**Miika Immonen**

# Java-sovelluksen kestävyystestausprosessin automatisointi

Tietotekniikan Pro gradu -tutkielma

2. elokuuta 2023

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

**Tekijä:** Miika Immonen

**Yhteystiedot:** immomipe@student.jyu.fi

**Ohjaaja:** Tommi Mikkonen

**Työn nimi:** Java-sovelluksen kestävyystestausprosessin automatisointi

**Title in English:** Automating endurance testing process of a Java application

**Työ:** Pro gradu -tutkielma

**Opintosuunta:** Ohjelmisto- ja tietoliikennetekniikka

**Sivumäärä:** 55+0

**Tiivistelmä:** Ohjelmistotestauksella etsitään testattavissa ohjelmistoissa mahdollisesti olevia virheitä ja tarkistetaan ohjelmistoilta vaadittavien toimintojen toimivuutta. Ohjelmistojen kestävyystestauksella etsitään virheitä erityisesti resurssienkäytössä, kun ohjelmistoa suoritetaan pitkiä yhtäjaksoisia aikoja. Tutkielman kohteena olevalle Java-sovellukselle kestävyystestausta suoritettiin lähtötilanteessa manuaalisesti. Nämä kestotesteiksi kutsutut jaksot kestivät useita viikkoja ja ne suoritettiin yhdelle ohjelmistoversiolle kerrallaan. Kestotestien ongelmina olivat niiden vaatima aika ja suuri työvoiman tarve. Testijaksojen vaatimaa työmäärää pyrittiin vähentämään testiautomaation avulla mahdollistaen ohjelmiston resurssinkäytön seuraamisen ja mahdollisten virheiden löytämisen. Resursseista seurattiin sovelluksen aiheuttamaa prosessorikuormaa ja muutoksia sovelluksen muistinkäytössä. Erityisesti pyrittiin tarkkailemaan mahdollisten muistivuotojen ilmenemistä. Tutkielmassa tutkittiin sovelluksen kestävyystestausta automaattisen testauksen keinoin. Tutkimus suoritettiin suunnittelututkimuksena, jossa muodostettiin ohjelmiston kestotestausta automatisoiva artefakti. Ratkaisuna ongelmiin kehitetyn testiautomaation avulla kestotestijakson vaatima manuaalisen työn määrä väheni. Testiautomaation avulla suoritetuilla testijaksoilla kyettiin myös osoittamaan ohjelmistosta muistivuotoja. Muodostettu artefakti mahdollistaa ohjelmiston resurssienkäytön seurannan. Muodostetun testiautomaation avulla kestotestejä voidaan suorittaa myös rinnakkain kahdelle ohjelmistoversiolle ilman merkittävää lisätyötä.

**Avainsanat:** ohjelmistotestaus, automaattinen testaus, kestävyystestaus, suorituskykytestaus,

Jenkins, Robot Framework, Java

**Abstract:** Software testing is used in determining possible errors in the software under test and checking the functionality of the functions required from the software. Software endurance testing looks for errors, especially in the resource usage, when the software application is kept running for extended periods of time. For the Java application under study, endurance testing was performed manually at baseline. These endurance test cycles lasted several weeks and were each performed on one version of the software at a time. The problems with the endurance tests were the substantial time and manpower requirements. Test automation was used to reduce the workload required in testing the application, to monitor resource usage of the application and to detect possible errors. The monitored resources consisted of CPU load imposed by the application and changes in the memory usage of the software under test. In particular, the aim was to monitor the occurrence of possible memory leaks. In this thesis, the endurance testing of the application was investigated by means of automated testing. The study was conducted using design science research method where an artefact was created to automate the software endurance testing. As a solution to the presented problems test automation was developed to reduce the amount of manual work required during the endurance test cycle. The test automation test cycles that were run also detected memory leaks in the software. The created artefact allows the monitoring of the software's resource usage. The artefact also allows the parallel endurance testing of two different software versions without the need of substantial additional labour.

**Keywords:** software testing, automated testing, endurance testing, performance testing, Jenkins, Robot Framework, Java

# Kuviot



# Taulukot

# Sisällys

# 1 Johdanto

Ohjelmistotestauksen ammattilaisten parissa yhtenä alan suurimpana haasteena nähdään testausaktiviteettien automatisointiin liittyvät seikat (Garousi ja Mäntylä 2016). Kun ohjelmistoalan akateemisen maailman kiinnostus testiautomaatiossa on kohdistunut esimerkiksi automatisointiin testitapausten luomisen yhteydessä, ohjelmistoteollisuudessa puheenaihe on keskittynyt erityisesti testitapausten suorittamiseen automaation avulla (Garousi ja Felderer 2017).

Testiautomaatiolla pyritään täydentämään, ei korvaamaan, manuaalisesti tehtyä testaustyötä (Kasurinen 2015). Manuaalinen testaus ei välttämättä vaadi osaamista testaustyökalujen käytöstä, mutta testitapausten suorittaminen vaatii automaattiseen testaukseen verrattuna enemmän panostusta ja resursseja kuten työntekijöitä (Umar ja Zhanfang 2019). Testiautomaatiota hyödyntämällä on ohjelmistoprojekteissa siis mahdollista uudelleenohjata testauksen vaatimia resursseja.

Kestävyystestauksen avulla voidaan selvittää ohjelmiston suoriutumista käytettäessä sitä erilaisilla kuormilla (ISO/IEC 29119-1 2022a). Täten pyrkimys on selvittää esimerkiksi mahdollisia ongelmakohtia sen resurssien käytössä tai vasteajoissa, kun ohjelmistoa käytetään erityisen pitkäkestoisina jaksoina (ISO/IEC 29119-1 2022b). Kun kestävyystestausta suoritetaan viikkokausia kestäviä jaksoja, vaatii testauksen suorittaminen automaation hyödyntämistä (Kaner, Bach ja Pettichord 2002).

Tutkielmassa selvitettiin, millä tavoin uudistetaan tutkittavan kehityksen alaisen Java-ohjelmiston testausta siten, että automaattisten testien avulla voidaan onnistuneesti korvata lähtötilanteessa manuaalisesti suoritettavaa kestävyystestausta. Kestävyystestauksen avulla imitoidaan loppuasiakkaan todellista käyttöä, jossa ohjelmistoa tullaan käyttämään vuorokauden ympäri pitkiä aikoja. Jatkuvassa käytössä olevalle ohjelmistolle on asetettu useita laatuvaatimuksia, joista yhtenä on sen resurssien käytölle asetetut vaatimukset esimerkiksi muistin osalta.

Kestävyystestausta on kohteena olevalle järjestelmälle suoritettu osana kestotestijaksoja. Kestotesti on yrityksen käyttämä nimitys pitkäkestoisesta yhtämittaisesta testausjaksosta, jonka

aikana ohjelmistoa ei sammuteta. Tutkielmassa lähestyttiin testausprosessin kehittämistä tilanteesta, jossa järjestelmälle manuaalisesti suoritetut kestotestijaksot vaativat runsaasti manuaalista työpanosta. Tutkielmassa ongelmaa ratkottiin suunnittelututkimuksen periaatteiden mukaisesti ja toteutettiin artefaktina kestotestijakson testiautomaatio.

Luvussa 2 käydään läpi ohjelmistotestauksen perusperiaatteita sekä keskeisimpiä käsitteitä. Lisäksi selvennetään, mitä ohjelmistotestauksella tavoitellaan sekä käydään läpi automaattisen ja manuaalisen testauksen pääpiirteitä ja eroavaisuuksia toisiinsa nähden.

Luvussa 3 käsitellään kestävyystestauksen piirteitä ja merkitystä. Käydään läpi millaisia seikkoja kestävyystestauksen yhteydessä ohjelmistosta pyritään tarkkailemaan, sekä mitä työkaluja kestävyystestauksessa voidaan käyttää apuna.

Luvussa 4 käydään läpi tutkimuksen tavoitteet ja suoritustapa. Lisäksi esitellään käytetty tutkimusmenetelmä, tutkimuksen kohteena oleva järjestelmä ja syyt, miksi se valikoitui tutkimuksen kohteeksi. Luvussa käydään läpi esimerkiksi Java-ohjelmointikielen piirteitä, joita kestävyystestauksessa on hyvä ottaa huomioon.

Luvussa 5 esitellään tutkimuksessa muodostetun artefaktin kehitysprosessi. Lähtötilanteen ongelmien kartoituksen jälkeen kuvataan artefaktin muodostamiseksi käyty prosessi.

Luvussa 6 arvioidaan, millä tavoin artefaktin muodostamisessa ja tutkimuksen suorituksessa onnistuttiin.

Luvussa 7 käydään läpi koonti tutkimuksen tuloksista ja esitellään mahdolliset jatkotutkimuskohteet.

# 2 Ohjelmistotestaus

Laadun täytyy olla osana ohjelmistotuotannon prosessia, eikä laatua voida lisätä tuotteeseen pelkällä testauksella (Ammann ja Offutt 2016). Kaikessa testaamisessa lähtökohtana on se, mitä ohjelmistolta odotetaan. Ohjelmistolle määritettävät vaatimukset kuvaavat ohjelmiston toimintaa ja näihin vaatimuksiin tulee myös linkittyä se, millaisia testejä ohjelmistolle tehdään (Ammann ja Offutt 2016). Näiden testien avulla pyritään varmistumaan siitä, että ohjelmisto suoriutuu riittävällä tavalla siltä vaadituista toimenpiteistä.

Ohjelmistotestaukselle voidaan määritellä testaustasoja, jotka sidotaan ohjelmistokehityksen eri vaiheisiin (Kasurinen 2015). Yksikkötestauksessa testataan tarkoin rajattua yksikköä ohjelmistosta, joka voi olla esimerkiksi ohjelmiston yksittäinen moduuli. Integraatiotestauksessa taas tarkastellaan useiden eri moduulien yhteistoimintaa. Järjestelmätestauksessa testataan nimensä mukaisesti kokonaista ohjelmistoa. Järjestelmätestausvaihe suoritetaan vielä tarkoin hallitussa ohjelmistokehittäjän sille määrittämässä testausympäristössä. Tulee huomioida, että järjestelmätestaus ei itsessään määritä käytettävää testaustyyppiä, vaan niitä voi olla käytössä useita. Viimeisenä lenkkinä ohjelmistotestauksessa voidaan pitää hyväksymistestausta. Tällöin asiakas pääsee käyttämään ohjelmistoa osana omaa järjestelmäänsä ja täten pääsee testaustason nimen mukaisesti hyväksymään valmiin ohjelmiston otettavaksi käyttöön, jos sen nähdään täyttävän ohjelmistolle asetetut vaatimukset. Tämän tutkielman käsittelemä kestävyystestaus on yksi järjestelmätestauksen yhteyteen sijoitettava testaustyyppi.

Alaluvussa 2.1 käsitellään pääpiirteittäin, mitä ohjelmistotestauksella tarkoitetaan. Alaluvussa 2.2 käydään lyhyesti läpi ohjelmistotestauksen tavoitteita. Alaluvussa 2.3 kuvaillaan, mitä on manuaalinen ohjelmistotestaus ja käsitellään eroavaisuuksia automaattiseen testaukseen. Alaluvussa 2.4 esitellään automaattisen testauksen pääpiirteitä ja sen etuja manuaaliseen testaukseen nähden. Lisäksi tarkastellaan, missä tilanteissa automaattista testausta olisi syytä hyödyntää. Alaluvussa 2.5 esitellään yksi automaattisessa testauksessa hyödynnettävä työkalu, Robot Framework.

## 2.1 Mitä ohjelmistotestaus on?

Ohjelmistotestauksessa pyritään selvittämään, onko muodostettava ohjelmisto tarkoituksenmukainen ja suoriutuuko se siltä vaadituista toiminnoista oikein (Kasurinen 2015). Testauksen kohde voi olla kokonainen järjestelmä tai esimerkiksi vain osa siitä (ISO/IEC 29119-1 2022a). Käytännössä ohjelmistotestauksessa pyritään muodostamaan sellaisia testitapauksia, joilla todennäköisesti saadaan selville ohjelmistossa piileviä ongelmia. Näille testitapauksille taas tulee voida määrittää, milloin testistä on suoriuduttu oikein.

Testioraakkelin avulla voidaan määrittää, onko testitapaus suoritettu hyväksytysti vai ei (ISO/IEC 29119-1 2022a). Käytännössä oraakkeli voi olla esimerkiksi ohjelmistolle asetettu tarkka vaatimus. Täydellisesti muodostettu testioraakkeli voisi määrittää aina, onko testitapauksen lopputulos hyväksytty vai ei (Kasurinen 2015). Tällainen mahdollistaisi muun muassa kyseisen kohteen testaamisen automatisoinnin (ISO/IEC 29119-1 2022a). Täydellisen testioraakkelin määrittäminen on usein hankalaa tai mahdotonta.

Testauksen suorittaja voi olla ohjelmistokehittäjä tai erillinen testaaja. Useimmiten yksikkötestauksen suorittavat itse ohjelmoijat (Kasurinen 2015). Yleisellä tasolla kehittämisen ja testaamisen suorittamisen eriyttäminen eri henkilöille on kuitenkin suotavaa, sillä tällä tavoin ohjelmistossa mahdollisesti piileviä virheitä löydetään todennäköisimmin (ISO/IEC 29119-1 2022a).

Ohjelmistoa ja sen toimintaa voidaan arvioida monin eri keinoin, kuten esimerkiksi ohjelmistoon kohdistetun staattisen ja dynaamisen testauksen avulla (Naik ja Tripathy 2013). Staattinen testaus kattaa toimenpiteitä, joissa itse ohjelmistoa ei suoriteta. Tällaista toimintaa voidaan suorittaa ohjelmistokehityksen missä tahansa vaiheessa. Kun ohjelmistoa tällä tavoin arvioidaan, voidaan puhua staattisesta testauksesta, johon liittyy tutustuminen ohjelmistoon liittyvään dokumentaatioon esimerkiksi lähdekoodin tarkastelua tekemällä (ISO/IEC 29119-1 2022a). Staattisessa testauksessa voidaan käyttää apuna erillisiä ohjelmistoja, kuten koodianalysaattoreita, joiden avulla voidaan löytää esimerkiksi syntaktisia virheitä ohjelmakoodista (Kasurinen 2015). Staattisen testauksen etu piilee kustannuksissa, sillä kehitystyön varhaisessa vaiheessa löydettävät ongelmat ovat usein edullisimpia korjata.

Dynaamisessa testauksessa ohjelmistoa suoritetaan (Naik ja Tripathy 2013). Dynaamista tes-

tausta voidaan siis suorittaa vasta siinä ohjelmistokehityksen vaiheessa, kun ohjelmakoodin valmiusaste mahdollistaa ohjelmiston suorittamisen (ISO/IEC 29119-1 2022a). Dynaamista testausta voidaan täten pitää staattisen testauksen vastakohtana, sillä ohjelmiston osasten toimintaa tarkastellaan käytännön suorituksen aikana (Kasurinen 2015). Dynaaminen testaus täten voi sisältää ohjelmiston toimintojen vertaamista niille asetettuihin vaatimuksiin sekä esimerkiksi suorituskyvyn tarkastelua. Tässä tutkielmassa huomio keskittyy dynaamiseen testaukseen.

## 2.2 Ohjelmistotestauksen tavoitteet

Ohjelmistotestauksen avulla voidaan haluta varmistaa ohjelmiston osioiden toiminnan virheettömyyttä tai vaihtoehtoisesti osoittaa mahdollisia virheellisiä elementtejä (Naik ja Tripathy 2013). Ohjelmiston toiminnasta pyritään saamaan riittävästi tietoa, jotta se voitaisiin luottavaisin mielin ottaa käyttöön (ISO/IEC 29119-1 2022a). Mahdollisten virheiden olemassaolon lisäksi testauksen avulla selvitetään, puuttuuko ohjelmistosta sille asetetuissa vaatimuksissa esitettyjä seikkoja (Jamil ym. 2016). Ohjelmistotestauksella ei kuitenkaan voida osoittaa ohjelmiston täydellistä virheettömyyttä (Ammann ja Offutt 2016). Onkin mielekkäämpää tarkastella ohjelmiston toiminnan oikeellisuutta suhteessa siltä odotettuihin ominaisuuksiin. Näitä ominaisuuksia voivat olla esimerkiksi ohjelmiston toiminnan luotettavuus, turvallisuus tai tehokkuus. Ohjelmistotestaus toimii työkaluna ohjelmiston laadun verifioinnissa ohjelmiston kehityskaaren aikana (Naik ja Tripathy 2013).

Testaamalla voidaan hankkia myös tietoa ohjelmiston toiminnasta, jotta esimerkiksi ohjelmistokehittäjät voisivat kehittyä työssään (ISO/IEC 29119-1 2022a). Testauksesta saatavaa informaatiota käytetään myös sen päättämiseksi, onko ohjelmistoa jo testattu tarpeeksi.

## 2.3 Manuaalinen testaus

Testaamista voidaan suorittaa joko käsin testaavan henkilön toimesta tai automaation avulla(ISO/IEC 29119-1 2022a). Usein testaamiseen liittyvä testitapausten muodostaminen ja testauksen suorittaminen vaatii manuaalista työtä (Naik ja Tripathy 2013). Yleisellä tasolla manuaalinen testaus voidaan esimerkiksi jakaa ennalta käsikirjoitettuun testaamiseen ja niin

sanottuun ad hoc -testaukseen, joka on täysin suunnittelematonta (Kasurinen 2015).

Kun ohjelmistotestausta tehdään manuaalisesti testaajan toimesta, hän voi improvisoida toimintaansa ja huomioida myös odottamattomia asioita (Kaner, Bach ja Pettichord 2002). Manuaalista testausta tekevä henkilö voi tarvittaessa keskeyttää testitapauksen suorittamisen tai muuttaa suoritustapaa, kun taas automaatiotestit pyrkivät toistamaan niille ennalta määriteltyä suoritustapaa. Yhtä manuaalisesti suoritettua testitapausta ei siis voi suoraan rinnastaa yhteen automaation avulla suoritettuun testiin. Automaattisissa testeissä ei välttämättä pystytä varautumaan ohjelmiston mahdollisiin virhetilanteisiin, jotka manuaalitestaaja voi usein ratkaista testitapauksia suorittaessaan. Tällaisessa tilanteessa automaattisten testien suoritus voi pysähtyä virhetilanteeseen, kun taas manuaalisia testitapauksia voidaan pystyä jatkamaan.

## 2.4 Automaattinen testaus

Testiautomaatiota voidaan kuvata siten, että testausaktiviteetteja tai niiden osia suoritetaan koneellisesti ajamalla niin sanottuja testiskriptejä (Dustin, Rashka ja Paul 1999). Tällöin itse testitapausten suoritus, kuten ohjelmiston toiminnallisuuksien käyttäminen, tapahtuu ohjelmallisesti ilman, että testaaja itse antaisi testien suorittamisen aikana käskyjä ohjelmistoon esimerkiksi käyttöliittymän kautta. Tällöin esimerkiksi testitapausten suoritusnopeus voi poiketa merkittävästi manuaalisesta suorittamisesta.

Automaattisessa testauksessa käytetään apuna erityisiä automaatiotestaukseen tarkoitettuja ohjelmistoja ja useimmat testitapaukset suoritetaan lähes kokonaan ilman testaajan vaikuttamista niihin testien suorituksen aikana (Oliinyk ja Oleksiuk 2019). Testitapausten suorittamista automaattisesti tulee kuitenkin tarkkailla esimerkiksi mahdollisten virhetilanteiden varalta (Taipale ym. 2011).

Regressiotestaus on toimenpide, jonka avulla tarkastellaan ohjelmiston toimivuutta, kun ohjelmistoon on tehty muutoksia olemassa olevien toimintojen testaamisen jälkeen (ISO/IEC 29119-1 2022a). Täten regressiotestauksella pyritään tarkistamaan, onko uusilla muutoksilla ollut haitallisia vaikutuksia ohjelmiston aiemmin kehitettyihin toiminnallisuuksiin ja ominaisuuksiin. Testausta automatisoimalla voidaan vähentää manuaalisen regressiotestauksen

määrää. Koska regressiotestauksella pyritään varmistamaan nimenomaan niiden toimintojen oikeellisuus, joihin ei ole tehty muutoksia, sopii automaation hyödyntäminen regressiotestaukseen hyvin.

Automaattisista testeistä voidaan hyötyä tilanteessa, jossa samoja muodostettuja automaatiotestejä voidaan käyttää uudelleen useita kertoja. Yleisesti ottaen, jos samoja testitapauksia aiotaan suorittaa ainakin 5 kertaa, testien automatisointi voi olla taloudellisesti kannattavaa (ISO/IEC 29119-1 2022a). Testitapausten automatisoiminen ei kuitenkaan kannata, jos muodostettavia automaattitestejä pitää jatkuvasti muuttaa (Kasurinen 2015). Tällöin automatisoinnilla säästetty työmäärä voi siirtyä manuaalisen testauksen suorittamisesta automaattitestien vaatimaan korjaamiseen. Käytännössä automatisoitavat testitapaukset tulisi siis valita siten, että testattaviin ohjelmiston osiin ei kohdistu painetta muutoksille. Automaatiotestit kuitenkin vaativat myös ylläpitoa, mikä tulee huomioida siirryttäessä manuaalisesta testauksesta kohti automaatiota (Taipale ym. 2011). Mahdolliset muutokset ohjelmiston toiminnassa voivat aiheuttaa tarvetta automaatiotestien muuttamiselle tai uudelleen muodostamiselle. Uusien toimintojen lisääminen vaatii testitapausten suunnittelun lisäksi automaattitestien muodostamiseen vaadittavaa työtä. Testitapausten ja testiautomaatiojärjestelmän huolellinen dokumentointi on tärkeää, koska muutoin tehtäviin muutoksiin voi tuhraantua paljon aikaa, jota voitaisiin käyttää käsin tehtävään testaukseen (Kasurinen 2015). Ihmisen toimintaa vaaditaan lisäksi usein testien luomistyöhön automatisointityön alkuvaiheessa ja suoritettujen automaatiotestien tulosten tarkastamiseen.

Testiautomaatio ei korvaa manuaalista testausta, vaan tarkoitus on täydentää sitä (Kasurinen 2015). Testiautomaatiota voidaan käyttää apuna haluttaessa suorittaa suuria määriä testitapauksia lyhyessä ajassa (Umar ja Zhanfang 2019). Manuaalista testausta kuitenkin tarvitaan esimerkiksi monimutkaisemmissa testitapauksissa, joita ei voida toistaa yhä uudelleen samanlaisina (Taipale ym. 2011). Erityisesti jos testitapauksien suorittaminen vaatii paljon kohdealueen tietämystä, tulee testien muuntaminen manuaalisista automaattisiksi olemaan hankalaa. Automaattisiksi testeiksi kannattaa muuttaa siis ensisijaisesti yksinkertaisia ja usein toistettavia testitapauksia.

Automaation käyttö apuna testaamisessa on viime vuosina yleistynyt kaikilla eri testaamisen tasoilla (Hynninen ym. 2018). Kun ohjelmistokehitysprosessin resurssit ovat rajalliset, ei

testiautomaation kehitykselle kuitenkaan välttämättä riitä sen vaatimia ylimääräisiä resursseja (Taipale ym. 2011). Testiautomaatiotyökalujen pilotointi ja käyttöönotto yhdessä osaavien testaajien kouluttamisen ja palkkaamisen kanssa vievät resursseja (ISO/IEC 29119-1 2022a). Yleensä testiautomaation käyttäminen vaatii testaajilta työkalujen tuntemuksen lisäksi ohjelmointitaitoja.

Ketterässä ohjelmistokehityksessä on tavanomaista, että kehitettävästä ohjelmistosta tuotetaan lyhyissä jaksoissa iteratiivisesti uusia testattuja versioita (Abrahamsson ym. 2017). Myös testausprosessin tulee tällöin pystyä seuraamaan ohjelmistokehityksen iteraatioita ja täten muodostettavien uusien versioiden testauksen tulee olla tehokasta (Collins, Dias-Neto ja Lucena Jr 2012). Pitkäaikainen ja useita iteraatioita sisältävä ohjelmistokehitysprosessi voi siis hyötyä automaatiosta enemmän kuin lyhyempi. Automaation avulla voidaan tuoda testaamiseen tehokkuutta, mutta testiautomaation hyödyllisyyttä voi myös laskea jatkuvasti muuttuva ohjelmisto, jos automaatiotestejä joudutaan muuttamaan. Niin kutsutussa vesiputousmallissa ohjelmistoa kehitetään toisistaan eriytetyissä vaiheissa, jotka voidaan jakaa vaatimusmäärittelyyn, suunnitteluun, toteutukseen ja testaamiseen (Adenowo ja Adenowo 2013). Tämäntapaisessa lineaarisesti etenevässä ohjelmistokehityksessä testiautomaatiota voidaan kehittää ilman riskiä ohjelmiston osioiden muuttumisesta. Toisaalta testiautomaation luomisesta saatavat hyödyt voi jäädä saavuttamatta, jos testitapauksia ei aiota suorittaa useita kertoja uudelleen. Tällöin manuaalisen testauksen hyödyntäminen voi olla kannattavampaa.

## 2.5 Robot Framework

Robot Framework on avoimeen lähdekoodiin perustuva automaattisten testitapausten muodostamisen mahdollistava sovelluskehys (“Robot Framework Documentation” 2021). Sen avulla on mahdollista testata monilla erilaisilla teknologioilla toteutettuja ohjelmistoja. Robot Framework pohjautuu Python-ohjelmointikieleen ja mahdollistaa testitapausten luomisen käyttäen erilaisia avainsanoja. Avainsanoja voidaan toteuttaa useilla ohjelmointikielillä, kuten Javalla tai Pythonilla. Avainsanoja yhdistelemällä saadaan muodostettua robottitestejä, joilla voidaan suorittaa monimutkaisiakin operaatioita testattavan ohjelmiston käyttöliittymässä. Avainsanalla voidaan tässä tapauksessa tarkoittaa esimerkiksi tietyn ohjelmiston

painikkeen painamista, hiiren siirtämistä tai tietyn näkymän avaamista. Tämän lisäksi avainsanojen avulla voidaan esimerkiksi käynnistää ja sammuttaa prosesseja, kuten testauksen alainen ohjelmisto. Tulee kuitenkin huomioida, että kehityksenalaisessa ohjelmistossa esimerkiksi painikesijoittelun muutokset voivat aiheuttaa muutospaineen testeille, joissa käyttöliittymän osia kuten painikkeita käytetään hiirellä klikaten.

Robot Frameworkilla suoritetuista testitapauksista muodostuu automaattisesti HTML-formaatissa lokit ja raportit, jotka sisältävät tarkat tiedot esimerkiksi testitapausten suoritusajoista ja mahdollisista virheistä (“Robot Framework Documentation” 2021). Tämä helpottaa automaation avulla suoritettujen testien tulosten tarkastelua ja mahdollisten virheiden paikantamista.

# 3 Kestävyystestaus

Ohjelmistolle määritellään kehitysvaiheessa oleellisia laatuominaisuuksia ja niille tavoitetasoja. Nämä laatuominaisuudet realisoituvat laatuvaatimuksina ja myös testitapauksina, joiden avulla laatua mitataan. Laatumittaristossa voidaan tunnistaa käsitteitä kuten suorituskyky, toiminnallisuus, käytettävyys, luotettavuus, siirrettävyys ja ylläpidettävyys (ISO/IEC 25010 2011). Kestävyystestauksella voidaan tarkastella ohjelmiston suorituskykyä suhteessa sille asetettuihin laatuvaatimuksiin käytettäessä ohjelmistoa pitkäkestoisesti jatkuvan rasituksen aikana. Vasteaikojen pidentyminen tai järjestelmäresurssien käytön suureneminen kuvastavat suorituskyvyn heikentymistä (Laaber 2019).

Yksi esimerkki tarkkailtavista ohjelmistojen käyttämistä järjestelmäresursseista on keskusmuisti. Yksi muistinkäyttöön liittyvä mahdollinen ongelma on muistivuodot, joissa ohjelmiston muistinkäyttö lisääntyy vähitellen ja ohjelmisto hidastuu, kunnes muistin loppuessa ohjelmisto voi kaatua (Ghanavati ym. 2020). Kestävyystestauksen näkökulmasta muistinkäyttö on siis oleellinen testattava ominaisuus ja testausjaksoista pyritäänkin tekemään riittävän pitkiä, jotta tällaiset ongelmat saataisiin ilmaantumaan.

Alaluvussa 3.1 käydään läpi kestävyystestauksen piirteitä ja kestävyystestauksessa ohjelmistoista tarkasteltavia seikkoja. Alaluvussa 3.2 kuvataan, miten kestävyystestausta tehdään ja mitä esimerkiksi testitapausten kuormituksen valinnassa tulee huomioida. Alaluvussa 3.3 käydään läpi kestävyystestauksessa käytettävien työkalujen valinnassa huomioitavia seikkoja.

## 3.1 Kestävyystestauksen määritelmä

Ohjelmiston suorituskykyä voidaan testata hyvin erilaisilla painotuksilla. Yleisesti ottaen suorituskykytestauksessa ohjelmistoa altistetaan testauksen aikana niin sanotusti tyypilliselle kuormalle (ISO/IEC 29119-1 2022b). Suorituskykytestauksen tavoitteista riippuen ohjelmistoa kohtaan aiheutettava kuormitus voidaan asettaa vastaamaan ohjelmiston tavanomaista, minimaalista tai suurinta mahdollista sille jokapäiväisessä käytössä odotettavaa kuormaa. Omaksi testaustyypikseen voidaan erottaa esimerkiksi rasitustestaus (engl. stress testing),

jossa ohjelmistoa altistetaan normaalissa käytössä odotettavan kuormituksen ylittävälle kuormalle (ISO/IEC 29119-1 2022a). Tällaista testausta voidaan suorittaa myös ajamalla ohjelmistoa järjestelmällä, joka alittaa esimerkiksi muistin määrän tai prosessorin laskentatehon osalta ohjelmistolle määritetyt järjestelmävaatimukset.

Myös kestävyystestauksessa tarkastelun kohteena on ohjelmiston suorituskyky. Kestävyystestaus on testauksen tyyppi, jolla selvitetään testattavan tuotteen suoriutumista sen toiminnoista ohjelmiston toiminnalle määriteltyjen resurssien puitteissa (ISO/IEC 29119-1 2022a). Myös järjestelmänä, jolla ohjelmistoa suoritetaan, käytetään ohjelmistolle asetetut vaatimukset täyttävää kokoonpanoa. Kuormitus pyritään kestävyystestauksessa pitämään sellaisena, että se vastaisi mahdollisimman hyvin ohjelmiston normaalia käyttötilannetta. Erityisenä yksityiskohtana kestävyystestaukseen liittyy ohjelmiston jatkuva suorittaminen pitkäkestoisten testausjaksojen ajan.

Kestävyystestauksen tarve riippuu ohjelmistolle asetettavista vaatimuksista. Kestävyystestauksen merkitys luonnollisesti korostuu ohjelmistoissa, joita halutaan käyttää pitkäaikaisesti. Myös suuria käyttäjämääriä yhtäaikaisesti palvelevilla ohjelmistoilla suorituskykyyn liittyvät seikat nousevat erittäin tärkeäksi ja ongelmat suorituskyvyssä voivat aiheuttaa suuria taloudellisia tappioita (Alghmadi ym. 2016).

## 3.2 Miten kestävyystestausta tehdään

Kestävyystestauksen ollessa yksi järjestelmätestauksen testaustyypeistä sijoittuu se vaiheeseen, jossa järjestelmää testataan pienempien osien sijaan kokonaisuutena (Kasurinen 2015). Suorituskykytestaus, jonka yhdeksi testaustyypiksi kestävyystestaus sijoittuu, eroaa toiminnallisuuden testaamisesta monilta osin (Weyuker ja Vokolos 2000). Keskittyminen kohdistuu aiheutetun kuormituksen järjestelmän toiminnalle aiheuttamiin vaikutuksiin toimintojen oikeellisuuden tarkkailun sijaan. Kestävyystestaukseen valittava kuormituksen suuruus onkin yksi päätettävistä seikoista. Luotaessa järjestelmälle normaalia käyttöä kuvaavaa kuormitusta tulee selvittää, millainen tämä normaali kuormitus on. Kuormitus on seurausta esimerkiksi ohjelmistolla suoritetuista toiminnoista. Yleisesti käytettyjen toimintojen ja niiden käytön toistuvuuden selvittäminen muodostuu toiseksi tarkasteltavaksi seikaksi. Tyypillisesti näiden

määrittämiseen vaaditaan kohdealueen asiantuntijan tietämystä (Weyuker ja Vokolos 2000).

Itse kuormitus valitaan kestävyystestaukseen siis tyypillisimmin keskimääräistä kuormitusta kuvaavana (Weyuker ja Vokolos 2000). Kuormitus voidaan mahdollisesti valita ohjelmiston käytöstä saadun historiatiedon perusteella, mutta välttämättä tällaista aiempaa historiatietoa ohjelmiston käytöstä ei ole saatavilla päätöksenteon tueksi. Keskimääräinen kuormitus voidaan myös valita hyvin eri tavoin riippuen siitä, kuinka pitkältä ajanjaksolta keskimääräistä kuormaa laskelmoidaan. Ohjelmistolla on usein esimerkiksi intensiivisempiä käyttöjaksoja, joiden keskimääräinen kuorma voi eroa merkittävästi laajemman aikajakson kuormituksesta.

Myös testausympäristön valinta on osa testausprosessia. Testausympäristöä valittaessa tulee huomioida testattavat ominaisuudet. Jos mitataan esimerkiksi keskimääräistä vasteaikaa loppukäyttäjän käyttäessä ohjelmistoa, tulee ympäristön vastata mahdollisimman hyvin käyttäjän ympäristöä (Weyuker ja Vokolos 2000). Lisäksi testausympäristö tulee olla sellainen, että testitapausten suoritusten tulokset on mahdollista saada kerättyä.

Ohjelmiston suoriutumisen mittaamiseksi tulisi ennen testauksen suorittamista olla selvillä ohjelmistolle asetetuista vaatimuksista (Weyuker ja Vokolos 2000). Mitattaessa esimerkiksi resurssienkäyttöä voitaisiin hyötyä ennalta asetetuista vaatimuksista, jotka asettavat tietyt raja-arvot, joita ohjelmiston ei tulisi resurssienkäytön osalta ylittää. Näiden vaatimusten myös tulisi olla sellaisia, että niihin liittyviä arvoja on mahdollista mitata testattavasta järjestelmästä.

Itse testitapaukset tulee luoda toiminnallisesta testauksesta poiketen siten, että pääpaino on kuormituksen toteuttamisessa ja sen ajoittamisessa (Weyuker ja Vokolos 2000). Järjestelmästä tulee myös pyrkiä tunnistamaan kuormitukselle alttiita osia tai prosesseja testitapauksia varten. Kestävyystestaukseen valittavien testitapausten määrän hallitsemisessa auttaa, jos valinta kohdistetaan lähinnä merkittävimmän vaikutuksen järjestelmän suorituskykyyn aiheuttaviin prosesseihin (Weyuker ja Vokolos 2000).

## 3.3 Kestävyystestauksen työkalut

Kestävyystestauksen työkalujen valinta lähtee liikkeelle tarpeesta ja testauksen tavoitteesta. Ohjelmiston suorituksen aikana mitattavia resursseja voivat olla työaseman muistinkäyttö, viive toimintojen suorituksessa tai esimerkiksi prosessorin kuormitus. Ohjelmistolta voidaan haluta nopeutta ja toimintavarmuutta ja kestävyystestauksesta onkin tullut hyvin tärkeä osa ohjelmistokehitystyötä (Srivastava, Kumar ja Singh 2021). Parhaan työkalun valinta riippuu kuitenkin niin käytettävästä budjetista kuin järjestelmästä ja esimerkiksi käytettävissä olevien testaajien määrästä.

Kestävyystestausta suoritettaessa ohjelmistoa siis kuormitetaan, joten vaaditaan työkaluja kuormituksen aiheuttamiseksi sekä kuormituksen mittaamiseksi. Työkalujen valintaa ohjaa myös se, mitä järjestelmäresursseja seurattavaksi valitaan. On myös merkitystä sillä, suoritetaanko seuranta manuaalisesti vai automaattisesti. Manuaalista testausta tekevän henkilön voi olla helpompi seurata resurssinkäytön yksityiskohtia reaaliaikaisesti graafisen käyttöliittymän avulla. Automaattisissa testitapauksissa tärkeämmäksi seikaksi voi muodostua resurssinkäytön tallentaminen lokitietoihin mahdollistaen tietojen myöhemmän tarkastelun.

Muistiongelmia tutkittaessa pitää työkalun pystyä luotettavasti kertomaan yksityiskohtia muistinkäytöstä. Muistivuotojen ilmaantuvuutta ja keinoja niiden toteamiseen tutkittaessa on huomattu sekä ohjelmakoodin analysoinnin että ohjelmiston muistinkäytön ajonaikaisen seurannan tuoneen hyviä tuloksia. Staattisen testauksen suurimpana heikkoutena voidaan nähdä niin sanottujen väärien positiivisten löydösten suuri ilmaantuvuus (Ghanavati ym. 2020). Selkeästi suurin osa muistinkäyttöön liittyvien virheiden toteamisesta on huomattu tapahtuvan tarkasteltaessa ohjelmiston käyttäytymistä ajonaikaisesti (Ghanavati ym. 2020). Itse muistinkäytön tarkasteluun on käytetty usein onnistuneesti kolmannen osapuolen muistianalysoijia (Ghanavati ym. 2020).

Java-ohjelmiston ollessa kyseessä voidaan muistinkäyttöä tarkkailla esimerkiksi *jmap*-työkalua käyttäen (“Oracle Java Documentation - Jmap” 2014). Työkalun avulla voidaan muodostaa kuvaus esimerkiksi tietyn Java-prosessin varaamasta kekomuistista. Työkalulla muodostettuja muistivedoksia taas voidaan tarkastella useilla eri työkaluilla. Yksi näistä työkaluista on *Eclipse Memory Analyzer (MAT)*, jonka avulla muistivedoksista voidaan tutkia esimerkik-

si muistivuotojen ilmenemistä (“The Eclipse Memory Analyzer” 2023). *MAT* on avoimeen lähdekoodiin perustuva työkalu, jota pääasiassa hyödynnetään muistinkäytön ongelmien selvittämisessä.

# 4 Tutkielman tavoitteet ja tutkimusmenetelmä

Tutkielman kohteena olevan Java-sovelluksen testausprosessia haluttiin kehittää vastaamaan paremmin yrityksen tavoitteita. Manuaalisen testauksen työmäärän vähentämistä ja järjestelmän virheiden, kuten muistivuotojen, löytämistä tavoiteltiin testausprosessin automatisoinnilla.

Alaluvussa 4.1 määritellään tutkimusmenetelmä ja tutkimuksen tavoitteet. Tutkimusmenetelmää ja sen soveltamista tässä tutkimuksessa kuvataan pääpiirteittäin alaluvussa 4.2. Alaluvussa 4.3 kuvaillaan tutkimuksen kohteena olevaa järjestelmää, kuten sen rakennetta ja muodostamisessa käytettyjen teknologioiden mukanaan tuomia haasteita ja erityispiirteitä. Lisäksi alaluvussa 4.4 esitellään kestotestiprosessin lähtötilanne. Näiden yksityiskohtien avulla annetaan lähtökohdat sille, miten järjestelmään liittyvää testausta tulee lähestyä.

## 4.1 Tutkimuksen tavoitteet ja suoritustapa

Tutkielman tavoitteena oli luoda testauskonsepti, jolla toteutettaisiin kehitysaikaisen sovelluksen kestotestausta menetelmillä, jotka sopivat sen toimintojen luotettavuuden, kuten muistinkäytön virheettömyyden, mittaamiseen ja todentamiseen kestävyystestauksen näkökulmasta. Tutkimuskysymykseksi muodostui, millaisen järjestelmän avulla saadaan kehitettyä nykytilanteen testausta siten, että automaattisten testien avulla voidaan onnistuneesti korvata lähtötilanteessa manuaalisesti suoritettavaa kestävyystestausta.

Tutkimuksen kohteena olevaa järjestelmän testausprosessia pyritään kehittämään korvaamalla manuaalisen testauksen elementtejä automaattisilla testeillä. Yhtenä tavoitteena on saada kestävyystesteistä sellaisia, että niitä voidaan suorittaa useammin ilman jatkuvaa tarvetta ihmistestaajan läsnäolosta. Lähtötilanteessa kestävyystestijakso suoritetaan kaksi kertaa vuodessa ja jakson aikana suoritetaan kestävyystestausta lähinnä manuaalisesti.

Ohjelmistolle suoritetuissa kestävyystesteissä on ohjelmiston toiminnassa havaittu puutteita, kuten muistivuotoisuutta, jonka ilmenemistä halutaan testata. Automaattisella kestävyystestauksella halutaan löytää ohjelmistolle asetettuihin laatuvaatimuksiin nähden poikkeamia,

kuten muistivuotoja, manuaalista testausta tehokkaammin.

## 4.2 Tutkimusmenetelmä

Tutkimusmenetelmänä käytettiin suunnittelututkimusta, joka on hyvin asianmukainen ja suositu tutkimusmenetelmä tilanteissa, joissa tutkimuksen tavoitteena on luoda tietotekninen artefakti (Hevner ja Chatterjee 2010). Suunnittelututkimuksen avulla on mahdollista kehittää sovellutuksia ratkomaan ympäröivän maailman todellisia ongelmia, joita esimerkiksi yritysten ohjelmistoratkaisuissa voidaan havaita. Ohjelmistoteknologisia ratkaisuja käytetään parantamaan esimerkiksi yritysmaailman toimijoiden työskentelyn tehokkuutta (Hevner ja Chatterjee 2010).

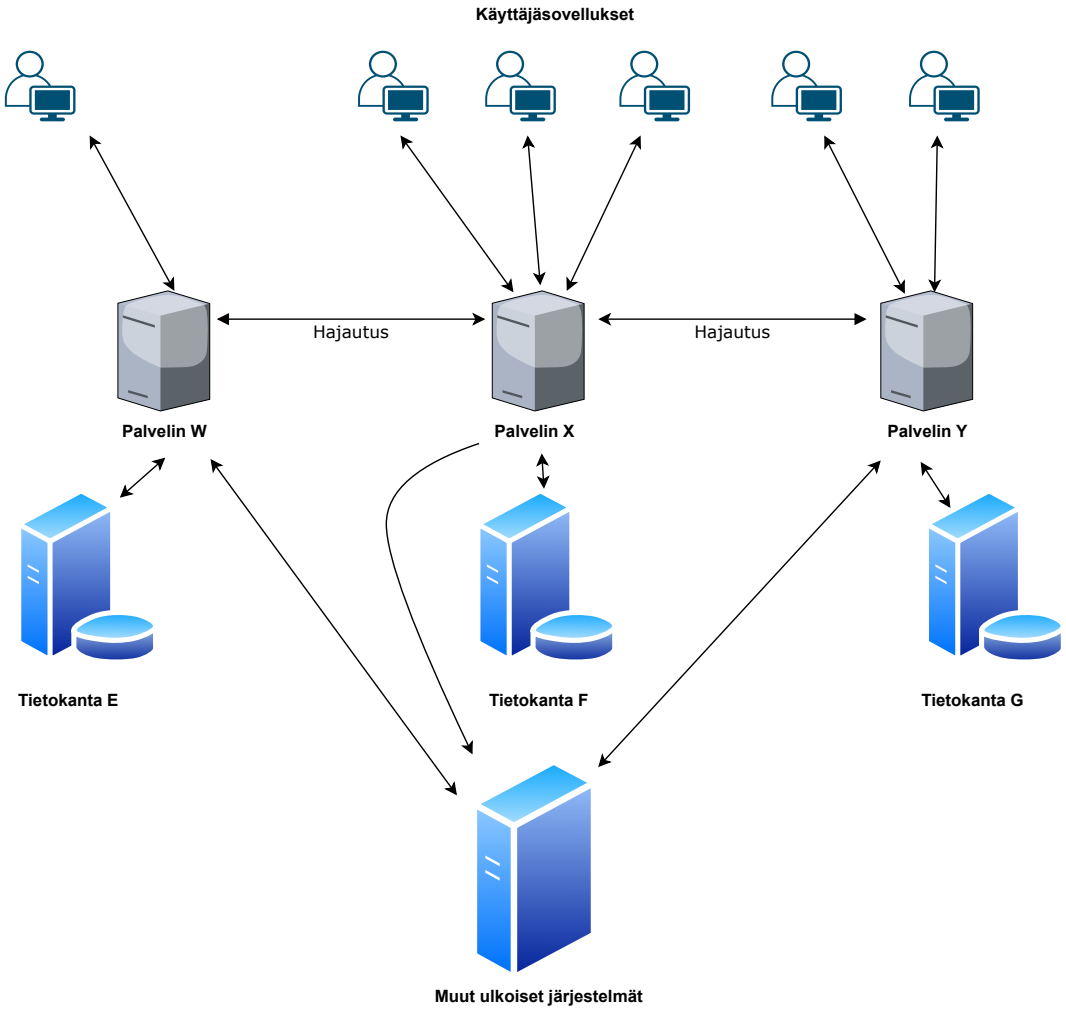
Suunnittelututkimuksen ytimessä on ongelmien ratkominen (Hevner ja Chatterjee 2010). Ongelman ratkaiseminen vaatii itse ongelman analysointia, ratkaisujen suunnittelua ja pyrkimystä myös toteuttaa toimivia ratkaisuja. Tähän pyritään valjastamaan ohjelmistoteknologisia työkaluja, joilla nuo ongelmat tehokkaasti ratkaistaan. Tämän toiminnan avulla voidaan myötävaikuttaa siihen, että ohjelmistokehitys menisi luotavien innovaatioiden avulla eteenpäin.

Suunnittelututkimuksessa täytyy pystyä muodostamaan artefakti, joka voi ongelmasta riippuen olla tietotekninen tuote tai vaihtoehtoisesti sen avulla voidaan esimerkiksi kuvailla ongelman ratkaiseva malli (Hevner ja Chatterjee 2010). Artefaktin tuottamisen lisäksi tärkeää on arvioida saatu artefakti ja esitellä suoritettu tutkimus tuloksineen tehokkaasti ja ymmärrettävästi. Näin luodaan pohjaa myös aiheeseen liittyvälle jatkotutkimukselle.

## 4.3 Tutkimuksen kohde

Tutkimuksen kohteena oleva ohjelmisto on reaaliaikaista tilannekuvaa käyttäjälle esittävä johtamisjärjestelmä, jonka rakennetta esitellään kuviossa 1. Ohjelmisto on asiakas-palvelin -mallin järjestelmä. Yksi toimipaikka kattaa yhden palvelimen, johon voi yhdistyä useita asiakkaita eli käyttäjäsovelluksia. Toimipaikkojen välille on toteutettu hajautusyhteys. Tietyt toiminnallisuudet mahdollistavat tiedon jakamisen muiden ohjelmistoa käyttävien toimi-

joiden kanssa. Käyttäjän on mahdollista syöttää järjestelmään tilannetietoa käyttöliittymän kautta, minkä lisäksi palvelin ottaa vastaan ja prosessoi tietoja muista ulkoisista järjestelmistä. Tästä muodostuu tilannekuva, joka esitetään käyttäjälle karttakäyttöliittymän avulla. Käyttäjän on mahdollista tallentaa järjestelmään oletusasetuksia, jotka liittyvät esimerkiksi käyttöliittymän dialogien asetteluun ja pikanäppäinten hallintaan.




Kuvio 1. Tutkimuksen kohteena oleva järjestelmä

Tällaisella asiakas-palvelin -mallin järjestelmällä tiedon onnistunut hajautuminen on yksi kriittinen osa järjestelmää. Mahdolliset ongelmat siinä johtavat eroihin eri käyttäjille esitettävässä tilannekuvassa. Tällainen voi johtaa suuriin ongelmiin ohjelmiston käyttäjien väli-

sessä yhteistyössä, joten hajautuksen toimivuuden testaaminen on tärkeää. Hajautuksen toimivuuden tarkkailemiseksi on järjestelmää testattava yhdenaikaisesti vähintään kahdella rinnakkain hajautuksessa olevalla toimipaikalla.

Testauksen kohteena oleva järjestelmä on ohjelmoitu suurimmaksi osaksi Java-ohjelmointikieltä käyttäen. Java-ohjelmistoille onkin voitu osoittaa ominaispiirteitä, jotka on hyvä ottaa huomioon suunniteltaessa kestävyystestausta.

Ohjelmiston muistinkäyttö on yksi mahdollisista ongelmakohdista Java-ohjelmistoilla. Muistinkäytön hallintaan Java tarjoaa kehittäjän avuksi muun muassa automaattisen roskienkeräyksen (Jones ja Lins 1996). Tämä on tarkoitettu helpottamaan muistinhallintaa, sillä roskienkerääjä vapauttaa automaattisesti ajonaikana muistia poistaen muistista olioita, joihin sovelluksessa ei enää viitata. Muistinkäyttöön liittyviä ongelmia tämä ei kuitenkaan kokonaan poista ja roskienkeruun sisältäviin ohjelmointikieliin sisältyy silti mahdollisuus esimerkiksi muistivuodoille (Distefano ja Filipovic 2010). Tämä on seurausta siitä, ettei automaattinen roskienkeruu pysty vapauttamaan muistia esimerkiksi tilanteissa, joissa ohjelmakoodissa jää viittaus olioon, jota ei enää kuitenkaan käytetä (Ghanavati ym. 2018).

Muistivuotojen lisäksi Java-ohjelmistoissa on todettu mahdollisuus myös muille resurssivuodoille, jotka liittyvät esimerkiksi yhteyksien ja säikeiden puutteelliseen sulkemiseen (Ghanavati ym. 2020). Tällaiseen tilanteeseen apua ei myöskään ole saatavilla automaattisesta roskienkeruusta, vaan ohjelmistokehittäjän tulee huolehtia näistä itse.

Koska Java-sovelluksissa tapahtuvissa muistivuodoissa ohjelmiston muistinkäyttö yleensä lisääntyy vähitellen, on tällaisen esiintymistä parasta testata pitkään jatkuvilla testausjaksoilla. Kestävyystestausjaksolta vaadittava pituus on kuitenkin yksilöllinen ja onkin hyvä tarkkailla, saavutetaanko testaukseen käytetyillä lisätunneilla lisähyötyä esimerkiksi löydettyjen virheiden muodossa (Collard 2005).

Käytännössä ohjelmiston rakentamisessa käytetty Java-teknologia mahdollistaa ohjelmiston käytön hyvin monenlaisissa ympäristöissä. Ohjelmiston elinkaaren suunnitellaankin olevan pitkä. Käyttäjäsovellus on suunniteltu käytettäväksi Microsoft Windows -käyttöjärjestelmässä. Palvelinohjelmistoa suoritetaan Linux-ympäristössä.

Järjestelmä on toiminnallisuuksiltaan laaja ja sitä laajennetaan yhä kehitystyön jatkuessa. Manuaalisesti suoritettava testaus vaatii paljon työvoimaa ja esimerkiksi kestävyystestaukseen käytettävä työmäärä ei kuitenkaan ole kasvatettavissa loputtomasti.

## 4.4 Kestotestiprosessin lähtötilanne

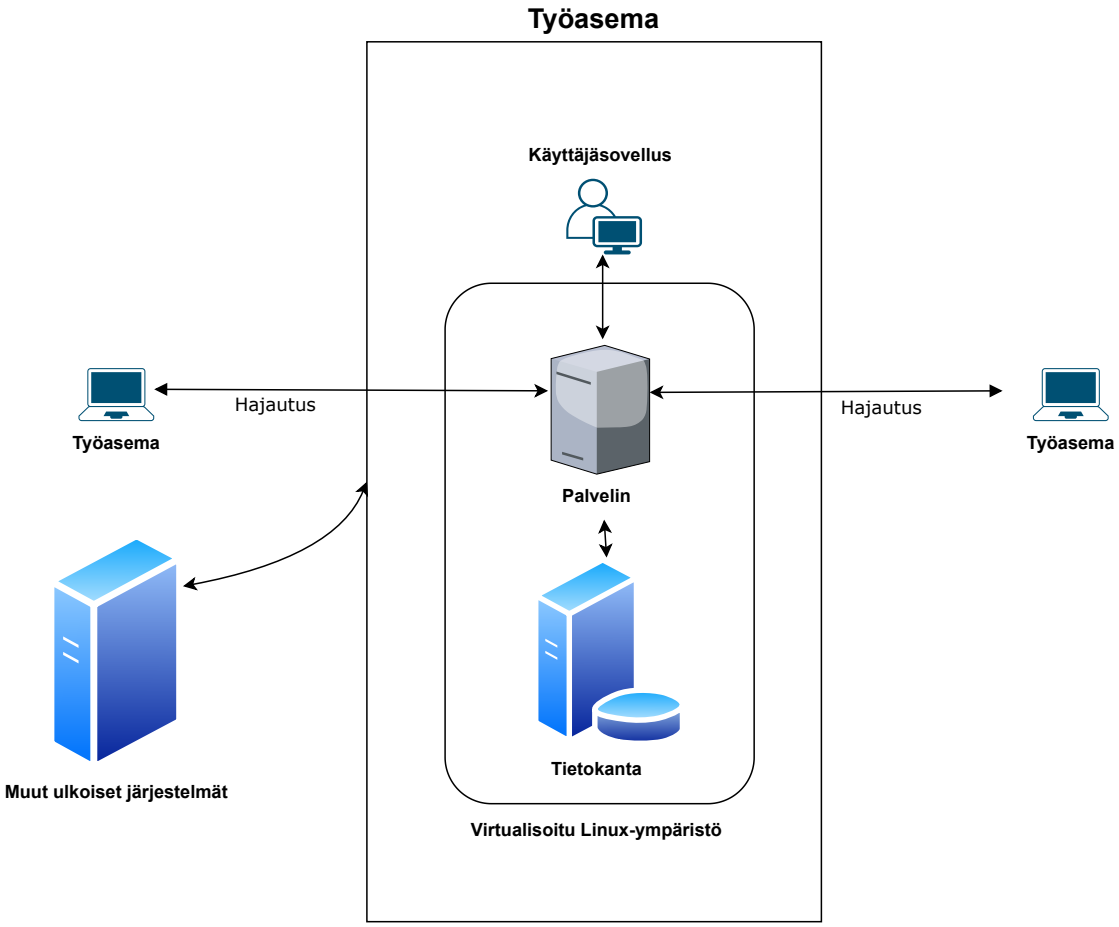
Ohjelmiston kestotestijakso on ennalta määrätyn mittainen aikajakso osana ohjelmiston järjestelmätestausta. Kestotestijaksoja järjestetään, koska kyseessä on korkean saatavuuden järjestelmä, jota halutaan voida käyttää yhtäjaksoisesti kuukausia tai jopa vuosia. Kestotestin aikana ohjelmistosta suoritetaan samanaikaisesti vaihtelevaa määrää käyttäjäsovelluksia ja sovelluspalvelimia. Ohjelmiston sovelluspalvelimet toimivat keskenään hajautetusti ja jakavat reaaliajassa dataa muihin testissä mukana oleviin ohjelmistoihin. Ohjelmistot ovat toiminnassa kestotestijakson ajan ympäri vuorokauden. Testausjakson aikana järjestelmään syötetään muun muassa generoitua dataa ulkoisista järjestelmistä, jolla voidaan kuvastaa esimerkiksi kuviteltua tilannekuvaa.

Lähtötilanteessa testaajat suorittivat jakson aikana ennalta määriteltyjä testitapauksia, joissa he syöttivät manuaalisesti normaalia käyttötilannetta kuvaavia syötteitä käyttöliittymien kautta. Testitapaukset ovat sisältäneet monia erilaisia toiminnallisuuksia, jotka järjestelmään on toteutettu asiakasvaatimusten perusteella. Niihin on kuulunut myös esimerkiksi yhteyskatkojen luomista työasemien ja palvelimen välillä.

Kestotestijakson pituus lähtötilanteessa on ollut noin kuukausi, jonka aikana testaajat ovat tarkkailleet ohjelmiston toimintaa eri tavoin. Testausta on viimeisimmillä testausjaksoilla suoritettu kolmella fyysisellä työasemalla, joilla kaikilla on ollut käytössä ohjelmistosta yksi käyttäjäsovellus ja tämän lisäksi virtualisoitu palvelin. Työasemia on kuvattu kuviossa 2. Tietokoneet on testausjaksoilla sijoitettu vierekkäin siten, että testaaja on voinut seurata kaikkien työasemien näyttöjen tapahtumia esimerkiksi suorittaessaan yhteyskatkoksia aiheuttavia testitapauksia. Testaaja on näin voinut seurata testitapauksien vaikutusta ohjelmiston toimintaan ja ohjelmiston toipumista katkoksista.

Työasemat ovat olleet kestotesteissä virtuaalisten sijaan fyysisiä, jotta käyttäjäsovelluksen resurssinkäytön seuraaminen olisi paremmin toteutettavissa. Kestotestissä on käytetty kan-

nettavia tietokoneita ja tällöin mahdollisten sähkökatkojen aikana laitteet pystyvät toimimaan pitkiä aikoja akun varassa, eikä erillistä varavirtalähdettä tarvita. Koneiden yhteyteen on lisäksi sijoitettu verkkokytkin, jonka avulla testaaja on voinut aiheuttaa yhteyskatkoksia palvelinten välille.




Kuvio 2. Testauksessa käytetyn työaseman kuvaus

Kahdella edeltävällä kestotestausjaksolla ovat testaajat manuaalista testausta suorittaessaan merkinneet yhteiseen dokumenttiin testitapauksien kuvaukset ja kestotestijakson suorituspäivämäärät. Tähän dokumenttiin testaajat ovat merkinneet kaikkien suorittamiensa testitapausten kohdalle kuittauksen testin tuloksesta ja suorituspäivästä. Mahdollisista virheistä dokumenttiin on merkitty viittaus *Jira*-projektinhallintatyökaluun. Näin testausjakson aikana ja sen jälkeen on ollut mahdollista seurata, minä päivinä tietty testaaja on suorittanut

tiettyjä testitapauksia. Jokaisella testauskerralla ei ole suoritettu kaikkia testitapauksia niiden runsaan määrän takia. Tämän sijaan testaaja on jokaisella testauskerralla omaa harkintaa käyttäen valinnut suoritettavat testitapaukset. Ohjelmiston toiminnassa virheen huomattuaan testaaja on merkinnyt virheen dokumenttiin ja lisännyt viitteen *Jira*-työkaluun muodostettuun huomioon, johon hän on laatinut tarkemman virhekuvauksen dokumentointia varten.

Taulukossa 1 kuvataan kahden edeltävän kestotestijaksojen pöytäkirjoista kerätyt arvot, kuten testitapausten lukumäärät, testitapausten suoritusten keskiarvot testauspäivinä, yhden testitapauksen suoritusten lukumäärän keskiarvo sekä raportoidut virheet kestotestijaksoilta. Raportoidut virheet kuvaavat tilanteita, joissa joko yksittäinen toiminto tai ohjelmiston havainnoitu käyttäytyminen testitapausta suoritettaessa ei ole testaajan mukaan vastannut ohjelmistolle asetettuja vaatimuksia. Laskuissa on huomioitu vain sellaiset päivät, jolloin on suoritettu ainakin yksi testitapaus. Tämä siitä syystä, ettei tuloksia vääristäisi kestotestijakson päivät, jolloin manuaalista testausta ei syystä tai toisesta suoritettu. Vaikka strukturoitua lähestymistapaa kestotestien aikana suoritettuihin manuaalisiin testitapauksiin on käytetty vasta kahdella kestotestijaksolla, on tuloksista silti nähtävissä, että testitapausten suorituskertojen määrän lisääminen on johtanut myös raportoitujen virheiden suurempaan määrään. Testitapausten määrässä on eroa kolmen testitapauksen verran ja nuo kyseiset testitapaukset on jätetty pois viimeisimmältä kestotestijaksolta, sillä niissä on ollut osittain päällekkäisyyttä muiden testitapausten kanssa. Muiden testitapausten kuvausten osalta kestotestijaksot ovat olleet identtiset. Näitä edeltävillä kestotestijaksoilla testaus on ollut vapaamuotoisempaa ja jaksoille oli määritetty kourallinen testitapauksia, joita manuaalitestaaja on suorittanut viikoittain. Näiden testitapausten suorittamisesta on kirjattu ylös tilanteita, joissa on todettu virheitä.

Kestävyystestausta on suoritettu ohjelmistolle osittain manuaalisesti ja osittain automaation avustamana osana suoritettavaa kestotestijaksoa. Ohjelmiston resurssinkäyttöä on seurattu mm. JVisualVM -ohjelmiston avulla, josta testaajan on mahdollista tarkistaa esimerkiksi prosessorille aiheutuvaa kuormaa ja varatun kekomuistin määrää (“Oracle Java Documentation - JVisualVM” 2014). Muistinkäyttöä seuraamalla testaajan on ollut mahdollista tunnistaa esimerkiksi muistivuotoja jo testijakson aikana. Automaatiota on yleisesti ottaen käytetty kestotestijakson aikana syöttämään ohjelmistolle nauhoitettua tilannekuvaa, jotta ohjelmis-

| | Kestotestijakso 1 | Kestotestijakso 2 |
|---|---|---|
| Suoritetut testitapaukset per testauspäivä (ka.) | 17 | 24 |
| Yksittäisen testitapauksen suoritusmäärä per testausjakso (ka.) | 4 | 9 |
| Raportoidut virheet | 6 | 10 |
| Testitapausten kokonaismäärä | 52 | 49 |

Taulukko 1. Edeltävien kestotestijaksojen tilastot

ton kuormitus vastaisi lähemmin normaalia käyttötilannetta. Resurssien käytön seuraamisen lisäksi kestotestijakso on kattanut runsaasti ohjelmiston toiminnallisuutta käsitteleviä testitapauksia.

Kestotestijakson aikana suoritettavien testitapausten määrä vaihtelee kestotestijaksojen välillä. Testitapaukset on pyritty jakamaan siten, että ne kattaisivat laajalti loppukäyttäjän yleisesti käyttämiä ominaisuuksia sekä ennakkoarvion mukaan resurssienkäytön osalta raskaita ominaisuuksia. Kestotestijakson testitapaukset voidaan jakaa useisiin eri testaustyyppeihin. Osa testitapauksista liittyy toipumistestaukseen, jossa instanssien välille luodaan yhteyskatkoja. Toipumistestaus perustuu yleensä suunnitelmaan siitä, millä tavoin järjestelmän tulee kyetä palautumaan järjestelmän kohtaamasta ongelmasta (ISO/IEC 29119-1 2022b). Yhteyskatkoja luodaan katkaisemalla fyysisesti tietoverkkoyhteys ohjelmistoinstansseja ajavien järjestelmien välillä. Tämän jälkeen yhteys instanssien välillä palautetaan normaaliksi ja tarkkaillaan hajautetun ympäristön palautumista tällaisesta vakavasta häiriöstä. Tilanteissa, joissa testitapauksen avulla tarkkaillaan ohjelmiston resurssienkäyttöä, voidaan puhua kestävyystestauksesta. Näissä testitapauksissa manuaalisen testauksen suorittaja on antanut käskyjä ohjelmiston käyttöliittymän kautta tarkkaillen samalla resurssienkäyttöä. Ohjelmistoon voidaan kestotestijakson aikana syöttää ulkopuolisista järjestelmistä nauhoitettua dataa, jota ohjelmiston käyttöliittymä prosessoi käyttäjän tulkittavaksi. Tämä nauhoitettu data koostuu nimensä mukaisesti vanhasta tietyllä ajanjaksolla toteutuneesta tilannekuvasta, joka on tallennettu myöhempää käyttöä varten. Tämän lisäksi ohjelmistoon voidaan syöttää erikseen tietyillä määreillä generoitua dataa, jolloin datan määrää voidaan tarkemmin hallita esimerkiksi puolittamalla tai tuplaamalla syötteiden määrä halutulla aikajaksolla. Käyttäjän

tekemät toiminnot on kuitenkin suoritettu testaajien toimesta käyttöliittymään manuaalisesti syöttäen.

Kestotestin lähtötilanteessa käytetyt testitapaukset voidaan lajitella esimerkiksi testaustyypin mukaan. Testaustyyppeinä voidaan tunnistaa toiminnallisuustestaus, toipumistestaus sekä yhteensopivuustestaus. Yhteensopivuustestauksen avulla testataan esimerkiksi ulkoisten järjestelmien kautta saatavan kommunikaatiotiedon käsittelyn toimintaa (ISO/IEC 29119-1 2022a). Käsiteltävän järjestelmän tapauksessa on mielekästä erottaa hajautuksen testaaminen omaksi kategoriakseen muusta toiminnallisuustestauksesta, vaikka molemmat testaustyypiltään ovat yhteneviä ja testaavat ohjelmiston toiminnallisuuksia. Kestävyystestauksen näkökulmasta ohjelmistolle aiheutuvaa kuormaa lisää sekä käyttäjän suorittamat toiminnot että hajautuksen ja muiden järjestelmien kautta tulevan tietoliikenteen aiheuttama kuorma. Esimerkkinä testitapausten jakautumisesta eri kategorioihin on taulukossa 2 esitetty taulukon 1 testitapaukset kestotestijakson 1 osalta.

| Kategoria | lukumäärä |
|---|---|
| Toiminnallisuus | 29 |
| Yhteensopivuus | 13 |
| Hajautus | 9 |
| Toipuminen | 1 |
| Testitapausten kokonaismäärä | 52 |

Taulukko 2. Kestotestijakson 1 testitapausten jakautuminen

Kestotestijakson aikana suoritettava manuaalinen testaus on hyvin toisteista työtä. Yksinkertaisempien testitapausten suorittaminen vie aikaa, jota voitaisiin käyttää monimutkaisempien vaikeammin automatisoitavien testitapausten suorittamiseen. Automaation avulla testaamiseen käytettäviä resursseja halutaan allokoida mielekkäämmin. Ohjelmistoprojekti on pitkäkestoinen ja siitä luovutetaan loppukäyttäjälle uusi versio kerran vuodessa. Kestotestijaksoja on lähtötilanteessa suoritettu kaksi kertaa vuodessa. Tähän toivotaan lisäystä ja automaattisten testien avulla kestotestijaksoja voitaisiin mahdollisesti suorittaa myös useita rinnakkain.

# 5 Testausprosessin kehittäminen

Testausprosessin kehittämiseksi tuli tutustua ohjelmiston nykyisiin testausmenetelmiin ja yleisesti ottaen testauksella tavoiteltaviin seikkoihin. Aiemmin käytössä olleiden testitapausten lisäksi selvitettiin lähtötilanteessa resurssinkäytön seurannassa käytettyjä työkaluja ja kenties jo olemassa olevia automaatioelementtejä, joita voitaisiin hyödyntää jatkossakin.

Lähtötilanteen arvioinnin perusteella suoritettiin testattavien osioiden valintaa ja pyrittiin muodostamaan näitä silmällä pitäen testauksen tavoitteet täyttäviä testitapauksia. Testitapauksista muodostettiin lopullinen testiautomaatiototeutus.

Alaluvussa 5.1 käydään läpi kestotestausprosessin lähtötilanteen ongelmia. Alaluvussa 5.2 kuvataan ohjelmiston testattavien osioiden valintaa. Alaluvussa 5.3 kuvataan testitapausten avulla mitattavia seikkoja ja kuvaillaan muodostettavia testitapauksia. Alaluku 5.4 keskittyy kuvaamaan testiautomaation luomista käytössä olevien työkalujen avulla. Alaluvussa myös kuvataan tarkemmin, millaista tietoa ohjelmiston toiminnasta tallennetaan testiautomaation suorituksen aikana.

## 5.1 Lähtötilanteen ongelmien kartoitus

Lähtötilanteessa käyttäjän suorittamia toimintoja syötettiin ainoastaan testaajien toimesta manuaalisesti. Tätä ei käytettävissä olevien resurssien puitteissa voitu tehdä vuorokauden ympäri koko testijakson kattamaa aikaa. Jo käyttäjän toimintaa kuvaavan jatkuvan kuormituksen syöttäminen käyttöliittymään vaatii runsaasti henkilöresursseja, minkä lisäksi testaajan pitäisi pystyä tarkkailemaan ohjelmistoa ongelmatilanteiden varalta. Kuukauden mittainen kestotestijakso vaati lähtötilanteessa suuren määrän henkilötyötunteja, jotta ohjelmiston toimintaa voitiin manuaalisen testauksen avulla tutkia kattavasti.

Ohjelmistossa on jo tällä hetkellä tuhansia erilaisia yksittäisiä toimintoja ja niitä kehitetään jatkuvasti lisää. Toiminnot vaihtelevat erilaisista karttakäyttöliittymäoperaatioista esimerkiksi hajautuvan tiedon luomiseen ja muuttamiseen. Toimintojen lisääntyessä myös niiden manuaaliseen testaamiseen käytettävä aika kasvaa. Kestotestijakson aikana samoja tes-

titapauksia toistetaan useita kertoja, joten tällaisten testitapausten automatisoiminen on perusteltua. Kestotestijakson testitapausten suorituskerrat on merkitty lähtötilanteessa manuaalisesti ylös, mutta automaattisten testien suorituksesta on mahdollista muodostaa tarkka lokitieto automaattisesti.

Ohjelmistossa mahdollisesti tapahtuvia muistivuotoja halutaan selvittää pitkäkestoisen kestotestijakson aikana altistettaessa ohjelmistoa arvioidulle normaalille käyttökuormalle. Manuaalisesti tehtävän testauksen aikana testaaja voi havaita poikkeavuuksia ohjelmiston muistinkäytössä ja raportoida näistä. Resurssienkäytön tarkka seuranta oli kuitenkin hankalaa suoritettaessa testitapauksia manuaalisesti ja poikkeamahuomioiden sitominen tiettyihin testitapauksiin on voinut jäädä puutteelliseksi.

Manuaalisesti testitapauksia suorittaa useita testaajia, joten suoritustavoissa voi tästä johtuen olla käyttäjäkohtaista hajontaa. Automaatiotestit toistetaan eri testijaksoilla lähtökohtaisesti identtisellä tavalla, mahdollistaen eri testikertojen tulosten paremman keskinäisen vertailuarvon.

## 5.2 Testattavien osioiden valinta

Ohjelmistolle manuaalisesti suoritettujen testauskertojen perusteella saatua informaatiota voidaan käyttää hyväksi automaation avulla testattavien osioiden valinnassa. Aiemmilla testauskerroilla huomatut taulukossa 1 mainitut virheet ovat olleet muun muassa hajautukseen liittyviä, ulkoasuun liittyviä eli kosmeettisia tai muistinkäyttöön liittyviä. Koska näillä manuaalisesti tehdyillä testauskerroilla ohjelmistosta on kyetty löytämään virheitä, oli syytä pyrkiä käyttämään näitä testitapauksia yhtenä lähtökohtana testattavien osioiden valinnassa.

Yhtenä työkaluna testattavien osioiden valinnassa käytettiin käyttäjien asetustiedostoja. Ohjelmiston käyttäjä voi tietyin osin valita käyttöliittymässä tälle näytettävät painikkeet ja lisäksi asetella painikkeita haluamiinsa sijainteihin. Tämän lisäksi käyttäjä voi säätää avattavien näkymien asemointia ja kokoa. Nämä tiedot käyttäjän on mahdollista tallentaa omaksi asetustiedostoksi, johon tallennetaan myös tieto avoinna olleista näkymistä. Käyttäjä voi luoda useita erilaisia käyttäjätiedostoja esimerkiksi erilaisille käyttäjärooleille. Täten avattaessa esimerkiksi uusi käyttöliittymäistunto käyttäjän on mahdollista palauttaa näkymä siihen

tilaan, mikä asetuksiin on tallennettu. Näistä tiedostoista on mahdollista saada tietoa siitä, miten loppukäyttäjä ohjelmistoa käyttää. Mielenkiinto kohdistui testitapausten valinnassa erityisesti näkymiin ja näkymien asettelun sijaan niiden määrään ja tyyppiin.

Käyttäjäasetustiedostoja saatiin käsiteltäväksi järjestelmän oletusasetusten lisäksi yhdeksän erilaista erityyppisille käyttäjille määritettyä kokonaisuutta. Tietyn tyyppiset näkymät toistuivat kaikissa tai lähes kaikissa tapauksissa, mutta asetuksissa oli nähtävissä paljon vaihtelua. Asetuksista voitiinkin tunnistaa yhteensä 35 erilaista näkymää, joita oli määritetty avattavaksi eri käyttäjärooleilla. Näistä 15 kappaletta oli sellaisia, joita tavattiin vain yksittäisessä asetustiedostossa. Yksittäisellä käyttäjällä avattavia näkymiä oli asetuksissa määritetty keskiarvolta yhdeksän kappaletta. Enimmillään tietyllä käyttäjäroolilla näkymiä oli määritetty avattavaksi 19 erilaista. Tässä tulee huomioida, että käyttäjä hyvin todennäköisesti tämän lisäksi hyödyntää myös muita näkymiä, vaikkei tämän tarkistamiseksi varsinaista lokitietoa ollut käytettävissä. Myöskään tietoa siitä, kuinka kauan tiettyjä näkymiä käyttäjät normaalisti pitävät avattuina, ei ollut saatavilla. Voidaan kuitenkin olettaa, että ne pidetään lähes aina auki sovelluksen ollessa käynnissä. Tiettyjä näkymiä käyttäjät voivat myös avata useita kappaleita samanaikaisesti.

Asetustiedostojen perusteella testattavien osioiden valinnassa huomioitiin erityisesti suosituimmaksi osoittautuneet näkymät. Näiden lisäksi avoinna olevien näkymien määrä ja tietyistä näkymistä useiden ilmentymien samanaikainen käyttö ohjasivat testitapauksiin luodun kuorman suunnittelua.

Koska ohjelmisto on kooltaan suuri, oli testattavien osioiden valinta hankalaa. Testattavien osioiden valinnassa käytettiin siis soveltuvin osin manuaalisesti toteutettujen kestotestijaksojen testitapauksia. Koska näitä testitapauksia oli suuri määrä ja ne koskettivat laajalti ohjelmiston eri ominaisuuksia, pyrittiin niistä valitsemaan kestävyystestaukseen sopivimpia. Käytännössä testitapauksista pyrittiin valitsemaan arvion mukaan resursseja kuluttavia. Tämän lisäksi valittiin lähtökohtaisesti helpommin automatisoitavissa olevia, kuten esimerkiksi vähemmän erilaisia vaiheita sisältäviä testitapauksia. Ensisijaisesti pyrittiin valitsemaan toimintoja, joihin ei ollut tulossa lähiaikoina muutoksia. Koska kestävyystestausprosessissa haluttiin testata kahta hajautuksessa olevaa järjestelmää, tuli testitapauksiin valita myös hajautukseen liittyvä toimintoja. Näin järjestelmästä voitaisiin tarkkailla myös hajautuksen

kautta aiheutuvan kuormituksen vaikutuksia.

## 5.3 Testitapausten muodostaminen

Automaattista testausta varten luotavien testitapausten muodostamisessa on useita pääkohtia. Testitapauksia haluttiin voida suorittaa ajastetusti ilman jatkuvaa testaajan vuorovaikutusta ja mahdollisimman vähäisellä määrällä manuaalisesti suoritettuja toimintoja. Testiympäristöiksi valitut työasemat sijaitsevat kehittäjien ja testaajien pääasiallisista työtiloista erillään, joten testitapausten suorittamisen käynnistäminen etänä tuli mahdollistaa. Testijaksojen aikana tulisi kerätä tietoa järjestelmän resurssienkäytöstä. Myös mahdolliset ohjelmiston antamat virheilmoitukset testausjakson ajalta tulisi saada kerättyä talteen jatkotarkastelua varten. Sekä resurssienkäytöstä saatava tieto, että virheilmoitusten kerääminen toteutettaisiin automatisoidusti testitapausten lomassa. Koska kyseessä on kestävyystestausjakso, ei testattavaa ohjelmistoa tule sammuttaa testausjakson aikana. Testausprosessi tulisi pyrkiä ylipäätään automatisoimaan siten, että testaaja joutuisi testausprosessin aikana suorittamaan mahdollisimman vähän manuaalisia toimintoja.

Muodostettaessa testitapauksia aiempien manuaalisten kestotestijaksojen pohjalta tuli huomioida manuaalisten testien ero automaattisten testien luonteeseen. Manuaalisella testausjaksolla testaaja tarkkailee jatkuvasti poikkeamia toiminnallisuuksissa ja järjestelmässä mahdollisesti tapahtuvia odottamattomia tapahtumia. Manuaalinen testaaja on myös voinut koska tahansa muuttaa toimintojen suoritusjärjestystä tai nopeutta seuraten tällaisten tekojen vaikutuksia saataviin tuloksiin. Automaattisissa testitapauksissa toimintojen suoritusjärjestys tuli päättää ennakkoon muun muassa siksi, että voitaisiin olla varmoja järjestelmän sen hetkisestä tilasta. Esimerkiksi tulee varmistaa, että sellainen näkymä on jo avattu, jonka kanssa vuorovaikutetaan. Tietyt näkymät avautuvat telakoituna käyttöliittymään, kun taas toiset avautuvat oletuksena niin sanotusti kelluviksi ikkunoiksi. Myös tällä on vaikutusta siihen, miten ikkunoiden kanssa automaattitesteissä tulee vuorovaikuttaa.

Koska aiemmin suoritetuissa testitapauksissa painottui toiminnallisuuden testaaminen, tulisi kestävyystestausjakson testitapauksista luonteeltaan erilaisia. Automaattisissa testitapauksissa toiminnallisuuden testaamisen sijaan keskityttäisiin kuormittamaan järjestelmää. Auto-

maattisten testien avulla esimerkiksi kosmeettisten poikkeamien seuranta olisi erittäin työlästä toteuttaa. Toimintojen oikeellisuuden testaamisen sijaan painotus keskittyikin järjestelmän kuormittamiseen ja toimintojen kuormittavuuden seuraamiseen.

Ohjelmistoa testattaessa ja kuormitettaessa on huomattu joidenkin osioiden olevan ohjelmistolle enemmän kuormittavia. Ohjelmistossa on tällä hetkellä yli sata erilaista käyttäjän avattavissa olevaa näkymää. Näkymien avulla käyttäjä voi tarkastella ja usein myös syöttää ohjelmistoon tietoa. Hajautuksen avulla käyttäjä voi jakaa näkymissä käsiteltäviä tietoja muille käyttäjille. Käyttäjällä voi olla yhdenaikaisesti avoinna useita näkymiä kuormittamassa järjestelmää. Koska tällainen toiminta on oletusarvoisesti usein toistuvaa käyttäjän jatkuvasti suorittamaa perustoimintaa, sen sisällyttäminen automaattisiin testeihin on perusteltua.

Yksi käyttäjän avattavissa oleva näkymätyyppi on apukartta, joka lisää käyttöliittymään uuden karttanäkymän. Käyttäjä tarkastelee karttanäkymää useilta eri alueilta tai laajemman karttanäkymän lisäksi useaa tarkempaa osiota kartasta. Käyttäjän on mahdollista avata tähän tarkoitukseen useita apukarttanäkymiä, jotka kuvastavat tiettyjä tarkempia osa-alueita kartasta. Näiden apukarttojen on huomattu lisäävän kuormitusta ja niiden käyttäjälle yhdenaikaisesti avoinna oleva määrä on tästä syystä rajattu. Yhtenä kuormitusta lisäävänä testattavana osiona testitapauksiin valittiin apukarttojen käyttäminen.

Apukarttojen käyttämisen lisäksi on syytä tarkastella myös muiden näkymien avaamisen ja niiden kanssa vuorovaikuttamisen vaikutusta resurssinkäyttöön. Tätä tukee muun muassa jo aiemmin ohjelmistoa testattaessa tehty havainto Java-ohjelmoinnissa käytettävien kuuntelijoiden käytöstä johtuneista sittemmin korjatuista muistiongelmista. Yleisesti ottaen kuuntelijoita käytetään tarkkailemaan käyttäjän toimintoja muun muassa avatuissa dialogi-ikkunoissa (Darwin 2001). Rekisteröityjen kuuntelijoiden vaillinainen käsittely onkin nähty yhtenä yleisimmistä muistiongelmien aiheuttajista (Bloch 2017). Testitapauksiin valittiin käsiteltäväksi useita erilaisia näkymiä ja niissä suoritettavia toimintoja. Näkymien käsittelyssä haluttiin käyttää hyödyksi testausjakson pituutta siten, että näkymien sulkemista pyrittäisiin välttämään, jotta voitaisiin tarkkailla ajan mittaan kertyviä muistivuotoja.

Järjestelmä jakaa hajautuksen avulla muille käyttäjille myös esimerkiksi käyttöliittymässä esitettävään pääkarttanäkymään tehtäviä merkintöjä. Tämän lisäksi käyttäjälle näkyvillä ole-

vaa karttaosuutta on käyttäjän mahdollista tarkentaa ja liikutella eri suuntiin. Kartalla tehtävien toimintojen sisällyttäminen testitapauksiin nähtiin perusteltuna, vaikkakin automaation avulla kartan käsittely olisi ennalta arvioituna hankalaa. Myös hajautuksen toiminta tuli tarkistaa, koska hajautuksen kautta saatava tietoliikenne aiheuttaa omalta osaltaan kuormitusta. Hajautuksen kautta saapuvan liikenteen puuttuminen voisi laskea järjestelmään aiheutuvaa kuormitusta ja näin vääristää tuloksia järjestelmän normaaliin käyttöön verrattaessa.

Testitapauksiin pyrittiin sisällyttämään seikkoja, joita manuaalisella testauksella on lähes mahdotonta toteuttaa. Testitapauksiin lisättiin toisteisuutta siten, että eri toimintoja, kuten näkymien avaamisia, suoritettaisiin useita kertoja peräkkäin. Näin saataisiin aikaan volyymia, joka itsessään lisää järjestelmän kuormitusta. Lisäksi järjestelmään aiheutettiin taustakuormaa mallintamaan ulkoisista järjestelmistä tulevaa tietoliikennettä. Taustakuorma pyrittiin luomaan mahdollisimman tasaiseksi, jottei se aiheuttaisi piikkejä resurssien käytössä eri puolille testijaksoa. Osa järjestelmään syötettävistä merkinnöistä, kuten karttaan tehtäviä kuvioita, generoitiin järjestelmän ohjelmakoodissa sen sijaan, että ne olisi piirretty käyttöliittymän työkaluilla. Joidenkin toimintojen osalta käytettiin molempia tapoja tiedon syöttämiseksi järjestelmään.

## 5.4 Testiautomaation muodostaminen

Ohjelmiston kehitystyön osia on aiemmin jo automatisoitu muun muassa Jenkinsin avulla, joka on avoimeen lähdekoodiin perustuva automaatiotyökalu (“Jenkins User Documentation” 2011). Jenkins mahdollistaa erilaisten CI-putkien (engl. continuous integration) luomisen. CI-putki kuvaillaan Jenkinsfilen avulla, jonka sisältö muodostetaan Groovy-ohjelmointikielen syntaksilla (“Jenkins User Documentation” 2011). Näin muodostettavan CI-putken avulla mahdollistetaan se, että useiden kehittäjien tekemät ohjelmistomuutokset ja näihin liittyvät testitapausten päivitykset saadaan koottua helposti yhteen. Tällöin esimerkiksi uusia versioita ohjelmistosta voidaan muodostaa ja testata virtaviivaisesti heti muutosten jälkeen. Jenkinsin avulla on jo aiemmin toteutettu ominaisuuksia, joiden avulla ohjelmakoodista voidaan muodostaa tiedostot ohjelmiston asentamiseksi.

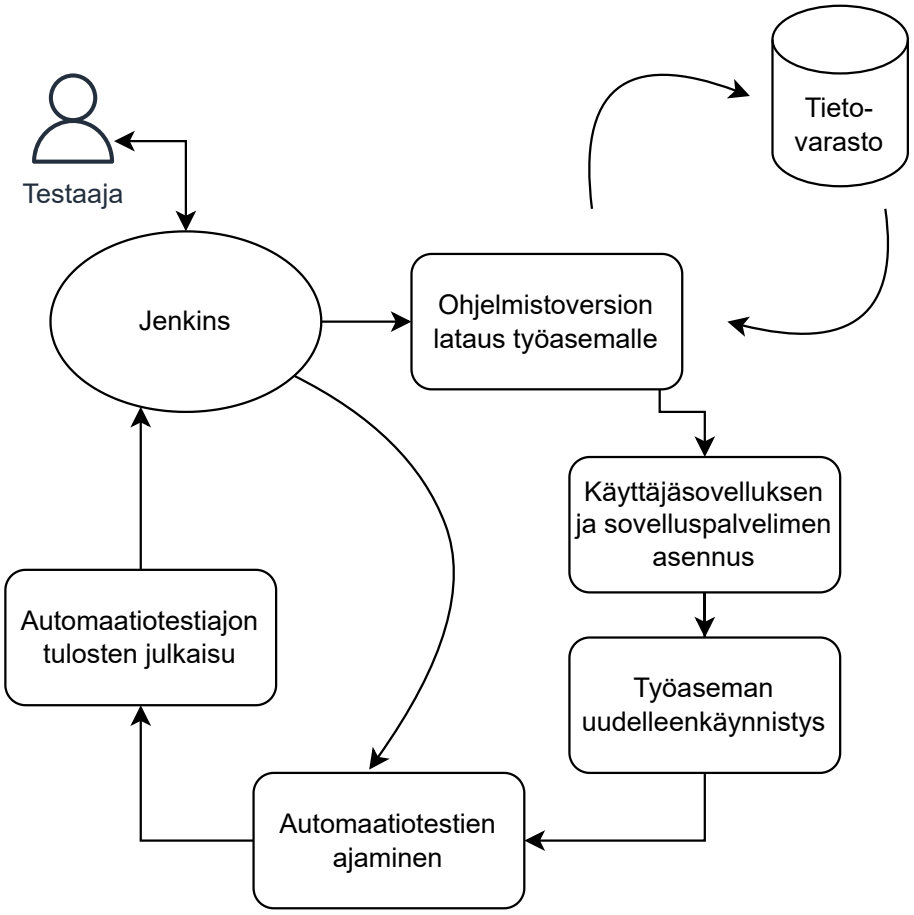
Jenkinsfile koostuu ’stage’ -osioista, joilla kuvataan CI-putken eri vaiheita (“Jenkins User

Documentation" 2011). Nämä vaiheet suoritetaan oletuksena lineaarisesti yksi kerrallaan, mutta ne on mahdollista myös määrittää soveltuvin osin suoritettavaksi rinnakkain. Testiautomaatiota varten muodostettiin kuviossa 3 yksinkertaistettuna nähtävä CI-putki, joka koostui yleisellä tasolla neljästä vaiheesta. Ensimmäisessä vaiheessa määritetylle työasemalle ladattaisiin valittuun ohjelmistoversioon liittyvät asennustiedostot niille varatusta tietovarastosta. Toisessa vaiheessa työasemalle ohjelmistosta asennettaisiin käyttäjäsovellus ja palvelin, kolmannessa vaiheessa työasema käynnistettäisiin uudelleen ja viimeisessä vaiheessa suoritettaisiin varsinaiset automaatiotestit. Näiden vaiheiden jälkeen Jenkins julkaisisi automaatiotestiajon tulokset tarkasteltavaksi. Koska automaatiotestit haluttiin suorittaa kahdelle keskenään hajautuksessa olevalle työasemalle, määritettiin molemmille työasemille edellä kuvatut vaiheet suoritettavaksi järjestyksessä rinnakkain. Erityisesti testiautomaation suorittaminen tuli tapahtua rinnakkain, jotta molemmilla työasemilla ohjelmistoja kuormitettaisiin yhdenaikaisesti.

Jenkinsissä muodostettavan CI-putken vaiheita pyrittiin parametrisoimaan siten, että testaaja voi valita ajossa suoritettavat vaiheet. Täten esimerkiksi työasemalle, jolle on jo aiemmin asennettu testattava versio ohjelmistosta, ei olisi välttämätöntä ladata ja asentaa ohjelmistoa uudelleen. Myös automaatiotestiajon ajastus toteutettiin Jenkinsin avulla siten, että ensimmäisen manuaalisesti käynnistetyn ajon jälkeen testejä toistettiin jatkuvana sarjana automaattisesti tunnin välein. Ensimmäisellä ajolla suoritettiin kaikki edellä kuvatut vaiheet kerran ja ajastuksen avulla suoritettiin vain varsinaiset automaatiotestit suorittava vaihe. Testijakson lopetusajo käynnistettiin manuaalisesti haluttuna ajankohtana. Tällöin automaation avulla ohjelmisto ja siihen kuormaa syöttävät taustajärjestelmiä mallintavat työkalut suljettiin hallitusti ja viimeiset resurssinkäyttötiedot tallennettiin järjestelmään.

Kestotestijaksoilla on aiemmin käytetty kerrallaan yhtä versiota ohjelmistosta. Automaation avulla haluttiin mahdollistaa, että ohjelmistosta voidaan valita tarvittaessa useampi versio, joita testataan samanaikaisesti. Ohjelmistoa kehitetään jatkuvasti ja täten uusimmat korjaukset sisältävä ohjelmistoversio saataisiin kestotesteihin nopeammin. Jenkinsfile:ssä kuvatuissa asetuksissa määritettiin testausjaksoa aloitettaessa asennettavaksi halutuille työasemille käyttäjän syöttämä järjestelmäversio.

Automaatiotestejä varten allokoitiin neljä fyysistä työasemaa, joilla testejä suoritettaisiin.




Kuvio 3. CI-putki

Jokaisella työasemalla suoritettaisiin yhtä käyttäjäsovellusta Windows-ympäristössä ja yhtä palvelinta virtualisoidussa Linux-ympäristössä. Yksittäisten työasemien konfiguraatio siis noudattaa kuviossa 2 nähtävää. Tässä tapauksessa työasemat konfiguroitiin pareiksi siten, että kahden työaseman sovelluspalvelimet toimivat hajautuksessa keskenään ja yhdellä parilla on käytössä keskenään sama versio testattavasta järjestelmästä. Täten tällaisella kokoonpanolla voitaisiin yhdenaikaisesti testata kahta eri versiota järjestelmästä. Parien työasemista toinen määritettiin päätyöasemaksi, jolla suoritettaisiin valtaosa testitapauksista. Hajautuksessa olevalle työasemalle määritettiin suoritettavaksi omat erilliset testitapaukset.

Itse automaattisten testien muodostaminen aloitettiin testitapausten muodostamisen lomassa. Tämä johtui siitä, että oli epävarmaa millaisia testitapauksia valituilla automatisointityökaluilla saataisiin toteutettua kohtuullisella työmäärällä. Testiautomaatiota kehitettäessä saatiin

jatkuvasti tietoa automaatiotyökalujen mahdollisuuksista. Robot Frameworkia hyödynnettiin luotaessa automaattisia testejä (jäljempänä robottitestejä). Ohjelmistoa pyrittiin kuormittamaan käskyttämällä sitä eri tavoin käyttöliittymän kautta näitä robottitestejä käyttäen.

Kestävyystestauksessa ohjelmistoa testataan pienempien osioiden sijaan kokonaisuutena, kuten loppukäyttäjä sitä tulisi käyttämään. Kuten on jo todettu, kuormituksen tuli vastata normaalia käyttötilannetta. Automatisoiduissa testitapauksissa kuormitus ohjelmistolle pyrittiin toteuttamaan sellaisena, että se vastaisi hieman suurempaa kuormaa kuin normaalissa käyttötilanteessa. Tällöin mahdolliset suorituskykyongelmat tulisivat esille aiemmin, kuin pienemmällä kuormalla käytettäessä. Testiautomaatiota muodostettaessa loppukäyttäjän aiheuttaman normaalin kuormituksen määrä jouduttiin kuitenkin arvioimaan, sillä varsinaista nauhoitetta tai muuta ennakkoon tallennettua tietoa ohjelmiston eri toiminnallisuuksien käytöstä ei ole kerätty. Ohjelmiston nykytilassa tällaisen käyttäjätiedon keräämistä ei ole suunniteltu.

Robot Frameworkille on tehty useita kirjastoja yksinkertaistamaan robottitestien muodostamista. Nämä kirjastot sisältävät valmiita avainsanoja käytettäväksi testitapausten muodostamisessa. Yksi näistä kirjastoista on RemoteSwingLibrary, jonka tarkoitus on helpottaa erityisesti Java-käyttöliittymäsovellusten testaamista (“RemoteSwingLibrary” 2014). Tämän kirjaston avulla mahdollistettiin testauksen alaisen Java-ohjelmiston toimintojen käskyttäminen suoraan ohjelmakoodin tasolla. RemoteSwingLibrary mahdollisti myös testien suorittamisen jo aiemmin käynnistetylle ohjelmistolle, joka oli erityisen tärkeää suoritettaessa robottitestejä ajastuksen avulla toistuvasti.

Testitapaukset merkittiin ”tagien” avulla, joka mahdollistaa Robot Frameworkissa testitapausten ryhmittelyn (“Robot Framework Documentation” 2021). Tämän ryhmittelymekanismin avulla testitapauksia on mahdollista jättää tietyn testiajon ulkopuolelle tai vaihtoehtoisesti sisällyttää vain tiettyyn ryhmään kuuluvat testitapaukset testiajoon. Tätä ryhmittelyä käyttäen testitapaukset jaettiin aloittaviin, lopettaviin ja jatkuvasti suoritettaviin robottitesteihin.

Aloittavien testitapausten tarkoitus oli käynnistää testattava ohjelmisto ja kuormitusta tuottavat ulkoiset järjestelmät. Lisäksi niissä käynnistettiin muun muassa kuormitusta käyttäjäsovellukseen syöttävät ulkoiset järjestelmät. Näistä tarkoitus on syöttää nauhoitettua tilanne-

kuvaa, kuten on tehty myös manuaalisesti suoritetuilla kestotestijaksoilla.

Jatkuvasti suoritettavat testitapaukset sisälsivät pääosan käyttöliittymän kautta kuormitusta aiheuttavista testitapauksista. Testitapauksissa oli hajautusta testaavia testitapauksia sekä vain yksittäistä toimipaikkaa kuormittavia testitapauksia. Hajautusta sisältävissä testitapauksissa luotiin käyttöliittymässä muutamia kohteita, joille annettiin yksilöivät tunnistenumerot. Nämä tunnistenumerot oli tallennettu erilliseen resurssitiedostoon, joka oli saatavilla molemmilla hajautuksessa olevilla toimipaikoilla. Hajautuksessa olevalla toimipaikalla tarkistettiin, näkyykö käyttöliittymässä kohteita, joille on annettu nämä tunnistenumerot. Lisäksi luotiin runsaasti kohteita, joiden olemassaoloa ei erikseen tarkistettu hajautuksessa olevilla toimipaikoilla, mutta jotka tiedon hajautuessa luovat rasitusta molemmille toimipaikoille.

Käyttöliittymää kuormittavaksi tarkoitetuissa testitapauksissa avattiin ensimmäisessä testiajossa useita erilaisia näkymiä ja lisäksi useisiin niistä luotiin tietosisältöä eri tavoin. Yksi tavoista syöttää tietoa oli käydä läpi käyttöliittymässä käyttäjälle esitettävää listaa karttanäkymän kohteista. Tästä listasta testitapausten aikana avattiin peräkkäin useita satoja kohteita Robot Frameworkin ja RemoteSwingLibraryn avulla. Kohteen tietojen avaus toisi aktiiviseksi kohteeseen liittyvän näkymän päivittäen kohteen tiedot näkymään. Kohteita olisi listassa muun muassa ulkoisten järjestelmien luoman kuorman ansiosta useita erityyppisiä, joten myös avattavissa näkymissä olisi mukana tyypiltään erilaisia. Tämä kuvastaisi käyttäjän suorittamaa toimintaa ja kuormittaisi käyttöliittymää tyhjien näkymien avaamista tehokkaammin. Näin testattaisiin useisiin näkymiin liittyvien kuuntelijoiden mahdollisesta vaillinaisesta käsittelystä johtuvia ongelmia.

Toistettavilla testiajoilla näkymien tietoja päivitettäisiin, mutta kestotestijakson aikana näkymät suljettaisiin vasta koko jakson lopussa. Näin mahdollisten muistivuotojen vaikutukset näkyisivät testijaksolla todennäköisemmin.

Lisäksi testattavaa järjestelmää käskytettiin ohjelmallisesti esimerkiksi piirtämään karttanäkymään kohteita, joiden tiedot samalla päivittyisivät piirroksia listaavassa näkymässä. Myös tiedon syöttämistä Robot Frameworkin avulla suoraan näkymiin hiiritoimintoja käyttäen suunniteltiin, mutta se nähtiin automaatiotestejä muodostettaessa epäluotettavaksi.

Lopettavat robottitestit sulkivat itse testattavan järjestelmän ja ulkoiset kuormitusta luovat

järjestelmät hallitusti. Tämän lisäksi loput robottitestit huolehtivat viimeisten resurssimittausten tallentamisesta ja tuloksia sisältävien tiedostojen keräämisestä tarkasteltavaksi ennalta päätettyyn tiedostosijaintiin.

Lisäksi ryhmittelyn avulla määriteltiin suoritetaanko robottitesti niin kutsutulla päätyöasemalla vai sen kanssa hajautukseen määritetyllä työasemalla. Testiajolla suoritettavat testitapaukset määriteltiin Jenkinsissä näiden ryhmien avulla. Ryhmittelyn käyttäminen helpotti myös testitapausten kehitystyötä, koska kerralla suoritettavia testejä oli näin helppo rajata myös kehitysympäristössä.

Robot Framework muodostaa jokaisesta suoritetusta testiajosta lokin, jossa on eriteltynä eri robottitestien ja niiden vaiheiden onnistuminen tai epäonnistuminen (“Robot Framework Documentation” 2021). Käytännössä testitapauksissa testin epäonnistuminen tarkoitti sitä, ettei esimerkiksi valittua näkymää saatu avattua tai valitun painikkeen kanssa vuorovaikuttaminen epäonnistui. Epäonnistuneista testitapauksista määritettiin lokitietojen mukana tallennettavaksi näyttökaappaus testauksen alaisesta ohjelmistosta. Lisäksi määritettiin testitapaukset, jotka tarkistivat käyttäjäsovelluksen ja sovelluspalvelimen virhelokin. Näissä lokeissa löytyvät virheet aiheuttaisivat tämän testitapauksen epäonnistumisen. Jotta lokitietojen tarkastelu helpottuisi, käytettiin Jenkins-lisäosaa, jolla automaattisten testien lokitiedot kerätään esitettäväksi Jenkinsissä (“Jenkins Robot Plugin” 2011).

Testitapauksista muodostettavien lokitietojen lisäksi haluttiin seurata ohjelmiston resurssinkäyttöä. Tämän mahdollistamiseksi muodostettiin uusia Robot Frameworkin käyttämiä avainsanoja Java-ohjelmointikielellä. Testattavan Java-ohjelmiston resurssienkäytöstä käynnissä olevan prosessin osalta voitiin näin kerätä tietoa Javan Runtime-luokan metodeja hyväksikäyttäen. Metodien avulla pyrittiin kutsumaan roskienkeruuta, jonka jälkeen otettaisiin mittausarvo. Muodostetut avainsanat määritettiin siten, että robottitestin parametrina tulisi antaa sekunteina, kuinka usein arvoja ohjelmiston toiminnasta mitataan. Tämän perusteella kerättiin ja tallennettiin annetusta aikamääreestä riippuen muutama peräkkäinen mittausarvo. Resurssinkäytön mittaaminen aloitettiin ja keskeytettiin robottitestien kautta näiden avainsanojen avulla.

Arvoina kerättiin työaseman prosessorin kuormitus prosentteina, ohjelmiston käyttämän ke-

komuistin määrä, käytetyn kekomuistin määrän muutos prosentteina verrattuna koko testijakson alkuun sekä mittausajankohdan aikaleima. Tiedot tallennettiin ajonaikana työaseman kiintolevylle CSV-formaatissa. Tällöin resurssinkäytön tiedot tallentuisivat, vaikka ohjelmiston suoritus testijakson aikana pysähtyisi odottamattomasti. Näiden lisäksi sovelluksen muistinkäytöstä muodostettiin histogrammi *jmap*-komentorivityökalua käyttäen. Lisäksi robottitesteissä määritettiin muistivedoksen (heap dump) ottaminen *jmap*-työkalulla testattavaa sovellusta käynnistettäessä ja suljettaessa.

# 6 Testausprosessin arviointi

Tutkimuksessa kehitetylle artefaktille asetettiin tavoitteita, joiden ratkomiseksi suunniteltiin testitapauksia. Näiden testitapausten pohjalta muodostettiin varsinainen testiautomaatio, jolla pyrittiin mahdollistamaan kestävyystestaukseen nähden riittävä kuormitus ja lisäksi sovelluksen kuormitukselle seuranta.

Kehitettävälle kestotestiprosessille muodostettuja testitapauksia ja niiden kattavuutta arvioidaan alaluvussa 6.1. Alaluvussa 6.2 arvioidaan testitapausten pohjalta toteutettua testiautomaatiota ja testiautomaation toteutuksessa käytettyjen työkalujen tarkoituksenmukaisuutta. Tämän lisäksi alaluvussa 6.3 arvioidaan millä tavoin tutkielman tavoitteet saavutettiin.

## 6.1 Testitapausten arviointi

Muodostettavan artefaktin testitapauksia kehitettiin muun muassa aiempien kestotestijaksojen pohjalta. Koska testitapausten avulla pyrittiin löytämään esimerkiksi mahdollisia muistivuotoja, valittiin niihin osioita, joiden on nähty tyypillisesti aiheuttavan ongelmia Java-ohjelmistoissa. Vaillinainen kuuntelijoiden käsittely ohjelmistossa käytettävien näkymien yhteydessä nähtiin tällaisena kohteena ja erilaisia ohjelmistosta löytyviä näkymiä hyödynnetäänkin testitapauksissa runsaasti.

Näkymiin syötetään testitapauksissa tietoa erilaisten toimintojen avulla pyrkien matkimaan käyttäjän oletettua toimintaa. Näkymät on valittu painottaen pääosin käyttäjän useimmiten hyödyntämiä dialogi-ikkunoita. Tätä tukemaan testitapaukset on suunniteltu hyödyntäen loppukäyttäjien käyttämiä järjestelmäasetuksia, joissa on määritetty oletuksena avautuvia näkymiä. On kuitenkin hankala arvioida missä määrin suoritetut testitapaukset kuvaavat loppukäyttäjän päivittäistä toimintaa.

Osana järjestelmätestausta kestotestijaksolla on pyrkimys testata ohjelmistoa kattavasti eri osa-alueilta. Ohjelmistossa on paljon ominaisuuksia, jotka jätettiin kestotestijakson ulkopuolelle. Syynä tähän on joiltain osin toimintojen testauksen hankalasti toteutettavissa oleva automatisointi. Testitapauksissa käytetyt näkymät liittyvät kuitenkin laajalti erilaisiin käyttä-

jien tavallisesti hyödyntämiin toiminnallisuuksiin, sillä testitapauksissa avataan yhteensä 28 erilaista näkymää, kun niitä käyttäjien hyödyntämistä järjestelmäasetuksista voitiin tunnistaa 35 erilaista. Hyödynnettävien näkymien osalta testitapaukset ovat täten kattavia. Testattavaksi valitut osiot ovat myös sellaisia, ettei niihin kohdistu välittömiä paineita muutoksille.

Tietyissä testitapauksissa avataan näkymiä, joihin ei tuoteta tietosisältöä. Tällaisten testitapausten aiheuttama kuormitus jää pieneksi ja kyseiset testitapaukset tuovat nykymuodossaan vain vähän lisäarvoa testijaksolle. Tällaiset testitapaukset pikemminkin luovat mahdollisuuden jatkokehitykselle siten, että kaikkiin avattaviin näkymiin pyrittäisiin tuottamaan tietosisältöä. Eniten ongelmia nousi testitapauksissa, joissa käyttöliittymän kanssa pyrittiin vuorovaikuttamaan suorilla käyttöliittymäsyötteillä. Tällöin esimerkiksi näkymien sijainti saattoi erota kehitysympäristön ja testausympäristön välillä, johtaen testitapausten epäluotettavaan toimintaan. Tämän vuoksi tällaisia testitapauksia pyrittiin välttämään ja osa näin toteutetuista testeistä täytyi hylätä lopullisesti toteutuksesta.

Muodostettujen testitapausten aikana kyettiin seuraamaan resurssienkäyttöä, kuten muistinkäytön muutoksia ja prosessorikuormitusta. Testitapauksissa luotiin riittävästi kuormaa, jotta ohjelmiston resurssinkäytössä huomattaisiin vaihtelua. Testitapaukset on muodostettu siten, että ne ovat myös automaation avulla toistettavissa. Jokaisella testiajolla suoritettaisiin sama määrä toimintoja samassa järjestyksessä. Tällöin aiheutettava kuormitus pysyy yhdenmukaisena. Täten testitapausten perusteella suoritettavat automaatiotestijaksot ovat kuormituksen osalta toisiinsa nähden verrattavissa. Täten esimerkiksi testiajosta toiseen lisääntyvä muistinkäyttö on hyvin todennäköisesti osoitus muistivuodosta.

Testitapauksia käytettiin sekä kestävyys- että hajautustestaustarkoituksessa. Ohjelmiston hajautusominaisuuksia testataan testitapausten avulla hyvin niukasti ja pääpaino on kestävyystestaukseen liittyvillä testeillä.

Automaatiotestejä kehitettiin rinnakkain testitapauksia muodostettaessa. Kehityksen aikana myös testattavaan ohjelmistoon lisättiin ominaisuuksia ja olemassa olevia ominaisuuksia kehitettiin edelleen. Automaatiotestejä pyrittiin suorittamaan aina uusimmalla saatavilla olevalla ohjelmistoversiolla. Muodostettuja testitapauksia voitiin käyttää ohjelmiston eri versioilla ilman muutoksia.

## 6.2 Testiautomaatiototeutuksen arviointi

Muodostettuja automaatiotestejä koeajettiin niin sanotuilla harjoitustestijaksoilla, joissa testit suoritettiin niille valmistelluissa testausympäristöissä. Näin ollen voitiin saada tietoa muodostetun artefaktin toimivuudesta. Harjoitustestijaksojen lopullista kestoa tai määrää ei ollut määritetty ennalta, vaan tarkoitus oli tarkkailla, pystyykö automaatiotestejä suorittamaan yhtäjaksoisesti varsinaisella kestotestijaksolla. Harjoitustestijakso keskeytettäisiin mahdollisten ongelmien ilmentyessä. Tällaisia ongelmia olisivat esimerkiksi selkeä nousu resurssienkäytössä tai ohjelmiston toiminnassa havaittava muu virhe. Testitapaukset määritettiin aiheuttamaan varoitus yli 10 prosentin noususta muistinkäytössä ja yli 25 prosentin noususta prosessorikuormassa. Vertailu tapahtui testijakson alussa mitattuun nähden.

Ensimmäisellä harjoitustestijaksolla havaittiin ongelma muistinkäytössä, sillä käyttäjäsovelluksen muistinkäyttö nousi paikoin n. 70 prosenttia. Itse ohjelmistossa on olemassa toiminto, jolla voidaan luoda automaattisesti karttakäyttöliittymässä esitettäviä kohteita. Tätä ominaisuutta käytetään lähinnä testausjaksoilla, sillä toiminnon avulla luodaan satunnaisia arvoja sisältäviä kohteita. Syy muistinkäytön nousuun pystyttiin paikantamaan metodiin, jolla näitä kohteita karttakäyttöliittymään luodaan. Osa ilmentymistä luotiin virheellisillä arvoilla, jolloin niiden poistaminen epäonnistui aiheuttaen kertymistä muistiin. Tämä johti suoritusaikojen pidentymiseen ja yksittäisen testiajon suoritus pidentyikin noin viisinkertaiseksi alussa havaittuun nähden. Tämä pidentyminen tapahtui jo vuorokauden mittaisella testijaksolla. Toimintoa käyttävä avainsana poistettiin testeistä virheen korjaamisen ajaksi.

Toisella harjoitustestijaksolla huomattiin edelleen testijakson suoritusajan pidentymistä. Pidentyminen paikannettiin yksittäiseen testitapaukseen, jonka tarkoituksena oli avata käyttöliittymässä olevasta listanäkymästä erityyppisiin karttanäkymässä esitettäviin kohteisiin liittyviä näkymiä. Kohteita käytiin läpi satoja jokaisella testiajolla, joten suoritusajan pidentyminen oli helposti ja nopeasti havaittavissa. Syyksi paljastui muistivuoto yhdessä ohjelmiston käyttämässä metodissa, joka päivitti tietyn kohdetyypin tietoja kohteelle varatussa näkymässä.

Kolmannella harjoitustestijaksolla robottitesteihin palautettiin aiemmin käytöstä poistettu avainsana. Lisäksi poistettiin käytöstä joitakin testitapauksia, joissa avattiin suurikokoisia

näkymiä, joiden tiedettiin myös olevan vähemmän käytettyjä normaalissa toiminnassa. Näkymien suuren koon todettiin hankaloittavan vuorovaikuttamista karttakäyttöliittymän kanssa. Aiemmin kehityksen aikana ongelmaa ei havaittu, sillä kehitysympäristössä käytetty näytön resoluutio poikkesi testaukseen allokoitujen työasemien näyttöjen resoluutioista. Muistinkäytössä havaittiin edelleen suurenemaa, jonka perusteella löydettiin muistivedoksia tutkimalla kaksi muistivuotoista Java-luokkaa.

Neljännellä harjoitustestijaksolla käytetty ohjelmistoversio sisälsi korjauksia muistivuotoisiksi todettuihin Java-luokkiin ja lisäksi muutoksia tehtiin resurssimittausta tekevän avainsanan toteutukseen. Muutokset tehtiin, koska mitatuissa muistiarvoissa oli epätarkkuuksia ja yksittäisiä poikkeavia arvoja. Pystyttiin toteamaan, että tehty mittaus ei tapahtunut aina heti roskienkeruun jälkeen, joten kaikki mitatut arvot eivät olleet vertailukelpoisia. Avainsanaa muutettiin myös siten, että tehtyjen mittausten määrää vähennettiin. Mitattu arvo olisi aina viimeisimmän toteutuneen roskienkeruuhetken mukainen, jotta voitaisiin varmistua arvon oikeellisuudesta. Kyseisellä testausjaksolla havaittiin joidenkin testitapausten suoritusaikojen pidentymistä. Erityisesti satoja kohteita peräkkäin avaavan testitapauksen hidastuminen oli huomattavissa lokiin kirjautuneista suoritusajoista. Osasyyksi epäiltiin tapaa, jolla sovellus käsitteli auki olevan näkymän aktivoinnin. Näkymän sisältö generoitiin uudestaan, mutta vanhaa sisältöä ei siivottu yhtä tarkasti, kuin ikkunaa suljettaessa. Tämän korjaaminen vähensi hidastumista, mutta ei kokonaan poistanut sitä. Lopulta syyksi paljastui muistivuoto kolmannen osapuolen komponentissa. Havaittiin, että kolmannen osapuolen komponentista oli julkaistu päivitetty versio, jossa muistivuoto oli korjattu. Päivityksen jälkeen testitapauksen suorituksen hidastuminen loppui.

Suoritettujen harjoitustestijaksojen avulla saatiin arvokasta tietoa testiautomaation toiminnasta. Näiden avulla saatiin myös kehitettyä testiautomaatiototeutusta tavoitteisiin sopivammaksi siten, että huomioitiin erityisesti kestävyystestauksen näkökulmia pyrkien painottamaan testitapauksissa niiden kuormittavuutta. Tätä toteutettiin muun muassa testien kuormitusta säätämällä. Harjoitustestijaksojen perusteella myös osa testitapauksista robottitesteissä jaettiin pienempiin kokonaisuuksiin, jotta tietyn vaiheen epäonnistuminen ei johtaisi loppujen testitapauksen vaiheiden suorituksen ohittamiseen.

Testiautomaation toteutuksen aikana käytettiin työkaluina erilaisia avoimen lähdekoodin työ-

kaluja. Suurin osa artefaktin toteuttamiseen käytetyistä työkaluista oli valittu valmiiksi yrityksessä aiemman käyttökokemuksen perusteella. Täten työkalujen osalta ei varsinaisesti suoritettu vertailua niiden sopivuudesta. Jenkins tarjosi hyviä mahdollisuuksia CI-putken muodostamiseksi ja se mahdollisti putkelta vaaditut toiminnallisuudet. CI-putken toteutus mahdollistaa tässä vaiheessa testiajon kahdelle hajautuksessa olevalle toimipaikalle, joka oli artefaktin tarjoamalla toteutuksella tavoitteena. Lisätoimipaikkojen tuominen osaksi testiajoa vaatisi muutoksia automaatioputkeen. Jenkins tarjosi CI-putken toteuttamiseen selkeän syntaksin ja useita tärkeitä toiminnallisuuksia, kuten tiettyjen vaiheiden rinnakkaisen ajamisen sekä toistuvien ajojen ajastamisen. Jenkinsin muovautuvuus mahdollistaa automaatioputken muuttamisen tukemaan myös useampia toimipaikkoja. Täten uusien toimipaikkojen lisääminen automaatiotesteihin tulisi onnistua kohtuullisella vaivalla. Useamman toimipaikan yhdenaikaisen toiminnan voisi nähdä lisäävän hajautuksen kautta tulevaa kuormitusta, joten tämä on hyvä nähdä yhtenä jatkokehityskohteena artefaktilla.

Robot Frameworkin avulla saatiin muodostettua robottitestejä, jotka kykenevät vuorovaikuttamaan käyttöliittymän kanssa luoden ohjelmistolle kuormitusta. Robottitesteihin oli mahdollista luoda kohtuullisella vaivalla uusia avainsanoja käyttäen Java-ohjelmointikieltä, jolla testattava ohjelmisto on pääosin luotu. Tämän voi nähdä selkeänä etuna Robot Frameworkin käytölle. Robot Framework mahdollistaa myös käyttöliittymän kanssa vuorovaikutuksen esimerkiksi hiiren avulla, mutta tämä sisältää paljon heikkouksia. Hiiren avulla vuorovaikuttaminen ei ollut tarpeeksi luotettavaa, että olisi voitu luottaa kestotestijakson aikana tapahtuvien satojen toistokertojen onnistuvan aina samalla tavalla. Muodostetut robottitestit tehtiin tästä johtuen hyödyntämään avainsanoja, jotka käskevät järjestelmää ohjelmallisesti ilman vuorovaikutusta käyttöliittymän kanssa. Joiltain osin robottitesteissä hyödynnetään esimerkiksi hiiritoimintoja, joiden toiminnan virheettömyyttä on syytä tarkkailla. Robot Frameworkilla kirjoitettuja testitapauksia on niiden selkeän syntaksin vuoksi kuitenkin helppo jatkokehittää.

Koska robottitestit hyödyntävät laajalti avainsanoja, jotka käskevät järjestelmää suoraan ohjelmallisesti käyttöliittymäsyötteiden sijaan, tulevat ne hyvin todennäköisesti toimimaan ohjelmistoversiosta toiseen. Tällaisissa tapauksissa muutokset ohjelmiston toiminnassa siirtyvät suoraan myös testitapauksiin. Osa testitapauksista hyödyntää tietyissä dialogeissa olevia

tekstejä, joissa tapahtuvat muutokset vaatisivat testitapausten päivittämistä. Yksittäiset robottitestit vuorovaikuttavat käyttöliittymän kanssa hiiritoimintoja käyttäen, joten muutokset käyttöliittymässä voivat vaikuttaa näiden toimivuuteen. Pelkästään testaustyöaseman näytön resoluution eroaminen kehittämiseen käytetyn työaseman resoluutiosta aiheutti ongelmia lopullisessa testien toiminnassa. Nämäkin testit ovat kuitenkin muutettavissa käyttämään suoraan ohjelmiston metodeja kutsuvia avainsanoja, joten muutoksiin mahdollisesti vaadittava työmäärä jää näin pieneksi. Suuret muutokset ohjelmiston toimintalogiikassa voivat luoda painetta myös testitapausten muutoksille.

Muodostettuun artefaktiin on myös mahdollista lisätä robottitestejä suhteellisen runsaasti, ilman CI-putkessa vaadittavia muutoksia. Robottitestien määrän lisääntyessä voimakkaasti, tulee myös testiajojen ajastusta tarvittaessa harventaa, sillä nykyisellään automaatiotestit suoritetaan tunneittain. Testiajon pidentyminen voisi muuten johtaa suoritusten jonoutumiseen. Myös ajastuksen muuttaminen on hyvin yksinkertainen operaatio.

Resurssimittausten osalta robottitestien avainsanat on muodostettu siten, että tiedon tallentamista käytetystä kekomuistista ja prosessorikuormasta voidaan haluttaessa lisätä robottitesteissä siihen käytettyä avainsanaa toistamalla. Myös muistivedoksia voidaan tarvittaessa ottaa useampia testijakson aikana.

Työaseman kuormituksesta tieto tallennetaan yksiselitteisenä datana CSV-formaatissa olevaan tiedostoon. Tässä tiedostossa kerrotaan aikaleimoilla merkittynä prosessorikuorma prosentteina, ohjelmiston käyttämän kekomuistin määrä sekä käytetyn muistin muutos verrattuna testijakson alun vastaavaan. Tästä datasta voidaan helposti nähdä, jos ohjelmistossa on viitteitä muistivuodoista. Yksittäisellä testiajolla resurssienkäytöstä tallennetaan aina yhden mittausajankohdan resurssitiedot. Tämä johtaa siihen, ettei tästä tiedostosta yksinään voida erotella, mikä testitapaus mahdollisen muistivuodon aiheuttaa. Robottitestit tallentavat testitapausten aikaleimat ja suoritusajat. Näistä on mahdollista nähdä yksittäisen testitapauksen hidastuminen peräkkäisiä testiajoja vertailemalla. Robottitestit voivat siis auttaa löytämään yksittäisiä testitapauksia, joiden suoritusajoissa pidentymistä tapahtuu.

Testijakson aikana tallennettiin lisäksi muistivedoksia, joista on mahdollista selvittää tarkemmin mahdollisten muistivuotojen aiheuttaja. Muistivedosten tulkitseminen vaatii esimerkiksi

Eclipse Memory Analyzer -ohjelman käyttöä, eikä tulosten tulkinta näiden osalta ole yksiselitteistä. Muistivedoksista muodostettiin ohjelmallisesti myös histogrammivertailut, joista on mahdollista nähdä muistia kuluttavien Java-luokkien nimiä. Tulosten tarkkaan tulkintaan on testaajan ja ohjelmistokehittäjän yhteistyö suositeltavaa. Tulosten tulkinnan osalta siis vaaditaan jonkin verran manuaalista työtä, joka tulee huomioida kestotestijaksoja suunniteltaessa.

Myös testien suorittaminen vaatii manuaalista työtä, sillä testijakso käynnistetään ja sammutetaan Jenkinsin kautta testaajan toimesta. Toistuvat testiajot tapahtuvat kuitenkin automaattisesti. Testaajan on mahdollista myös keskeyttää testiajo siten, että sitä voidaan myöhempänä ajankohtana jatkaa. Testiajot eivät varsinaisesti vaadi jatkuvaa seurantaa, mutta se on jossain määrin suositeltavaa virhetilanteiden varalta. Seuranta voidaan suurimmaksi osaksi tehdä ilman suoraa vuorovaikutusta testaukseen käytetyn työaseman kanssa. Mahdollinen virhetilanne useimmiten johtaa robottitestien epäonnistumiseen, joka on nähtävissä Jenkinsin hallintaan käytetylle verkkosivulle kerättävistä lokeista. Myös mahdollinen resurssinkäytön nousu tarkistetaan, jolloin siitä aiheutuu merkintä robottitestien lokeihin. Epäonnistuneiden robottitestien määrä ja yksittäiseen testiajoon kulunut aika on tarkistettavissa hyvin nopeasti, mikä vähentää manuaaliseen työhön kuluvaa aikaa. Merkittävin osa manuaalisesta työstä kuluu siis testijakson lopussa tulosten tulkintaan.

Robottitestien avulla kestotestausta voidaan suorittaa tarvittaessa jatkuvasti. Kestotestijakson pituus on päätettävissä portaattomasti, joten lähtötilanteeseen verrattuna testauksen määrää on helppo lisätä runsaasti vähäisellä manuaalisella työllä. Automaatiotestit eroavat merkittävästi aiemmin suoritetuista testitapauksista, jotka kuuluivat suurimmalta osalta toiminnalliseen testaukseen. Automaatiotestien pyrkimys on ensisijaisesti kuormittaa käyttöliittymää toiminnallisuuden testaamisen sijaan. Täten muodostetut automaatiotestit tuovat kestotestijaksoa lähtötilanteeseen verrattuna lähemmäs kestävyystestausta toiminnallisuustestauksen sijaan.

Harjoitustestijaksoja järjestettiin, jotta voitaisiin tarkkailla testiautomaation toimivuutta. Testijaksoilla löydettiin merkkejä muistivuodoista testattavassa ohjelmistossa. Tämä antaa viitteitä siitä, että muodostetun artefaktin avulla voidaan löytää ohjelmistossa ilmeneviä poikkeamia. Yleisesti ottaen poikkeamat huomattiin testiajojen pidentymisenä ja muistinkäytön lisääntymisenä. Joissakin tapauksissa testiajon pidentymiselle ei löytynyt syytä, joten jää

epäselväksi, ovatko kaikki hidastumiset merkkejä testattavan ohjelmiston poikkeamista vai johtuuko osa esimerkiksi Robot Frameworkista tai työaseman suorittamista taustaprosesseista. Tarvittaessa automaatiotestejä voisi kehittää siten, että myös muiden kestotestiin liittyvien ohjelmistojen kuormitusta tarkkailtaisiin, jotta esimerkiksi käytetyn työaseman hidastuminen ja testattavan käyttäjäsovelluksen hidastuminen voidaan erottaa toisistaan. Tämän lisäksi voidaan tarvittaessa tutkia myös muita ohjelmallisia ratkaisuja Robot Frameworkin tilalla käytettäväksi.

Lähtötilanteessa kestotestijaksot vaativat runsaasti manuaalista työtä, jota automaation avulla pyrittiin helpottamaan. Kestotestijaksoja voidaan jatkossa suorittaa useammin ja yksittäisen jakson suorittaminen vaatii vähemmän henkilötyövoimaa. Lähtötilanteessa vaaditut alkuvalmistelut, kuten testausympäristöjen asennukset ovat vaatineet useita työtunteja. Nykytilanteessa kestotestijakson alkuvalmistelu tarkoittaa sitä, että testaaja tarkistaa esimerkiksi etäyhteydellä testaamisessa käytettävien tietokoneiden olevan käynnissä ja käynnistää testiautomaation testaukseen valitulla ohjelmistoversiolla Jenkinsin kautta. Tämä voidaan suorittaa alle tunnissa, jos lasketaan mukaan aika, joka testaajalta kuluu seuratessaan asennuksen ja ensimmäisten automaatiotestien onnistuneen suorituksen.

Noin kuukauden mittaiset kestotestijaksot ovat sisältäneet lähtötilanteessa aktiivista testitapausten suoritusta sisältävien päivien lisäksi päiviä, jolloin testattavaa ohjelmistoa ei ole manuaalisesti kuormitettu. Aktiivisina päivinä testitapausten suoritukseen arvioidaan käytetyn noin 1–2 tuntia kerrallaan. Keskiarvoltaan tällaisia päiviä on ollut kestotestijaksoista noin puolet eli 14 kappaletta. Ilman testitapausten suorituksen estäviä virhetilanteita, kuten järjestelmässä tai laitteistossa tapahtuvia virheitä, testiautomaation suorittamat testitapaukset eivät vaadi jatkuvaa seurantaa ja mahdollisesti epäonnistuvien testitapausten yksityiskohdat voidaan tarkistaa jälkikäteen lokitiedoista minuuteissa. Testattavassa ohjelmistossa mahdollisesti ilmenevien virhetilanteiden selvitykseen kuluva aika on tilannekohtainen, mutta sen voidaan arvioida pysyvän yhtäläisenä lähtötilanteeseen nähden.

Lähtötilanteessa kestotestijakson tulosten tulkinta ja kirjaaminen on vienyt muutamia tunteja. Tähän kuluvan ajan voidaan arvioida pysyvän yhtäläisenä lähtötilanteeseen nähden, sillä tulosten tulkinta vaatii manuaalista työtä. Manuaalisesti suoritetuilla kestotestijaksoilla on esimerkiksi kerätty tietoa ohjelmiston tilasta muistivedoksia tallentamalla, mutta tämä suori-

tetaan automatisoiduilla kestotestijaksoilla automaattisesti. Tämän työvaiheen osalta nykytilassa työaikaa säästetään korkeintaan joitakin minuutteja verrattuna lähtötilanteeseen.

Testiautomaation avulla yhdellä kestotestijaksolla voidaan kokonaisuudessaan arvioida säästettävän aikaa jopa kymmeniä tunteja. Säästyvä aika nähdään tarkemmin, kun tulevilla kestotestijaksoilla saadaan lisätietoa esimerkiksi siitä, eroaako testiautomaation avulla kerättävien tulosten tulkinnan vaatima aika manuaalisten kestotestijaksojen vastaavasta. Automaatiolla säästyvä aika tulee myös olemaan sitä suurempi, mitä useampia kestotestijaksoja tulevaisuudessa voidaan suorittaa ilman testiautomaatiossa vaadittavia muutoksia.

## 6.3 Tutkielman suorituksen arviointi

Tutkielman tavoitteena oli luoda testauskonsepti, jolla toteutetaan kohteena olevan sovelluksen testausta kestävyystestauksen näkökulmasta. Tutkimuskysymyksen mukaisesti tutkimuksen aikana testiautomaation avulla korvattiin lähtötilanteessa manuaalisesti suoritettavaa kestävyystestausta. Tutkielman avulla saatiin tietoa muun muassa käytettyjen teknologioiden sopivuudesta kestävyystestaukseen ja esitettiin huomioita tukemaan mahdollista jatkotutkimusta.

Tutkimusmenetelmänä suunnittelututkimuksen voidaan nähdä sopineen tutkimuksen tavoitteisiin hyvin. Tutkimuksen aikana suunniteltiin ja toteutettiin sovelluksen automaattisen kestotestauksen mahdollistava artefakti. Artefaktin arvioidaan saavuttaneen siltä vaaditut tavoitteet. Käytetyn tutkimusmenetelmän tekninen lähestymistapa tuki ohjelmistoteknologisen kokonaisuuden luomisprosessia. Käytännönläheinen lähestyminen tuki ratkaisun selvittämistä tähän reaalimaailman haasteeseen, jonka testausprosessin kehittäminen asetti.

# 7 Johtopäätökset

Ohjelmistotestaus on tärkeä osa ohjelmistokehitystä, jossa tavoitteena on saada aikaan luotettava ja ohjelmistolta vaaditut tavoitteet saavuttava kokonaisuus. Kestävyystestauksen rooli on varmistaa ohjelmiston suoriutuminen päivittäisestä kuormituksesta siten, että ohjelmiston kaikkien osien toiminta voidaan varmistaa myös käytettäessä ohjelmistoa pitkällä aikavälillä.

Tutkimuksessa pyrittiin selvittämään, miten automaatiotestauksen avulla voitaisiin korvata kohteena olevan sovelluksen lähtötilanteessa manuaalisesti suoritettavaa kestotestausta. Ratkaisuna toteutettiin artefakti, jonka avulla sovelluksen luotettavuutta tarkkaillaan kestävyystestauksen näkökulmasta. Tämä tapahtuu kuormittamalla ohjelmistoa tavanomaisilla toiminnoilla automaatiotestien avulla tallentaen samanaikaisesti tietoa sen resurssinkäytöstä. Lisäksi hyödynnetään riittävän pitkiä testausjaksoja, jotta saadaan kerättyä tietoa ohjelmiston pitkäaikaisesta käytöstä. Artefakti mahdollista tutkimuksen kohteena olevan järjestelmän automaattisen testaamisen ilman jatkuvaa testaajan läsnäoloa. Näin saadaan kevennettyä kestotestien vaatimaa manuaalisen työn määrää. Kestotestejä voidaan täten nykytilassa myös suorittaa lähtötilannetta useammin. Automaatiotestien avulla on kyetty tunnistamaan ohjelmistosta muistivuotoja, jotka ovat yksi kestävyystestauksen aikana mahdollisesti ohjelmistoissa esille nouseva ongelma.

Tutkimuksen perusteella kestotestausprosessin kehittäminen nähtiin kannattavana, sillä muodostetun artefaktin avulla saatiin varteenotettavaa tietoa ohjelmiston toiminnasta jo artefaktin kehitystyön aikana. Artefakti myös tarjoaa hyvän pohjan kestotestiprosessin automatisoinnin jatkokehitykselle.

Tutkimuksen aikana voitiin tunnistaa useita organisaation näkökulmasta hyödyllisiä jatkotutkimuskohteita. Ohjelmiston osien tarkempi analysointi kuormittavimpien osa-alueiden löytämiseksi auttaisi testiautomaation jatkokehityksessä testattavien osien valinnassa. Lisäksi kuormituksen mittaustapojen ja mittauksessa käytettävien työkalujen vertailulla ja kehitystyöllä voitaisiin mahdollisesti saada tarkempaa tietoa ohjelmiston kuormituksesta. Tässä yhtenä toimintamallina voisi olla manuaalisen testaamisen yhdistämisen automaation käyttöön siten, että testaaja voisi ohjelmallisesti suorittaa esimerkiksi robottitestien avulla testitapauk-

sia seuraten samalla ohjelmiston käyttäytymistä. Tällöin ohjelmistolle luotava kuorma saataisiin kasvatettua robottitestien mahdollistamalle tasolle hyödyntäen manuaalitestaajan taitoa ohjelmiston toiminnan tarkkailussa. Tutkimuksessa automaatiotestien muodostamisessa hyödynnettiin avoimeen lähdekoodiin perustuvia ohjelmistokirjastoja kuten Robot Frameworkia ja RemoteSwingLibrarya. Jatkotutkimuskohteena voidaan nähdä vaihtoehtoisten ohjelmistokirjastojen selvittämisen ja näiden suorituskyvyn keskinäisen vertailun hyödynnettäessä niitä kestävyystestauksessa.

# Lähteet


Abrahamsson, Pekka, Outi Salo, Jussi Ronkainen ja Juhani Warsta. 2017. “Agile software development methods: Review and analysis”. *arXiv preprint arXiv:1709.08439.*

Adenowo, Adetokunbo AA ja Basirat A Adenowo. 2013. “Software engineering methodologies: a review of the waterfall model and object-oriented approach”. *International Journal of Scientific & Engineering Research* 4 (7): 427–434.

Alghmadi, Hammam M, Mark D Syer, Weiyi Shang ja Ahmed E Hassan. 2016. “An automated approach for recommending when to stop performance tests”. Teoksessa *2016 IEEE international conference on software maintenance and evolution (ICSME),* 279–289. IEEE.

Ammann, Paul ja Jeff Offutt. 2016. *Introduction to Software Testing.* 2. painos. Cambridge, England: Cambridge University Press.

Bloch, Joshua. 2017. *Effective Java.* 3. painos. Boston, MA: Addison-Wesley Educational.

Collard, Ross. 2005. *System Performance Testing, Case Study.*

Collins, Eliane, Arilo Dias-Neto ja Vicente F de Lucena Jr. 2012. “Strategies for agile software testing automation: An industrial experience”. Teoksessa *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops,* 440–445. IEEE.

Darwin, Ian F. 2001. *Java Cookbook.* Sebastopol, CA: O’Reilly Media.

Distefano, Dino ja Ivana Filipovic. 2010. “Memory Leaks Detection in Java by Bi-abductive Inference.” Teoksessa *FASE,* 10:278–292. Springer.

Dustin, E, J Rashka ja J Paul. 1999. *Automated software testing: Introduction, management, and performance: Introduction, management, and performance.* Addison-Wesley Professional.

Garousi, Vahid ja Michael Felderer. 2017. “Worlds apart: Industrial and academic focus areas in software testing”. *IEEE Softw.* 34 (5): 38–45.

Garousi, Vahid ja Mika V Mäntylä. 2016. “A systematic literature review of literature reviews in software testing”. *Inf. Softw. Technol.* 80:195–216.

Ghanavati, Mohammadreza, Diego Costa, Artur Andrzejak ja Janos Seboek. 2018. “Memory and resource leak defects in java projects: An empirical study”. Teoksessa *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings,* 410–411.

Ghanavati, Mohammadreza, Diego Costa, Janos Seboek, David Lo ja Artur Andrzejak. 2020. “Memory and resource leak defects and their repairs in Java projects”. *Empirical Software Engineering* 25:678–718.

Hevner, Alan ja Samir Chatterjee. 2010. “Design Science Research in Information Systems”. Teoksessa *Integrated Series in Information Systems,* 9–22. Springer US.

Hynninen, T, J Kasurinen, A Knutas ja O Taipale. 2018. “Software testing: Survey of the industry practices”. Teoksessa *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO).* Opatija: IEEE.

ISO/IEC 25010. 2011. *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.*

ISO/IEC 29119-1. 2022a. *ISO - ISO/IEC 29119-1:2022 -Software and systems engineering — Software testing — Part 1: General concepts.*

———. 2022b. *ISO - ISO/IEC 29119-1:2022 -Software and systems engineering — Software testing — Part 4: Test techniques.*

Jamil, Muhammad Abid, Muhammad Arif, Normi Sham Awang Abubakar ja Akhlaq Ahmad. 2016. “Software testing techniques: A literature review”. Teoksessa *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M).* Jakarta, Indonesia: IEEE.

“Jenkins Robot Plugin”. 2011. Viitattu 6. toukokuuta 2023. https://plugins.jenkins.io/robot/.

“Jenkins User Documentation”. 2011. Viitattu 16. huhtikuuta 2023. https://www.jenkins.io/doc/.

Jones, Richard ja Rafael Lins. 1996. *Garbage collection: algorithms for automatic dynamic memory management.* John Wiley & Sons, Inc.

Kaner, Cem, James Bach ja Brett Pettichord. 2002. *Lessons learned in software testing.* Nashville, TN: John Wiley & Sons.

Kasurinen, Jussi Pekka. 2015. *Ohjelmistotestauksen käsikirja.* Docendo.

Laaber, Christoph. 2019. "Continuous software performance assessment: detecting performance problems of software libraries on every build". Teoksessa *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* Beijing China: ACM.

Naik, Kshirasagar ja Priyadarshi Tripathy. 2013. *Software testing and quality assurance: Theory and practice.* Barking, England: Lulu.com.

Oliinyk, B ja V Oleksiuk. 2019. "Automation in software testing, can we automate anything we want". Teoksessa *Proceedings of the 2nd Student Workshop on Computer Science & Software Engineering (CS&SE@ SW 2019),* 224–234. Kryvyi Rih; Ukraine.

"Oracle Java Documentation - Jmap". 2014. Viitattu 14. toukokuuta 2023. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jmap.html.

"Oracle Java Documentation - JVisualVM". 2014. Viitattu 29. toukokuuta 2023. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jvisualvm.html.

"RemoteSwingLibrary". 2014. Viitattu 16. huhtikuuta 2023. https://github.com/robotframework/remoteswinglibrary.

"Robot Framework Documentation". 2021. Viitattu 16. huhtikuuta 2023. https://robotframework.org/robotframework/4.1.3/RobotFrameworkUserGuide.html.

Srivastava, Nishi, Ujjwal Kumar ja Pawan Singh. 2021. "Software and Performance Testing Tools". *J. Infor. Electr. Electron. Eng.* 2 (1): 1–12.

Taipale, Ossi, Jussi Kasurinen, Katja Karhu ja Kari Smolander. 2011. "Trade-off between automated and manual software testing". *Int. J. Syst. Assur. Eng. Manag.* 2 (2): 114–125.

"The Eclipse Memory Analyzer". 2023. Viitattu 14. toukokuuta 2023. https://www.eclipse.org/mat/.

Umar, Mubarak Albarka ja Chen Zhanfang. 2019. “A Study of Automated Software Testing: Automation Tools and Frameworks”. *International Journal of Computer Science Engineering (IJCSE)* 6:217–225.

Weyuker, Elaine J ja Filippos I Vokolos. 2000. “Experience with performance testing of software systems: issues, an approach, and case study”. *IEEE transactions on software engineering* 26 (12): 1147–1156.