi itse viedä uusia tietoja tietokantaan.

Taulukko 4. Järjestelmätestit

	Testin kohde	Testi	Milloin ja miten tehdään
3a	Tietokantamuutokset	Olemassa olevat tiedot ok	Kun tietokannan rakennetta muutettu. Snapshotin avulla vertailu. Kehittäjä tekee.
3b	Tietojen lisäys	Aiemmin syötetyt tiedot ok	Kun tietokantaan lisätty uutta dataa. Snapshotin avulla vertailu. Käyttäjä tekee.
3c	Tietojen päivitys	Aiemmin syötetyt tiedot ok	Kun tietokannan dataa päivitetty. Snapshotin avulla vertailu. Käyttäjä tekee.
3d	QGIS-työtila / raportti	Karttanäkymä toimii raportin määritysten mukaan	Kehityksen aikana karttanäkymän ollessa valmis testattavaksi. Manuaalinen vertailu. Kehittäjä ja käyttäjä.

3e	QGISin xls-export	Taulukkonäkymästä luotu Excel-raportti vastaa määrityksiä	Kehityksen aikana taulukkonäkymästä tuotetun Excel-raportin ollessa valmis testattavaksi. Manuaalinen vertailu. Kehittäjä ja käyttäjä.
3f	Päivitystyökalu	Tiedon päivittäminen tietokantaan onnistuu	Kehityksen aikana päivitystyökalun ollessa valmis testattavaksi. Syötetään päivitysdataa tietokantaan työkalua käyttäen ja tarkistetaan päivitettyjen tietojen näkymisen QGIS-työtilassa. Kehittäjä ja käyttäjä.

Ensimmäiset kolme testiä kohdentuvat tilanteisiin, jossa tietokannan rakennetta muutetaan (3a), tietokantaan lisätään uutta tietoa (3b) tai kun tietokannassa olevaa tietoa päivitetään (3c). Nämä testit voidaan toteuttaa regressiotestein tuottamalla tietokannasta ns. snapshot-tilannekuva ennen muutoksen tekemistä ja tarkastamalla kuvan avulla, että tehtyjen muutosten jälkeenkin aiemmat tiedot ovat tietokannassa kuten pitää.

Testit 3d ja 3e sisältävät manuaalista testausta käyttöliittymässä. Sekä kehittäjä että käyttäjä testaavat käyttöliittymän toimintoja (karttanäkymät ja Excel-exportit) varmistaakseen, että tietokannassa oleva tieto näkyy oikein.

Asiakkaan päivitystyökalu (toistaiseksi suunnittelematta) ja sen toimivuutta testaava testi 3f tulee olemaan järjestelmän käytettävyyden kannalta erittäin keskeinen. Testi tulee varmistamaan, että käyttäjä voi syöttää työkalulla tietoja tietokantaan ja tiedot näkyvät tämän jälkeen oikein käyttöliittymässä. Testin tulee pystyä paljastamaan vietävän datan mahdolliset ongelmat mahdollisimman tarkasti myös silloin, kun ne eivät ole aivan ilmiselviä. Esimerkki tällaisesta vaikeammin havaittavasta aineistovirheestä olisi tilanne, jossa esimerkiksi jätelajin koodi 3 on järjestelmällisesti aineistossa merkitty 4:ksi. Mahdolliset aineistovirheet tulee kartoittaa yhdessä asiakkaan kanssa.

Suorituskykytestit

Suorituskykytestien (Taulukko 5) tavoitteena on varmistaa, että järjestelmän suorituskyky on riittävä sille, että käyttäjä voi tehdä tarvitsemansa asiat ilman mainittavia suorituskykyyn liittyviä haasteita, kuten liian hidas tai katkeava tietokantayhteys. Järjestelmällä on vain muutama käyttäjä, joten kuormitus ei

tule olemaan ongelma. Suorituskyvyn osalta olennaista on nopeus, jolla tietokantaan pystytään viemään ja jolla sieltä saadaan haettua tietoa. Testit 4a ja 4b kohdistuvat tietokantayhteyden toimivuuteen. Suorituskykyongelmia saattaa ilmetä myös QGIS-käyttöliittymässä, minkä vuoksi myös sen suorituskyky tulisi voida varmistaa (4c).

Taulukko 5. Suorituskykytestit

	Testin kohde	Testi	Milloin ja miten tehdään
4a	Tietokanta	Tietokantayhteyden nopeus alle kriittisen rajan tietoja syötettäessä	Kehityksen aikana (kehittäjä) ja järjestelmätestin 3f yhteydessä (käyttäjä) vietäessä tietoa tietokantaan. Manuaalinen ajanotto.
4b	Tietokanta	Tietokantayhteyden nopeus alle kriittisen rajan tietoja luettaessa	Kehityksen aikana (kehittäjä) ja järjestelmätestien 3d ja 3e yhteydessä (käyttäjä) vietäessä tietoa tietokantaan. Manuaalinen ajanotto.
4c	QGIS-käyttöliittymä	Käyttöliittymän nopeus alle kriittisen rajan tietoja luettaessa	Kehityksen aikana, kehittäjä ja käyttäjä. Myös hyväksymistestin 5f aikana. Manuaalinen ajanotto.

Suorituskyvyn testauksen tekevät ensisijaisesti kehittäjät, mutta asiakkaalla on tärkeä rooli sen määrittämisessä, mikä on riittävä nopeus järjestelmää käytettäessä. Palvelimeen liittyvien mahdollisten ongelmien ratkaisemisessa tarvitaan yhteistyötä asiakkaan palvelintoimittajan kanssa, kun taas QGISin suorituskyvyn parantamiseen liittyviä toimenpiteitä voidaan tarvittaessa tehdä Gispon ja tarvittaessa avoimen lähdekoodin paikkatieto-osaajien muodostaman FOSS4G-yhteisön toimesta. Suorituskykytestien toteutukseen ei tässä järjestelmässä todennäköisesti tarvita erillisiä työkaluja.

Hyväksymistestit

Kun järjestelmä on valmistumassa käyttöönottovaiheeseen, se tulee testata hyväksymistestein (Taulukko 6). Näin voidaan varmistua siitä, että järjestelmä vastaa sille asetettuja vaatimuksia ja toimii odotetusti. Käyttäjän tulee voida käyttää järjestelmää itsenäisesti sillä tasolla, joka on määritetty. Näin ollen tarkastelussa ovat järjestelmän dokumentaatio (5a), järjestelmän varsinaiset toiminnot (5b-5e) sekä myös järjestelmän käytettävyys (5f).

Taulukko 6. Hyväksymistestit

	Testin kohde	Testi	Milloin ja miten tehdään
5a	Dokumentaatio	Järjestelmän käyttöönotto ja käyttäminen onnistuu	Käyttöönoton yhteydessä, (mielellään uusi) käyttäjä
5b	Järjestelmä	Karttanäkymät toimivat määritysten mukaan (sisältö, nopeus)	Käyttöönoton yhteydessä. Toimittaja ja asiakas määrittävät menetelmän ja läpimenokriteerit yhdessä, käyttäjä suorittaa.
5c	Järjestelmä	Excel-raportit vastaavat määrityksiä	Käyttöönoton yhteydessä. Toimittaja ja asiakas määrittävät menetelmän ja läpimenokriteerit yhdessä, käyttäjä suorittaa.
5d	Järjestelmä	Tietojen päivittäminen tietokantaan onnistuu	Käyttöönoton yhteydessä. Toimittaja ja asiakas määrittävät menetelmän ja läpimenokriteerit yhdessä, käyttäjä suorittaa.
5e	Järjestelmä	Päivitetyt tiedot näkyvät oikein käyttöliittymässä	Käyttöönoton yhteydessä. Toimittaja ja asiakas määrittävät menetelmän ja läpimenokriteerit yhdessä, käyttäjä suorittaa.
5f	Käytettävyys	Käyttöliittymä täyttää sille asetetut käytettävyysvaatimukset	Käyttöönoton yhteydessä, käyttäjät (mielellään mukana myös uusi käyttäjä), testitapa määritettävä yhdessä asiakkaan kanssa, mutta voi sisältää esim. ohjeidenmukaista käyttöä ja sen sujuvuuden raportointia.

Erityisesti dokumentaation (5a) ja käytettävyyden (5f) näkökulmasta testien eduksi olisi, jos käyttäjätestaaja olisi henkilö, joka ei ole ollut asiakkaan tai käyttäjän edustajana mukana järjestelmän kehitystyössä. Näin voitaisiin parhaiten varmistua siitä, että ohjeet ja käyttöliittymä toimivat myös ilman hiljaista tietoa, jota projektissa mukana oleville kertyy kehitysvaiheen aikana.

Käytettävyystestauksen osalta määritykset riittävän hyvästä käytettävyydestä tulee laatia joko asiakkaan toimesta tai Gispon ja asiakkaan yhteistyönä. Näiden pohjalta voidaan valita sopivat menetelmät käytettävyystestin suorittamiseen. Mahdollinen menetelmä voisi olla esimerkiksi sarja tehtäviä ja niiden sujuvuuden arviointi annetuilla kriteereillä.

Hyväksymisvaiheeseen päästäessä järjestelmän ydintoimintojen testauksen (5b–5e) tulisi olla melko suoraviivaista, koska järjestelmän osia on testattu kehityksen aikana järjestelmätestein. Gispo ja asiakas päättävät yhdessä, miten testit määritetään ja suoritetaan.

Hyväksymistestivaiheessa olennaista on käydä testien hyväksymiskriteerit, suoritustapa sekä tulokset huolellisesti läpi ennen projektin päättämistä. Näin voidaan todentaa järjestelmän toimivuus kokonaisuutena, minkä lisäksi hyvin laadittu testidokumentaatio toimii myös käytön ja mahdollisen jatkokehityksen tukena.

6 YHTEENVETO

Laadunvarmistus on ohjelmistoalalla yrityksen kilpailuvaltti, ja testaus on laadunvarmistuksen keskeinen työkalu. Tässä työssä laadittu testauksen viitekehys tarjoaa Gispolle yleisen työkalun ohjelmistoprojektien laadunvarmistuksen tueksi ja sen pohjalta tehty projektikohtainen testaussuunnitelma toimii paitsi työkaluna kyseiselle projektille, myös esimerkkinä testaustyön suunnittelusta yksittäisessä ohjelmistoprojektissa.

Hyvin suunniteltu testaus parantaa ohjelmistokehitystyön tehokkuutta ja laatua, mutta testausta varten tarjolla olevien tapojen ja työkalujen kirjo voi vaikuttaa liian monimutkaiselta etenkin silloin, kun tekeillä olevat projektit ovat kooltaan suhteellisen pieniä. Viitekehyksen avulla Gispo voi jatkossa hyödyntää jo sen tarpeisiin yleisellä tasolla valikoituja testauksen vaihtoehtoja ja valita niistä kulloisenkin projektin puitteisiin parhaiten sopivat menetelmät. Joskus pieneen projektiin riittää vähäinenkin määrä testausta, mutta niin pientä ohjelmistoprojektia ei olekaan, etteikö jonkinlainen testaus olisi tarpeen. Lisäksi jokainen projekti hyötyy siitä, että testaus on huomioitu nimenomaan kokonaisuutena. Vain tällä tavalla voidaan varmistua siitä, että myös ohjelmisto toimii kokonaisuutena eikä vain joiltakin osin.

Gispolla laadukas ohjelmisto nähdään asiakkaalle hyödyllisenä, tehokkaana, luotettavana, turvallisena ja helppokäyttöisenä, minkä lisäksi sujuva ylläpidet-

tävyys ja toteutettujen ratkaisujen siirrettävyys tuovat myös yritykselle liiketoimintaetuja. Koska Gispo pyrkii julkaisemaan luomansa paikkatietoratkaisut avoimesti ja tarjoamaan niitä vapaaseen käyttöön, laatu on keskeinen tekijä myös siinä, päätyykö joku hyödyntämään luotuja ratkaisuja joko suoraan tai oman jatkokehityksensä pohjana.

Lahden seudun jätehuoltoviranomaisen projektia varten laadittu testaussuunnitelma näyttää, miten viitekehyksen pohjalta voidaan laatia yksittäiselle projektille alustava testauskokonaisuus. Projektin tässä vaiheessa testit on suunniteltu melko yleisellä tasolla, mutta suunnitelma toimii raamina Gispon ja jätehuoltoviranomaisen yhdessä tekemän testaustyön suunnittelulle ja toteutukselle projektin edetessä. Suunnitelma voidaan nähdä myös eräänlaisena muistilistana, jonka avulla projektissa voidaan pysähtyä tarkastelemaan laadunvarmistukseen liittyviä asioita sitä mukaa kuin ne tulevat ajankohtaiseksi.

Itselleni opinnäytetyön tekeminen aiheesta, joka kiinnosti mutta josta minulla oli työhön tarttuessani melko vähän tietämystä, oli melko haastava tehtävä. Monimutkaiseen aiheeseen perehtyminen oli työlästä, koska kirjallisuudessa ja verkkolähteissä on paljon toisistaan poikkeavaa tietoa, ja erilaisia lähestymistapoja samoihin asioihin löytyi aina lukuisia. Se, millainen ohjelmisto testauksen kohteena on, vaikuttaa valtavasti testauskokonaisuuden muodostumiseen. Esimerkiksi tehdasjärjestelmät eroavat huomattavasti yksinkertaisista verkkosivustoista, mutta molempia tulee testata. Osittain menetelmät voivat olla samankaltaisia ja osittain aivan toisistaan poikkeavia. Yleiskuvan muodostaminen aiheesta ei näin ollen ollut helppoa. Mielestäni onnistuin kuitenkin lopulta saattamaan luomani yleiskuvan sopivalle tasolle nimenomaan Gispon näkökulmasta. Uskon myös työni tulosten olevan hyödyllisiä, mikä lisäsi motivaatiota työn tekemiseen.

Gispon ohjelmistoprojektien koko on kasvanut viime vuosina, mikä on luonnollisesti lisännyt tarvetta kehittää ohjelmistokehityksen toimintamalleja. Opinnäytetyön tulokset tarjoavat pohjan systemaattisempien testauskäytäntöjen hyödyntämiselle yrityksen ohjelmistoprojekteissa. Gispon tehtäväksi jää harkita, millä tavalla työ otetaan yrityksessä käyttöön ja esimerkiksi kokeilla työssä lis-

tattujen testaustyökalujen soveltuvuus Gispon todellisissa ohjelmistoprojekteissa tai vaihtoehtoisesti esimerkiksi jonkin toisen opinnäytetyön kehitystehtävänä. Jos Gispolla tehdään tulevaisuudessa nykyistä suurempia projekteja, tarvetta saattaa tulla myös henkilöstön koulutukselle monipuolisempien testityökalujen käyttöön. Tällä hetkellä riittävä osaaminen on nähdäkseni olemassa ja tärkeintä olisi sisällyttää testaus kattavasti ja kokonaisvaltaisesti sekä suunnitteilla että tekeillä oleviin projekteihin. Toivon tämän helpottuvan työni tulosten avulla.

LÄHTEET

Adam, J. 2021. What is the V-model approach to software development and testing? WWW-dokumentti. Saatavissa: https://kruschecompany.com/v-model-software-development-methodology/ [viitattu 8.11.2022].

Adam, J. 2022. What is Agile software development? WWW-dokumentti. Saatavissa: https://kruschecompany.com/agile-software-development/ [viitattu 8.11.2022].

Ambler, S. 2022. Why Agile Software Development Techniques Work: Improved Feedback. WWW-dokumentti. Saatavissa: https://www.ambysoft.com/essays/whyAgileWorksFeedback.html [viitattu 16.12.2022].

Apache. s.a. Apache JMeter™. WWW-dokumentti. Saatavissa: https://jmeter.apache.org/ [viitattu 29.12.2022].

Appium. s.a. appium – Automation for Apps. WWW-dokumentti. Saatavissa: https://appium.io/ [viitattu 29.12.2022].

Bernier, R. 2022. Working With Snapshotsin PostgreSQL. Blogi. Saatavissa: https://www.percona.com/blog/working-with-snapshots-in-postgresql/ [viitattu 28.12.2022].

Bertolino, A. 2001. Software Testing. Teoksessa Abran, A., Moore J., Bourque, P. & Dupuis R. (toim.) Guide to the Software Engineering Body of Knowledge. Los Alamitos: IEEE Computer Society, 88–105. PDF-dokumentti. Saatavissa: https://cmapspublic.ihmc.us/rid=1K1K7VJ2J-1QMZQ0M-2SXJ/SWEBOK.pdf#page=88 [viitattu 18.12.2022].

Contrast Security. 2014. The Unfortunate Reality of Insecure Libraries. PDF-dokumentti. Saatavissa: https://cdn2.hubspot.net/hub/203759/file-1100864196-pdf/docs/Contrast_-_Insecure_Libraries_2014.pdf [viitattu 29.12.2022].

Cruzes, D., Felderer, M. Oyetoyan T., Gander, M. & Pekaric I. 2017. How is Security Testing Done in Agile Teams? A Cross-Case Analysis of Four Software Teams. WWW-dokumentti. Saatavissa: https://www.researchgate.net/publication/316560451_How_is_Security_Testing_Done_in_Agile_Teams_A_Cross-Case_Analysis_of_Four_Software_Teams [viitattu 29.12.2022].

Dodds, K. 2021. The Testing Trophy and Testing Classifications. Blogi. Päivitetty 3.6.2021. Saatavissa: https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications [viitattu 28.12.2022].

Dominguez, J. 2009. The Curious Case of the CHAOS Report 2009. Blogi. Päivitetty 10.10.2021. Saatavissa: https://www.projectsmart.co.uk/it-project-management/the-curious-case-of-the-chaos-report-2009.php [viitattu 17.12.2022].

Fowler, M. 2021. On the Diverse And Fantastical Shapes of Testing. Blogi. Päivitetty 2.6.2021. Saatavissa: https://martinfowler.com/articles/2021-test-shapes.html [viitattu 28.12.2022].

GitHub Docs. s.a. Understanding GitHub Actions. WWW-dokumentti. Saatavissa: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions [viitattu 30.12.2022].

Hamilton, T. 2022. TEST PLAN: What is, How to Create (with Example). Blogi. Päivitetty 16.12.2022. Saatavissa: https://www.guru99.com/what-everybody-ought-to-know-about-test-planing.html [viitattu 27.12.2022].

Holcombe, M. 2008. Running an Agile Software Development Project. Hoboken: Wiley. E-kirja. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 17.12.2022].

ISO/IEC 25010:2011(en). 2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation — System and software quality models. WWW-dokumentti. Saatavissa: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en [viitattu 4.12.2022].

ISO/IEC/IEEE 29119-1:2022(en). 2022. Software and systems engineering — Software testing — Part 1: General concepts. WWW-dokumentti. Saatavissa: https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:29119:-1:ed-2:v1:en [viitattu 17.12.2022].

Jest. 2022. Jest. WWW-dokumentti. Saatavissa: https://jestjs.io/ [viitattu 29.12.2022].

Jorgensen, P. 2014. Software Testing. A Craftman’s Approach. Boca Raton: Taylor & Francis Group.

Jätelaki 15.7.2021/714.

Kasurinen, J P. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo. E-kirja. Saatavissa: https://www.ellibslibrary.com/book/978-952-5912-99-9 [viitattu 7.11.2022].

Kennett, R., Ruggeri, F. & Faltin, F. (toim.) 2018. Analytic methods in systems and software testing. Hoboken: Wiley. E-kirja. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 7.11.2022].

Kumar, D. & Mishra, K. 2016. The Impacts of Test Automation on Software’s Cost, Quality and Time to Market. *Procedia Computer Science* 79, 8–15. Saatavissa: https://www.researchgate.net/publication/300080121_The_Impacts_of_Test_Automation_on_Software's_Cost_Quality_and_Time_to_Market [viitattu 27.12.2022].

Locust. 2022. Locust – An open source load testing tool. WWW-dokumentti. Saatavissa: https://locust.io/ [viitattu 29.12.2022].

Luukkainen, M. 2022. Ohjelmistotuotanto 2022. Helsingin yliopisto. WWW-dokumentti. Saatavissa: https://ohjelmistotuotanto-hy.github.io/osa3/ [viitattu 16.12.2022].

Molina, A. 2021. Crafting Test-Driven Software with Python. E-kirja. Saatavissa: https://subscription.packtpub.com/book/web-development/9781838642655/2/ch02lvl1sec08/understanding-the-testing-pyramid-and-trophy [viitattu 27.12.2022].

Mullans, A. 2020. Keep all your packages up to date with Dependabot. Blogi. Saatavissa: https://github.blog/2020-06-01-keep-all-your-packages-up-to-date-with-dependabot/ [viitattu 29.12.2022].

Myers, G., Sandler, C. & Badgett T. 2012. The Art of Software Testing. Hoboken: Wiley. E-kirja. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 27.12.2022].

Mypy. 2014. WWW-dokumentti. Saatavissa: https://mypy-lang.org/ [viitattu 29.12.2022].

Nielsen, J. 2012. Usability 101: Introduction to Usability. WWW-artikkeli. Julkaistu 3.1.2012. Saatavissa https://www.nngroup.com/articles/usability-101-introduction-to-usability/ [viitattu 26.12.2022].

Niemelä, H. 2020. Sovelluksen käytettävyyden testaaminen. WWW-artikkeli. Julkaistu 2.6.2020. Saatavissa https://lehti.seamk.fi/alykkaat-ja-energiatehokkaat-jarjestelmat/sovelluksen-kaytettavyyden-testaaminen/ [viitattu 26.8.2022].

OWASP. s.a. OWASP Dependency-Check. WWW-dokumentti. Saatavissa: https://owasp.org/www-project-dependency-check/ [viitattu 30.12.2022].

Palamarchuk, S. 2015. Best Testing Practices for Agile Teams: The Automation Pyramid. Blogi. Päivitetty 26.10.2015. Saatavissa: https://abstracta.us/blog/test-automation/best-testing-practices-agile-teams-automation-pyramid/ [viitattu 29.12.2022].

Passby, K. 2021. How to create a test plan for software testing. Blogi. Päivitetty 22.9.2021. Saatavissa: https://www.wearedevelopers.com/magazine/how-to-create-a-test-plan-for-software-testing [viitattu 9.11.2022].

Playwright. 2022. Playwright enables reliable end-to-end testing for modern web apps. WWW-dokumentti. Saatavissa: https://playwright.dev/ [viitattu 30.12.2022]

Pries, K. & Quigley, J. 2010. Scrum Project Management. Boca Raton: Taylor & Francis Group. E-kirja. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 8.11.2022].

Puppeteer. 2022. Puppeteer. WWW-dokumentti. Saatavissa: https://pptr.dev/ [viitattu 30.12.2022].

PyAutoGUI. s.a. Welcome to PyAutoGUI's documentation! WWW-dokumentti. Saatavissa: https://pyautogui.readthedocs.io/en/latest/ [viitattu 29.12.2022].

Pytest. 2015. pytest: helps you write better programs. WWW-dokumentti. Saatavissa: https://docs.pytest.org/en/7.2.x/ [viitattu 29.12.2022].

Python Software Foundation. s.a. a. unittest – Unit testing framework. WWW-dokumentti. Päivitetty 30.12.2022. Saatavissa: https://docs.python.org/3/library/unittest.html [viitattu 29.12.2022].

Python Software Foundation. s.a. b. typing – Support for type hints. WWW-dokumentti. Saatavissa: https://docs.python.org/3/library/typing.html [viitattu 29.12.2022].

Robot Framework. s.a. Robot Framework. WWW-dokumentti. Saatavissa: https://robotframework.org/ [viitattu 29.12.2022].

Royce, W.W. 1970. Managing the Development of Large Software Systems. *Proceedings of IEEE WESCON* 26, 328–388. Saatavissa: http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf [viitattu 9.11.2022].

Scrum-opas. 2020. PDF-dokumentti. Saatavissa: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Finnish.pdf [viitattu 17.12.2022].

Selenium. 2022. Selenium automates browsers. That's it! WWW-dokumentti. Saatavissa: https://www.selenium.dev/ [viitattu 29.12.2022].

Slaughter, S., Harter D. & Krishnan M. 1998. Evaluating the Cost of Software Quality. *Communications of the ACM* 8, 67–73. Saatavissa: https://dl.acm.org/doi/pdf/10.1145/280324.280335 [viitattu 16.12.2022].

Steinfeld, R. 2022. 5 steps of test-driven development. Blog. Päivitetty 6.2.2022. Saatavissa: https://developer.ibm.com/articles/5-steps-of-test-driven-development/ [viitattu 28.12.2022].

The Standish Group. 2015. CHAOS Report 2015. WWW-dokumentti. Saatavissa: https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf [viitattu 4.12.2022].

Vala. 2022. Mitä on ohjelmistotestaus ja mitä hyötyä siitä on? Blogi. Päivitetty 10.11.2022. Saatavilla: https://www.valagroup.com/fi/blogi/mita-on-ohjelmistotestaus-ja-mita-hyotya-siita-on/ [viitattu 27.12.2022].

Vala. s.a. Software Quality Guidelines. How to focus on what matters the most? Saatavissa: https://www.valagroup.com/wp-content/uploads/2021/12/software-qualit_53686470-compressed.pdf [viitattu 28.12.2022].

Wallace, D. & Reeker, L. 2001. Software Quality. Teoksessa Abran, A., Moore J., Bourque, P. & Dupuis R. (toim.) Guide to the Software Engineering Body of Knowledge. Los Alamitos: IEEE Computer Society, 182–199. PDF-dokumentti. Saatavissa: https://cmapspublic.ihmc.us/rid=1K1K7VJ2J-1QMZQ0M-2SXJ/SWEBOK.pdf [viitattu 18.12.2022].

Wang, Y., Mäntylä, M., Liu, Z., Markkula, J. & Raulamo-Jurvanen, P. 2022. Improving test automation maturity: A multivocal literature review. *Software testing, verification & reliability* 32, 2–40. Saatavissa: https://arxiv.org/pdf/2202.09076.pdf [viitattu 9.11.2022].

Wiegers, K. 2021. The Cost of Quality in Software. Blogi. Päivitetty 20.9.2021. Saatavissa: https://medium.com/geekculture/the-cost-of-quality-in-software-5f113a26e759 [viitattu 16.12.2022].

Woke, G. 2022. Testing tools: a Classification. Blogi. Päivitetty 24.8.2022. Saatavissa: https://blog.openreplay.com/testing-tools-a-classification [viitattu 29.12.2022].