VAASAN YLIOPISTO

TEKNIIKAN JA INNOVAATIOJOHTAMISEN YKSIKKÖ

OHJELMISTOTEKNIIKKA

Jani Lehtisalo

LAITTEISTORAJAPINNAN TOTEUTUS TESTIAUTOMAATIOJÄRJESTELMÄÄN

Diplomityö, joka on jätetty tarkastettavaksi diplomi-insinöörin tutkintoa varten Vaasassa 27.3.2020

Työn valvoja Jouni Lampinen

Työn ohjaaja Mika Filander

SISÄLLYSLUETTELO

SYMBOLI- JA LYHENNELUETTELO 3

1 JOHDANTO 7

2 OHJELMISTOARKKITEHTUURI 10

2.1 Ohjelmistorajapinta 10
2.2 Kerrosarkkitehtuuri 12
2.3 Plug-in-kehys 13
2.4 REST-arkkitehtuurimalli 14
2.5 Malli-näkymä-ohjain-arkkitehtuuri 15
2.6 Play-ohjelmistokehys 17
2.7 Docker-säiliöinti 19

3 OHJELMISTOTESTAUS 22

3.1 Testaustasot 22

3.1.1 Yksikkötestaus 23
3.1.2 Integraatiotestaus 24
3.1.3 Järjestelmätestaus 25
3.1.4 Hyväksymistestaus 25

3.2 Testausmenetelmät 26

3.2.1 Regressiotestaus 26
3.2.2 Testausautomaatio 26

4 TYÖN SUUNNITTELU 29

4.1 Työvaiheet ja toteutus 29
4.2 Asiakasprojektin esittely 31
4.3 Asiakaslaitteet ja niiden ominaisuudet 32

5 RAJAPINNAN TOTEUTUS 34

5.1 Vaatimusten määrittely 35

5.2 Ohjelmistoarkkitehtuurin suunnittelu 39
5.3 Projektipohjan luominen 42
5.4 Rajapinnan ominaisuuksien toteuttaminen 44
5.5 Yksikkö- ja integraatiotestaus 48
5.6 Laitteistorajapinnan käyttö Docker-säiliössä 49
5.7 Laadun arviointi 50
5.8 Järjestelmätestaus 52

6 TULOKSET 59

6.1 Työn eteneminen ja havainnot 59
6.2 Yleinen laitteistorajapinta 61
6.3 Projektikohtainen laitteistorajapinta 63

7 JOHTOPÄÄTÖKSET 65

LÄHDELUETTELO 67

LIITTEET 71

LIITE 1. Laitteistorajapinnan yksityiskohtainen kuvaus 71
LIITE 2. Laitteistorajapinnan arkkitehtuuri luokkakaaviona 76
LIITE 3. Laitteistorajapinnan ominaisuuksien toteutuksen kuvaus 77

SYMBOLI- JA LYHENNELUETTELO

DPC	Devatus Partner Cloud, testiautomaatiojärjestelmän työnimi yrityksessä.
HTTP	Hypertext Transfer Protocol, muun muassa verkkoselaimien käyttämä tiedonsiirtoprotokolla.
IEEE	Institute of Electrical and Electronics Engineers, kansainvälinen tekniikan alan järjestö.
IP	Internet Protocol, numerosarja verkkoon kytkettävien laitteiden yksilöintiin.
JSON	JavaScript Object Notation, avoimen standardin tiedostomuoto tiedonvälitykseen.
MVC	Model-View-Controller, malli-näkymä-ohjain arkkitehtuuri.
REST	Representational State Transfer, HTTP-protokollaan perustuva arkkitehtuurimalli.
RPC	Remote Procedure Call, etäproseduurikutsu kommunikoinnin toteuttamiseksi hajautetussa järjestelmässä.
SOAP	Simple Object Access Protocol, sovellusohjelmien välinen viestipohjainen tietoliikenneprotokolla.
UML	Unified Modeling Language, standardoitu graafinen ohjelmiston mallinnuskieli.
URI	Uniform Resource Identifier, verkkoympäristössä käytettävä merkkijono, joka määrittelee resurssin sijainnin.
USB	Universal Serial Bus, elektronisten laitteiden kytkennässä käytettävä sarjaväyläarkkitehtuuristandardi.
WS	Web Service, ohjelmistojärjestelmä, joka mahdollistaa tietokoneiden välisen kommunikoinnin tietoverkon yli.
YAML	Tiedon serialisointiin käytettävä standardoitu dokumentointikieli.

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen yksikkö
Tekijä: Jani Lehtisalo
Diplomityön nimi: Laitteistorajapinnan toteutus testiautomaatiojärjestelmään
Valvoja: Jouni Lampinen
Ohjaaja: Mika Filander
Tutkinto: Diplomi-insinööri
Koulutusohjelma: Energia- ja informaatiotekniikan koulutusohjelma
Suunta: Ohjelmistotekniikka
Opintojen aloitusvuosi: 2010
Diplomityön valmistumisvuosi: 2020 **Sivumäärä:** 70

TIIVISTELMÄ:

Tämän diplomityön tarkoituksena on suunnitella laitteistorajapinta ohjelmistoyrityksen kehittämän testiautomaatiojärjestelmän ja siihen kytkettävien sulautetun järjestelmän sisältävien laitteiden välille. Järjestelmän tehtävänä on suorittaa erilaisia automatisoituja testejä ohjelmistoyrityksen asiakkaiden kehittämille laitteita ohjaaville sovelluksille. Rajapinnan on tarkoitus mahdollistaa laitteiden automatisoitu valmistelu testaukseen.

Työstä on rajattu pois yrityksen asiakkaiden kehittämät ohjelmistot, niihin liittyvät rajapinnat, järjestelmään liitettävät mobiililaitteet sekä itse testiautomaatio. Laitteistorajapinnan toteutuksessa keskitytään tiettyyn, ohjelmistoyrityksen valitsemaan asiakasprojektiin ja siinä esiin nouseviin vaatimuksiin, joiden pohjalta tässä työssä tavoiteltu rajapinta on tarkoitus kehittää. Tavoitteena on luoda asiakasvaatimukset täyttävä, toimiva ohjelmistorajapinta asiakaslaitteiden ja testijärjestelmän välille, mitä ohjelmistoyritys pystyy hyödyntämään myös muissa vastaavanlaisissa asiakasprojekteissaan.

Työn alussa laadittiin suunnitelma rajapinnan arkkitehtuurista ja halutuista toiminnoista. Tämän jälkeen toiminnot toteutettiin ketterän kehityksen menetelmiä soveltaen priorisoidussa järjestyksessä. Tavoitteena oli saada vaatimukset täyttävä laitteistorajapinta valmiiksi kymmenen kuukauden kehitystyön jälkeen. Suurin osa työstä tehtiin ohjelmointityönä, käyttäen Java-kieltä ja Play-ohjelmistokehystä.

Työn lopputuloksena luotiin annettujen vaatimusten mukaisesti skaalautuva ohjelmistorajapinta, jota on mahdollista hyödyntää tutkimustyön aikaisen asiakasprojektin lisäksi myös seuraavissa projekteissa. Työn aikana saatiin tietoa testiautomaatiojärjestelmän teknisestä soveltuvuudesta uusiin tilanteisiin ja käyttötarkoituksiin. Työn lopputulos loi myös edellytyksiä järjestelmän tuotteistamiselle tulevaisuudessa laajempaan käyttöön.

AVAINSANAT:
Testiautomaatio, regressiotestaus, ohjelmistorajapinta, plug-in-arkkitehtuuri, REST-arkkitehtuurimalli

UNIVERSITY OF VAASA
The School of Technology and Innovations

Author:	Jani Lehtisalo
Topic of the Thesis:	Implementation of hardware device interface for test automation system
Supervisor:	Jouni Lampinen
Instructor:	Mika Filander
Degree:	Master of Science in Technology
Degree Programme:	Degree Programme in Energy and Information Technology
Major of Subject:	Software Engineering
Year of Entering the University:	2010
Year of Completing the Thesis:	2020 **Pages:** 70

ABSTRACT:

The goal of this research is to design hardware interface between test automation system developed by software company and devices with embedded systems. The purpose of the test automation system is to enable running automated test cases for customer applications that control embedded systems. The purpose of the hardware interface is to enable automated preparation of devices for testing.

The research definition excludes customer developed application, other customer software, related interfaces, mobile devices and test automation itself. The implementation of the hardware interface is done by focusing to an individual customer project case and its requirements for the interface. Goal is to create a working hardware interface that fulfills the requirements of the case study. When completed software company would be able to utilize the common hardware interface for other similar customer projects as a part of their test automation system.

Before the starting of implementation a plan for architectural structure and desired features was created. The implementation was carried out by applying agile methods for prioritized work list. Goal was to complete the hardware interface after 10 months of development work. Majority of the work was carried out as programming work by using Java programming language with Play framework.

As a result of the research the scalable hardware interface was created based on given case study requirements. Because of its modularity the developed interface can also be utilized to other projects in the future. Research also provided information about the technical suitability of the test automation system for new use cases. Research and the resulting interface have also created foundation for commercialization of the test automation system.

KEYWORDS:
Test automation, regression testing, application programming interface, plug-in architecture, REST architecture

1 JOHDANTO

Tässä diplomityössä perehdytään Devatus Oy:n kehittämään DPC-testiautomaatiojärjestelmään, joka on luotu käytettäväksi yrityksen asiakkaiden tarpeisiin. Itse järjestelmä toimii Devatuksen ylläpitämällä palvelimella, jonka luomassa ympäristössä voidaan testata asiakkaiden itse kehittämiä sulautetun järjestelmän sisältävän laitteen ohjaamisen mahdollistavia sovelluksia. DPC-järjestelmään voidaan fyysisesti kytkeä erilaisia mobiililaitteita, joille asiakkaan kehittämä sovellus ladataan, ja sen toimintaa testataan automatisoidusti. Järjestelmän laitteistorajapinta mahdollistaa sovelluksen kanssa kommunikoimaan tarkoitettujen sulautettujen järjestelmien automaattisen valmistelun testausta varten.

DPC-järjestelmän ensisijainen tarkoitus on ajaa asiakkaan kehittämiä ohjelmia varten luodut automatisoidut testit erilaisissa, asiakkaan toiveiden mukaisissa testiolosuhteissa, ja lopuksi palauttaa testitulokset käyttäjälle. Tätä varten järjestelmään voidaan kytkeä erilaisilla käyttöjärjestelmillä varustettuja eri valmistajien mobiililaitteita sekä erilaisia sulautettuja järjestelmiä.

Ennen tämän diplomityön aloitusta eri laitteiden ja ohjelmien liittäminen järjestelmään tapahtui manuaalisesti ohjelmakoodia ja asetuksia muokkaamalla. Automatisoitu testaus oli mahdollista tiettyjen tiukkojen reunaehtojen puitteissa, ja muutokset järjestelmän vaatimuksiin olisivat edellyttäneet laajoja muutoksia tai jopa koko järjestelmän uudelleenrakentamista. Tämä vei tarpeettoman paljon työstä vastuussa olevien henkilöiden aikaa. Järjestelmän käytettävyyden parantamiseksi päätettiin ensimmäisenä kehitysaskeleena luoda useiden laitteiden kanssa yhteensopiva laitteistorajapinta ja sitä ohjaava väyläarkkitehtuuri parantamaan kommunikaatiota DPC-järjestelmän ja asiakaslaitteiden välillä.

Laitteistorajapinta toteutettiin plug-in-arkkitehtuurina, jossa rajapinnan ytimenä toimivaan yleiseen rajapintaan on mahdollista liittää uusia projektikohtaisia rajapintoja, niin sanottuja plug-in-kirjastoja, jotka mahdollistavat kommunikoinnin uusien laitteiden kanssa. Tämän diplomityön aihealue on rajattu tämän plug-in-arkkitehtuuria hyödyntävän yleisen rajapinnan ja yhden sen toteuttavan asiakasprojektikohtaisen plug-in-komponentin kehittämiseen.

Jotta laitteistorajapinnan toiminnalliset vaatimukset pystyttäisiin rajaamaan työn kannalta selkeästi, päätettiin tutkimuksessa keskittyä tiettyyn ohjelmistoyrityksen valitsemaan asiakasprojektiin ja sen järjestelmälle asettamiin vaatimuksiin. Asiakasprojektin vaatimusten pohjalta hahmoteltiin yleisen rajapinnan ominaisuudet, jotka myös tulevien asiakasprojektien tulisi täyttää.

Testiautomaatiojärjestelmän muiden tulevien osajärjestelmien, kuten esimerkiksi mobiililaitteita hallinnoivien komponenttien sekä testikokoonpanoja ja testitapauksien jonotusjärjestelmää hallinnoivan logiikkaytimen tarkka kuvaus ja toteutus on rajattu työstä pois niiden laajuuden vuoksi. Niiden toimintaa ja roolia käsitellään kuitenkin suppeasti osana tätä tutkimusta. Osa mainituista komponenteista tullaan kehittämään omina projekteinaan Devatus Oy:llä samanaikaisesti tämän diplomityöprojektin rinnalla.

Työn keskittymiskohteen kannalta epäolennaiset testiautomaatiojärjestelmässä käytettävät mobiililaitteet sekä asiakkaiden kehittämät sovellukset, niiden tarkat ominaisuudet ja niillä ajettavat testitapaukset on rajattu työstä pois, koska ne hyödyntävät järjestelmän mobiilirajapintaa eivätkä siten vaikuta laitteistorajapinnan toimintaan suoraan. Mahdolliset mobiilirajapinnan asettamat vaatimukset laitteistorajapinnalle kuitenkin huomioidaan. Koska laitteistorajapinta suunnitellaan alustamaan järjestelmään liitetyt sulautetut järjestelmät testausta varten, oletetaan testattavan sovelluksen kykenevän kommunikoimaan laitteistorajapinnan laitteiden kanssa.

Laitteistorajapinnan tietoturvaan liittyvät seikat päätettiin jättää työstä pois työmäärän rajaamiseksi. Tästä syystä myös tietoturvaan liittyvät rajapintavaatimukset on jätetty työstä kokonaan pois.

Työn tavoitteena on, että ohjelmistokehitystyön päätteeksi ohjelmistoyrityksellä on käytettävissään laitteistorajapinta, joka täyttää vaatimusmäärittelyssä sille annetut ei-toiminnalliset vaatimukset ja esimerkkiprojektin asiakaslaitteiden sille antamat tärkeimmät toiminnalliset vaatimukset. Tavoitteena on pyrkiä rakentamaan rajapinta niin, että sen omi-

naisuudet mahdollistavat myös uusien asiakasprojektien ja laitteiden käytön osana testiautomaatiojärjestelmää. Lopuksi työn aikana luodulle rajapinnalle suoritetaan järjestelmätestaus, jossa varmistetaan sen käyttäytyminen odotetulla tavalla.

Mikäli mainitut tavoitteet täyttyvät, voidaan työn tulosta todennäköisesti hyödyntää testiautomaatiojärjestelmän kehittämisessä edelleen lopulliseksi laajalle asiakaskunnalle markkinoitavaksi tuotteeksi. Tutkimuksessa toteutettua yleisen rajapinnan toiminnallisuuden toteuttavaa projektikohtaista rajapintaa tullaan todennäköisesti myös hyödyntämään mallina tuleville projektikohtaisille rajapintatoteutuksille.

2 OHJELMISTOARKKITEHTUURI

IEEE-SA Standards Board:n (2000: 3) standardi 1471-2000 määrittelee ohjelmistoarkkitehtuurin järjestelmän perustavanlaatuiseksi organisaatioksi, johon kuuluvat sen komponentit, niiden suhteet toisiinsa ja ympäristöönsä sekä periaatteet, jotka määrittävät ohjelmistoarkkitehtuurin suunnittelua ja kehitystä.

Ohjelmistoarkkitehtuuri määrittelee suunniteltavan ohjelmiston yksittäisiä komponentteja laajempana kokonaisuutena eli korkeammalla abstraktiotasolla. Ohjelmistoarkkitehtuurin avulla pystytään hahmottamaan järjestelmäkokonaisuuksia paremmin ja voidaan myös jakaa arkkitehtuuri loogisiin, toisistaan erotettavissa oleviin komponentteihin, jotka toimivat myös itsenäisesti. Toteutuksen jakaminen komponentteihin mahdollistaa myös työn jakamisen eri osioihin ohjelmistoprojektissa. Tällöin eri komponentteja voidaan rakentaa samanaikaisesti ja siten tehostaa tuotantoprosessia. Myös komponenttien testaus helpottuu. Ilman arkkitehtuurisuunnittelua useimpien nykyaikaisten ohjelmistojen suunnittelu on lähes poikkeuksetta haastavaa ja virhealtista, ellei mahdotonta. (Koskimies & Mikkonen 2005: 16–17.)

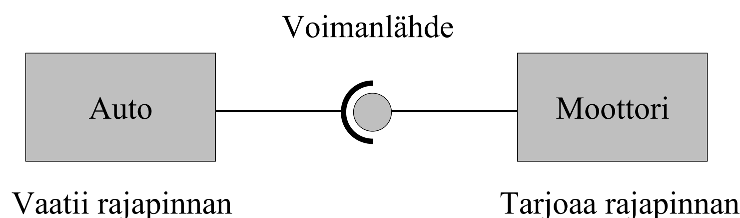
Arkkitehtuurin huolellinen suunnittelu on ohjelmistoprojektin onnistumisen kannalta ratkaisevan tärkeässä asemassa. Väärin toteutettu arkkitehtuuri, joka ei esimerkiksi ota kaikkia olennaisia ohjelmistovaatimuksia huomioon tai tekee vääriä oletuksia sovellettavan teknologian suhteen, voi pahimmillaan estää ohjelmiston käyttöönoton kokonaan. Huono arkkitehtuurisuunnittelu voi myös vaikeuttaa testausta ja ylläpitoa, jos muutosten tekemistä ei ole otettu alkuvaiheessa huomioon tarpeeksi hyvin. (Koskimies & Mikkonen 2005: 17.)

2.1 Ohjelmistorajapinta

Ohjelmistorajapinta on kehitetty välineeksi yhdistää eri ohjelmistokomponentteja toisiinsa yksinkertaisesti. Sen tärkein tehtävä on määritellä, miten tietyn komponentin tai

järjestelmän tarjoama palvelu otetaan käyttöön. Samalla se määrittelee tarvittavat parametrit, niiden tyypit sekä mahdollisen vastauksena saatavan ulostulon tyypin. Laadukas rajapinta määrittelee selkeästi myös tarjotun palvelun ominaisuudet, kuten esimerkiksi toiminnan rajoitteet, mahdolliset poikkeukset ja riippuvuudet ulkoisista resursseista. (Koskimies & Mikkonen 2005: 58.)

Ohjelmistokomponentti voi joko tarjota tai vaatia rajapinnan riippuen järjestelmän rakenteesta. Usein järjestelmässä voi olla useita erilaisia rajapintoja, joten yksittäinen komponentti voi samanaikaisesti sekä tarjota että vaatia useita rajapintoja. Yksinkertainen esimerkki rajapinnan käytöstä on esitetty kuvassa 1, jossa rajapintana toimii voimanlähde. Esimerkissä auto vaatii toimiakseen voimanlähteen. Moottoria taas voidaan käyttää voimanlähteenä autoon, joten se tarjoaa rajapinnan. (Koskimies & Mikkonen 2005: 59–60.)



Kuva 1. Esimerkki rajapinnan suhteesta komponentteihin (Perustuu: Koskimies & Mikkonen 2005: 60).

Alla on esitettynä yksinkertaistettu esimerkki rajapinnasta Java-kielellä toteutettuna. Rajapinta on Javassa niin sanottu abstrakti luokka eli se määrittelee metodit, jotka sen toteuttavan luokan tulee toteuttaa, mutta ei itse toteuta niitä. (Koskimies & Mikkonen 2005: 60.)

```
Interface PowerSource {
    void start();
    int temperature();
    void stop();
}
```

Rajapintojen yhteydessä voidaan puhua myös niin sanotuista roolirajapinnoista, joka tarkoittaa tietyn roolin mukaisten palvelujen toteuttamista. Roolirajapinta voi toimia osana suurempaa rajapintaluokkaa ja toteuttaa vain tietyn roolin mukaiset palvelut. Täten yksittäinen rajapinta voi toteuttaa useamman roolirajapinnan asiakaskomponentin/-komponenttien tarpeiden mukaan. Roolirajapintojen käyttö selkeyttää rajapintojen arkkitehtuuria ja parantaa ylläpidettävyyttä. (Koskimies & Mikkonen 2005: 77–79.)

Ohjelmistorajapintoja voidaan soveltaa hyvin erilaisiin käyttötarkoituksiin. Myös niiden rakenne ja toteutus voivat vaihdella huomattavasti käyttötarkoituksen mukaan. Yleisimpiä rajapintojen sovellusalueita ovat Biehlin (2015: 25) mukaan mobiilisovellukset, pilvipalvelut, web-sovellukset, järjestelmäintegraatiot, älytelevisiot, esineiden internet sekä niin sanotut monikanavaratkaisut.

Ohjelmistorajapinnan rakentamisen askeleet (Biehl 2015: 20):

1. Vaatimusten kartoitus: miten tulevat käyttäjät haluavat käyttää rajapintaa?
2. Sovita rajapinnan toteutus suunnittelevan yrityksen portfolioon sopivaksi, luontevaksi osaksi yrityksen tuotteita/palveluja.
3. Valitse arkkitehtuurityyli, esimerkiksi REST, RPC tai SOAP.
4. Suunnittele rajapinnan rakenne ja rakenna prototyyppi simuloimaan sen toimintaa. Käytä rajapinnan kuvauskieltä, kuten esimerkiksi Swagger:a.
5. Valitse rajapinnan toteutuksessa käytettävät teknologiat ja työkalut.
6. Käytä toteutuksessa generatiivista rajapintametodologiaa, mikäli mahdollista.

2.2 Kerrosarkkitehtuuri

Kerrosarkkitehtuurissa järjestelmä jaetaan abstraktoituihin ohjelmistotasoihin. Tyypillisesti ylin abstraktiotaso on lähimpänä käyttäjää ja voi olla esimerkiksi käyttöliittymä. Alin taso on yleensä laitetaso ja voi kuvata esimerkiksi käyttöjärjestelmää tai sen ajureita. (Koskimies & Mikkonen 2005: 126.)

Kerrosarkkitehtuurin perusajatus on, että ylempi taso käyttää hyväkseen alemman tason tarjoamia palveluja. Tästä periaatteesta on kuitenkin mahdollista poiketa esimerkiksi siten, että kutsu kulkeekin alemmasta tasosta ylempään tai niin, että kutsu ohittaa abstraktiotasoja ylhäältä alas kulkiessaan. (Koskimies & Mikkonen 2005: 126.)

2.3 Plug-in-kehys

Plug-in-kehyksen (plug-in framework) yleisin sovelluskohde on rajapinnan toteutus. Tarkoituksena on pystyä lisäämään rajapintaan sovelluskohtaisia toteutuksia eli laajennusyksiköitä (plug-in). Tällöin rajapinta voidaan erikoistaa toimimaan eri tavoilla tilanteen mukaan. (Koskimies & Mikkonen 2005: 198.)

Plug-in-arkkitehtuurien tyypillisimpiä sovelluskohteita ovat esimerkiksi PC-tietokoneilla käytettävät ohjelmistokehitysympäristöt, joihin on mahdollista ladata käyttäjän toimesta erilaisia laajennoksia. Laajennokset ladataan sovelluksen laajennoskohtaan, joka sisältää valmiin rajapinnan, jonka ladattava laajennos toteuttaa. Kehys eli laajennoskohta ottaa laajennusyksikön käyttöön lataamalla sen sovitusta sijainnista. (Koskimies & Mikkonen 2005: 199.)

Plug-in-kehyksen etuja muihin arkkitehtuurikehyksiin verrattuna ovat sen selkeys ja ketteryys. Laajennosyksiköt helpottavat erikoistuvien toimintojen organisointia arkkitehtuurissa. Modulaarisuus mahdollistaa myös sovelluksen toimintojen laajentamisen ajonaikaisesti. (Koskimies & Mikkonen 2005: 199.)

Plug-in-kehyksen suunnittelussa plug-in-komponenttien valinnaisuus on olennaista. Järjestelmää tulee pystyä käyttämään myös ilman ainuttakaan liitännäiskomponenttia. Plug-in-arkkitehtuuria voidaan soveltaa esimerkiksi seuraavissa tilanteissa (Chatley, Eisenbach & Magee 2003: 1.):

- Järjestelmän toiminnallisuuden laajentaminen
- Suuren järjestelmän hajauttaminen, jotta vain kyseisessä tapauksessa tarvittavaa ohjelmistoa käytetään

- Jatkuvasti käytössä olevan ohjelmiston ajonaikainen päivitys
- Kolmansien osapuolten kehittämien toiminnallisuuksien lisääminen järjestelmään

2.4 REST-arkkitehtuurimalli

REST-arkkitehtuurimalli on HTTP-protokollaan ja asiakas-palvelin-arkkitehtuuriin perustuva verkkopohjaisia rajapintoja varten suunniteltu arkkitehtuurimalli. Sen perusajatus on soveltaa World Wide Web:ssä toimiviksi todettuja ominaisuuksia verkkopalvelujen kehittämiseen. (Biehl 2015: 92; Bass, Clements & Kazman 2013: 109.) REST-malli kehitettiin 1990-luvulla helpottamaan tietoverkkojen rasitusta tietoliikenteen lisääntyessä jatkuvasti. REST-mallin tavoitteena oli minimoida siirrettävän informaation määrä keskittymällä resurssien ilmentymien ja esitysmuotojen siirtämiseen (Fielding & Taylor 2000: 9–10).

REST-mallin periaatteen esitteli Roy Thomas Fielding vuonna 2000 väitöskirjassaan *Architectural Styles and the Design of Network-based Software Architectures*, jossa hän myös loi yleisen määritelmän arkkitehtuurimallin ominaisuuksille. Kyseisen määritelmän mukaan REST on joukko arkkitehtuurisia rajoitteita, joiden tavoitteena on yhtä aikaa minimoida verkkoliikenteen määrä ja siinä esiintyvä viive sekä maksimoida ohjelmistokomponenttien skaalautuvuus ja itsenäisyys toisistaan. (Fielding & Taylor 2000: 1; Fielding 2000: 148.)

REST-malli on hajautettuun hypermediajärjestelmään kuuluvien elementtien abstraktio. Koska REST määrittelee olemassa olevien verkon ominaisuuksien pohjalta arkkitehtuurimallin verkkosovelluksille, voidaan ajatella, että myös World Wide Web on yksi sen ilmentymistä. REST-arkkitehtuurin olennaisena ideana on keskittyä järjestelmän komponenttien toteutuksen ja oikeanlaisen protokollasyntaksin sijaan ymmärtämään niiden keskinäisiä rooleja ja komponenttien välisen kommunikoinnin reunaehtoja. (Fielding & Taylor 2000: 1–3.)

Yksi REST-mallin tärkeimpiä ominaisuuksia on sen ’tilaton’ kommunikaatio järjestelmän komponenttien välillä, mikä osaltaan mahdollistaa komponenttien itsenäisyyden toisistaan. Tilattomuudella tarkoitetaan sitä, että jokaisen asiakaskomponentin palvelimelle tekemän pyynnön tulee sisältää kaikki sen ymmärtämiseen tarvittava tieto. Toimintaperiaatteen ansiosta asiakaskomponentin tekemä pyyntö ei ole riippuvainen palvelimen sisältämästä tiedosta, koska pyyntö voidaan käsitellä itsenäisesti. Kaikki järjestelmän kannalta tarpeelliset tilatiedot säilytetään asiakaskomponentissa. (Fielding 2000: 78–79.)

REST-asiakaskomponentti hyödyntää HTTP-protokollan mukaisesti URI-pohjaista osoitekommunikaatiota yhdistettynä sarjaan ’luo’, ’lue’, ’päivitä’, ’paikkaa’ ja ’poista’ -operaatioita. REST-mallissa nämä operaatiot ovat nimeltään: POST, GET, PUT, PATCH ja DELETE. (Bass, Clements & Kazman 2013: 109.) GET-operaatiota käytetään resurssin tietojen hakemiseen palvelimelta, kun taas POST-operaatiolla luodaan uusi resurssi. PUT-operaatiota käytetään yleensä resurssin päivittämiseen tai korvaamiseen uudella, ja PATCH on sen rajoitetumpi versio, jolla tehdään pienempiä muokkauksia resurssiin. DELETE-operaatio mahdollistaa resurssin poistamisen.

REST-arkkitehtuurissa resurssit pyritään ryhmittelemään kokoelmiksi. Esimerkiksi URI-osoite, joka osoittaa käyttäjien kokoelmaan näyttäisi tältä: *”http://api.esim.com/käyttäjät”*. REST:ssä resurssit kuvataan substantiiveja käyttäen ja resurssien kokoelmat aina monikkomuodossa. Kun halutaan kuvata tiettyyn kokoelmaan kuuluva yksittäinen resurssi, voidaan osoitteessa kokoelman perään lisätä resurssin yksilöivä tunniste: *”http://api.esim.com/käyttäjät/{tunniste}”*. (Restfulapi.net 2020.)

2.5 Malli-näkymä-ohjain-arkkitehtuuri

Malli-näkymä-ohjain-arkkitehtuurissa tärkeimpänä tavoitteena on pyrkiä erottamaan käyttöliittymä varsinaisesta sovelluslogiikasta, jotta järjestelmä pysyy helposti muunneltavana. Arkkitehtuurimallissa järjestelmän osat jaetaan kolmeen kategoriaan: malleihin (model), jotka edustavat jotakin osaa sovellustiedosta; näkymiin (view), jotka edustavat

tiettyä osaa graafisesta käyttöliittymästä; ja ohjaimiin (controller), jotka hallinnoivat tiedonkulkua mallien ja näkymien välillä ja päivittävät muutoksia molempiin. (Koskimies & Mikkonen 2005: 142.)

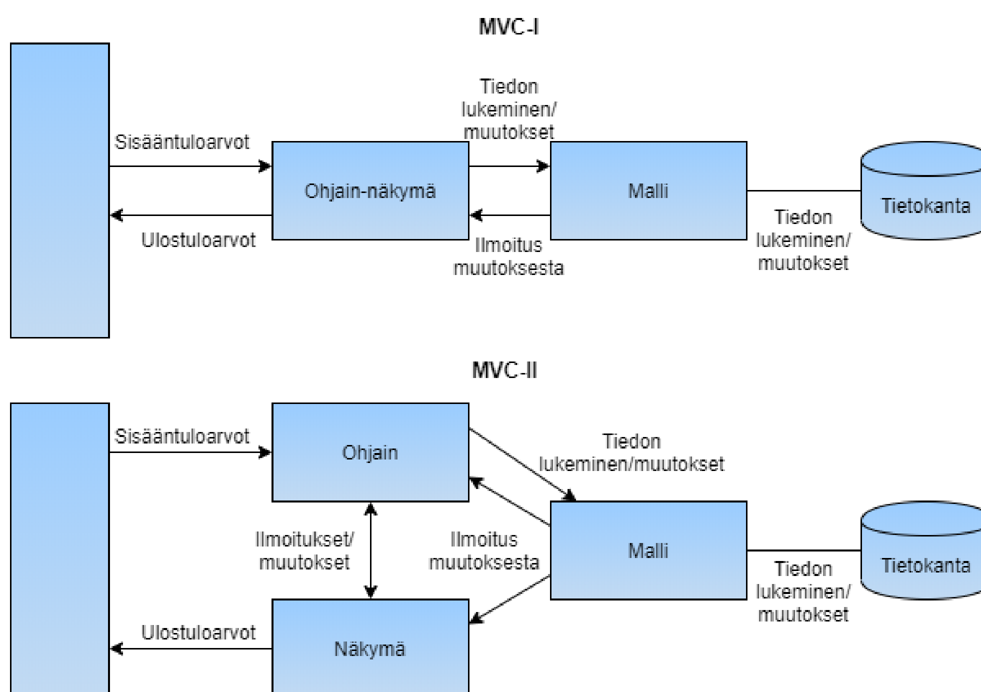
Malli-näkymä-ohjain-arkkitehtuuri esiteltiin ensimmäistä kertaa yleisenä suunnittelukonseptina Krasnerin ja Popen artikkelissa *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System* vuonna 1988 (Qian, Fu, Tao, Xu & Díaz-Herrera 2010: 201). Artikkelissa esitellään MVC-mallin periaate, sen tarjoamat mahdollisuudet sovelluskehityksen näkökulmasta ja tarkka kuvaus komponenttien rooleista.

Krasnerin ja Popen (1988: 2) vision mukaan ideaalisessa tilanteessa kehittäjä loisi ensin sovelluksen mallit eli sovellusdatan ja sen rakenteen, minkä jälkeen suunniteltaisiin käyttöliittymä eli sen eri näkymät, joissa mallin esittelemä tieto näytetään käyttäjälle. Lopuksi luodaan mallien ja näkymien väliset rajapinnat eli ohjaimet, joiden vastuulla on tiedon päivittäminen ja mallien muokkaaminen.

Malli-näkymä-ohjain-arkkitehtuurin vahvuuksia ovat käyttöliittymän helppo muuttaminen tai vaihtaminen uuteen ja malli-luokkien hyvä uudelleenkäytettävyys moniin eri käyttötarkoituksiin (Koskimies & Mikkonen 2005: 143–144). Järjestelmän jakaminen MVC-mallin mukaisiin rooleihin tekee rakenteesta modulaarisen. Tämän ansiosta kehittäjän on helpompi ymmärtää yksittäisen komponentin rakennetta ja pystyä muokkaamaan sitä perehtymättä koko sovelluksen rakenteeseen yksityiskohtaisesti (Krasner & Pope 1988: 2).

Järjestelmämallin heikkoutena voidaan pitää sen mahdollisesti lisäämää monimutkaisuutta arkkitehtuuriin. Riskinä on suuri ohjelmaluokkien määrä ja järjestelmän pirstoutuminen. Niin kutsuttujen tarkkailijoiden (observer) käyttäminen luokkien välisten tilamuutosten tarkkailuun lisää myös päivityskutsujen määrää järjestelmän sisällä ja saattaa ruuhkauttaa viestiliikennettä erityisesti, jos järjestelmä on hyvin pirstoutunut. Toisena suurena heikkoutena voidaan pitää näkymä- ja ohjainluokkien suurta riippuvuussuhdetta toisistaan, minkä takia niitä saattaa olla vaikea käyttää toisistaan irrallisina komponentteina. (Koskimies & Mikkonen 2005: 143–144.)

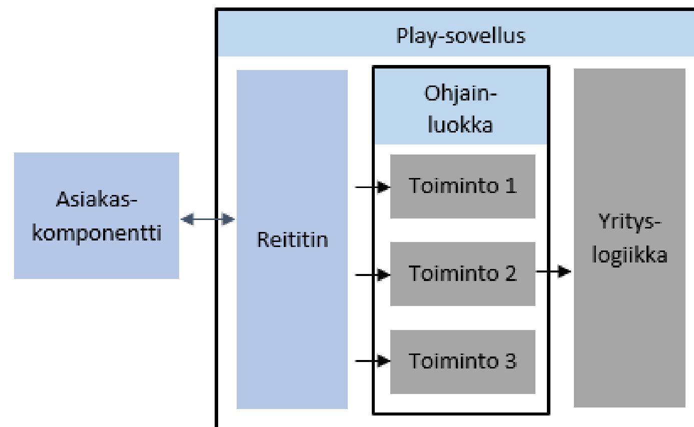
Malli-näkymä-ohjain-arkkitehtuurista on olemassa myös yksinkertaistettu versio, niin sanottu MVC-I, jossa perinteisen MVC-arkkitehtuurimallin eli MVC-II:n ohjain ja näkymä yhdistetään yksittäiseksi komponentiksi, joka huolehtii kaikesta sisään- ja ulostuloinformaation prosessoinnista. Tässä rakenteessa malli hallinnoi informaatiota ja sovelluksen tärkeimpiä toiminnallisuuksia sekä ilmoittaa muutoksista ohjain-näkymälle, joka hallinnoi tehtyjä muutoksia ja toteuttaa kunkin tilanteen mukaiset oikeat toimenpiteet. (Qian, Fu, Tao, Xu & Díaz-Herrera 2010: 202.) Molempien MVC-arkkitehtuurimallien rakenne on havainnollistettu kuvassa 2.



Kuva 2. MVC-I- ja MVC-II-arkkitehtuurimallien tiedonkulku (Perustuu: Qian, Fu, Tao, Xu & Díaz-Herrera 2010: 202, 206; Krasner & Pope 1988: 5).

2.6 Play-ohjelmistokehys

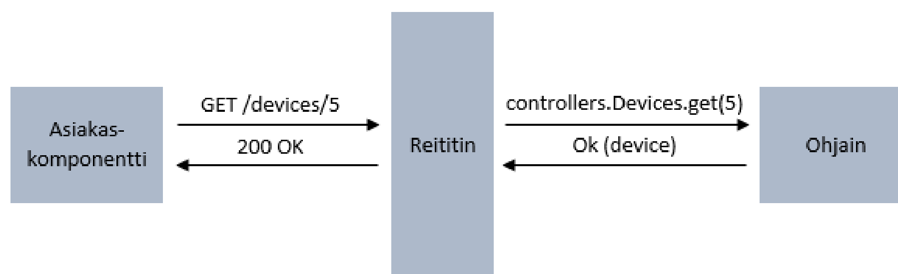
Play on verkkosovellusten rakentamiseen suunniteltu ohjelmistokehys. Sen arkkitehtuuri perustuu asiakas-palvelin-arkkitehtuuriin, joka käyttää HTTP-protokollan mukaisia kutsuja kommunikointiin asiakas- ja palvelinkomponenttien välillä. (Richard-Foy 2014: 23.) Play-sovelluspalvelimen rakenne ja suhde asiakaskomponenttiin on esitettynä kuvassa 3.



Kuva 3. Play-ohjelmistokehyksen kommunikaatioarkkitehtuuri (Perustuu: Richard-Foy 2014: 24).

Play on suunniteltu erityisesti Java- tai Scala-kielillä toteutettavien sovellusten kehittämiseen (Lightbend Inc. 2019a). Ohjelmistokehyksen sisäinen arkkitehtuuri on toteutettu malli-näkymä-ohjain-arkkitehtuurimallin mukaisesti (Lightbend Inc. 2019b). Rakenteensa ansiosta Play -ohjelmistokehys soveltuu hyvin niin sanottujen RESTful-sovellusten eli REST-arkkitehtuurimallin mukaisten verkkosovellusten, kuten rajapintojen, kehittämiseen.

Play-kehyksellä toteutettu sovellus sisältää niin sanotun reitittimen (router), jonka tehtävänä on välittää HTTP-kutsut sovelluksen ohjain-luokan oikeaa toimintoa vastaavaan metodiin (Richard-Foy 2014: 24). Play-sovelluksen reititysperiaate on esitettynä kuvassa 4. Siinä HTTP-kutsu lähtee ensin asiakaskomponentista (GET /devices/5) reitittimeen, joka ohjaa kutsun ja sen välittämät parametrit oikeaan ohjain-luokan toimintoon (controllers.Devices.get(5)). Kun ohjain-luokka on käsitellyt pyynnön, vastaus palautetaan ensin reitittimelle ja sieltä edelleen asiakaskomponenttiin, joka voi palvella esimerkiksi käyttäjän hallinnoimaa graafista käyttöliittymää.

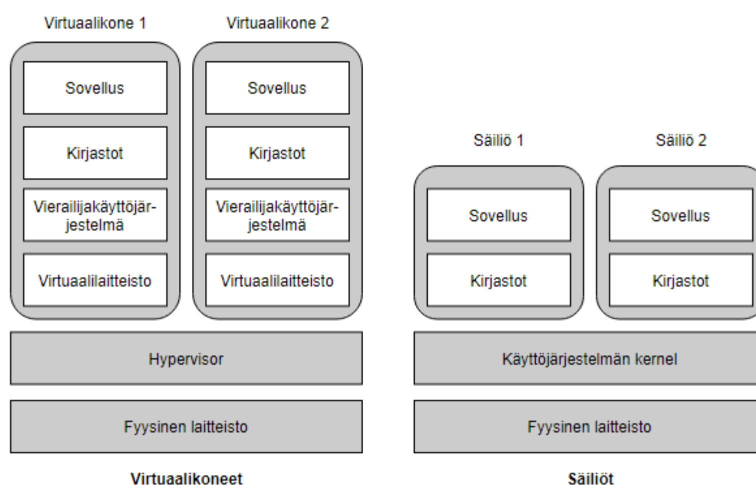


Kuva 4. Play-sovelluksen reititys (Perustuu: Richard-Foy 2014: 31).

2.7 Docker-säiliöinti

Docker on avoimen lähdekoodin moottori, jonka tavoitteena on helpottaa sovellusten asentamista eri ohjelmistoalustoille. Docker hyödyntää säiliöintiä, jossa haluttu ohjelmisto pakataan muusta järjestelmästä eristettyyn säiliöön, joka on eräänlainen kevytrakenteinen virtuaalikone (dotCloud 2019).

Säiliöinnistä käytetään myös nimitystä käyttöjärjestelmätason virtualisointi. Toisin kuin virtuaalikone, joka virtualisoi fyysisen tietokoneen luomalla siitä abstraktin mallin, jonka päälle virtuaalikoneen käyttöjärjestelmä ja muut osat sijoitetaan, säiliöinnissä virtualisoidaan käyttöjärjestelmän ydin, jonka hallinnoimat prosessit eristetään muusta järjestelmästä ja muista säiliöistä. (Chaufournier, Sharma, Shenoy & Tay 2016: 2.) Säiliöinnin erot virtuaalikoneeseen nähden on havainnollistettu kuvassa 5.



Kuva 5. Säiliöinnin toteutus suhteessa virtuaalikoneeseen (Perustuu: Chaufournier, Sharma, Shenoy & Tay 2016: 2).

Säiliöinnin etuna on virtuaalikoneita parempi suorituskyky, jonka säiliöinnin kevyt rakenne mahdollistaa. Koska säiliöt jakavat käyttöjärjestelmän ytimen keskenään, ne eivät toimi yhtä eristettyinä kuin virtuaalikoneet, joilla on tarkat resurssien käytölliset rajat. Jousto resurssien käytössä on eduksi esimerkiksi tilanteissa, joissa järjestelmän säiliöiden kuormitus on epätasainen. (Chaufournier, Sharma, Shenoy & Tay 2016: 11–12.)

Docker-säiliö koostuu kevennetystä käyttöjärjestelmäympäristöstä, jonka päälle rakennetaan kaikki halutut sovellukset ja riippuvuudet, kuten ajoympäristöt, kirjastot ja muut tarvittavat asetukset (Docker Inc. 2019). Haluttu käyttöjärjestelmä ja riippuvuudet sisällytetään komentosarjana Dockerfile-nimiseen tiedostoon. Kun sovellus halutaan ajaa kohdejärjestelmässä, Docker rakentaa komentosarjan perusteella säiliöstä niin sanotun säiliökuvan (container image), joka on säiliön ajossa oleva ilmentymä (Docker Inc. 2019).

Docker:n tarkoitus on tehdä sovelluksen siirtämisestä uudelle järjestelmäalustalle helpompaa säästämällä ohjelmistokehittäjien aikaa käsin tehtävältä riippuvuuksien asentamiselta. Standardoidun rakenteen ansiosta säiliöön pakattu sovellus toimii missä tahansa Docker:n kanssa yhteensopivassa käyttöjärjestelmässä.

Docker hyödyntää versionhallintaa ja kerroksittaista tiedostojärjestelmää, jonka tarkoitus on helpottaa sen käyttöä ketterän kehityksen projekteissa (Chaufournier, Sharma, Shenoy & Tay 2016: 12). Docker mahdollistaa myös ylöspäin skaalaamisen tai kuorman tasaamisen mahdollistamalla esimerkiksi uusien palvelinsovellusten käyttöönoton lyhyellä varoitusajalla (dotCloud 2019).

3 OHJELMISTOTESTAUS

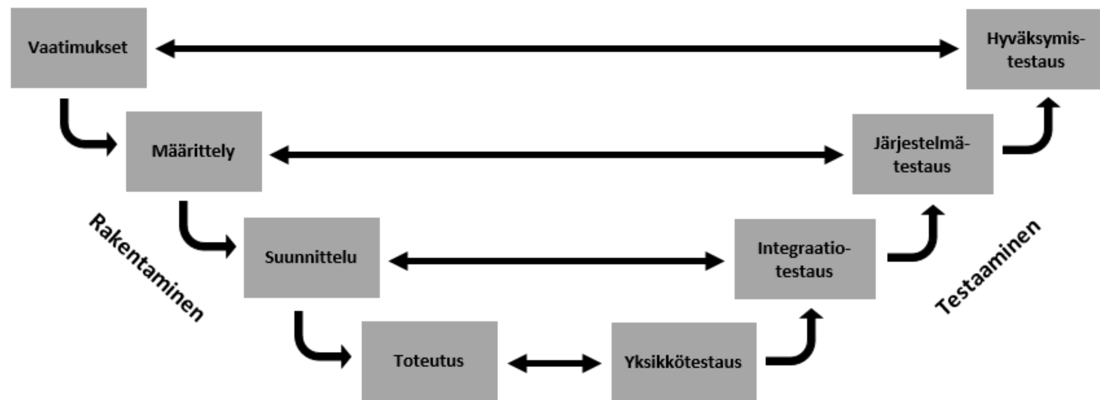
Ohjelmistotestaus voidaan yksinkertaistettuna määritellä suunnitelmalliseksi virheiden etsimiseksi. Sen tärkein tavoite on pyrkiä varmistamaan kehitettävän ohjelmiston laadukkuus. Tämä tarkoittaa sitä, että ohjelmiston ominaisuuksien tulee vastata mahdollisimman hyvin sille asetettuja vaatimuksia. (Haikala & Mikkonen 2011: 205; Kasurinen 2013: 10.)

Vaikka testauksessa pyritään ohjelmiston ominaisuuksien virheettömyyteen, käytännössä tähän tavoitteeseen on kuitenkin mahdotonta päästä. Ohjelmistoa testattaessa on yleensä mahdollista testata vain pieni osa kaikista mahdollisista tapauksista. Tämän vuoksi ohjelman virheetöntä toimivuutta kaikissa tilanteissa on mahdotonta luvata hyvistä tuloksista ja testitapauksista huolimatta. (Haikala & Mikkonen 2011: 205.)

Ohjelmistotestaus on käsitteenä hyvin laaja ja sen pääkohdat voidaan jakaa erilaisiin kategorioihin, joita ovat: testauksen peruskäsitteet, testaustasot, testausmenetelmät, testauksen mittarit, testiprosessi ja testaustyökalut (Bourque & Fairley 2014: 4–2). Tässä tutkimuksessa perehdytään ensisijaisesti testaustasoihin ja testausmenetelmiin.

3.1 Testaustasot

Testaustasoja ja testauksen suhdetta ohjelmistokehitystyöhön voidaan havainnollistaa niin kutsutulla V-mallilla, jossa testauksen kolme päävaihetta eli testaustasoa ovat yksikkötestaus, integraatiotestaus ja järjestelmätestaus. (Kasurinen 2013: 51; Haikala & Mikkonen 2011: 206.) V-malli (kuva 6) perustuu vesiputousmallin mukaiseen tuotantoprosessiin. Sen mukaan jokaisessa vaiheessa suunnitellaan etukäteen kyseisen abstraktiotason testaus. Varsinainen testaus suoritetaan käänteisessä järjestyksessä siten, että ylimmän tason testaus toteutetaan viimeisenä. (Haikala & Mikkonen 2011: 206.)



Kuva 6. Testauksen V-malli (Perustuu: Kasurinen 2013: 51).

Jokaisella testaustasolla testaus tehdään eri ohjelmistotasolla siten, että yksikkötestauksessa testataan ohjelmiston pienimmät osat, kun taas järjestelmätestaus kattaa koko järjestelmän toiminnan kokonaisuutena testiympäristössä. Lopuksi toteutettava hyväksymistestaus tehdään ohjelmiston suunnitellussa kohdeympäristössä ja sen toimintaa verrataan suoraan vaatimuksiin. (Kasurinen 2013: 51.) Testauksessa kulloinkin sovellettavan testaustason määrittää se, mitä testataan ja mikä on testauksen tavoite (Bourque & Fairley 2014: 4–2).

3.1.1 Yksikkötestaus

Yksikkötestaus on yleisin testauksen muoto, jota käytetään yksittäisten moduulien, olioiden tai funktioiden testaamiseen niiden toteutuksen yhteydessä. Yksikkötestauksen tekee yleensä ohjelmoija tai ohjelmistokehittäjä itse ja sen tarkoituksena on varmistaa, että juuri luotu toiminto toimii virheettä ja täyttää sille asetetut vaatimukset. (Kasurinen 2013: 51.)

Koska kehitettävät moduulit ja funktiot luodaan yleensä toimimaan osana suurempaa kokonaisuutta, voidaan toimivan yksikkötestin suorittamiseksi joutua luomaan testipetejä (test bed), joiden tehtävänä on simuloida muun järjestelmän toimintaa suhteessa testattavaan osaan. Testipeteihin sisällytetään testaukseen tarvittavia komponentteja, kuten tes-

tiajureita, tynkämoduuleita ja mock-olioita, joiden tarkoitus on imitoida tai korvata järjestelmässä jo olevia tai siitä mahdollisesti vielä puuttuvia komponentteja. (Haikala & Mikkonen 2011: 207.)

Esimerkkejä mahdollisista yksikkötestauksessa testattavista tilanteista (Kasurinen 2013: 53):

- **Syötettyjen arvojen tyyppi:** Ohjelmalle annettu syöte on eri muodossa kuin on oletettu.
- **Syötettyjen arvojen rajat:** Ohjelmalle annettu syöte, esimerkiksi lukuarvo, on annettujen raja-arvojen ulkopuolella.
- **Käyttäjän syöttämät arvot:** Käyttäjä antaa esimerkiksi liikaa tai liian vähän syötearvoja.
- **Valintarakenteet:** Toimivatko ohjelmassa olevat valintarakenteet oikein? Voivatko kaikki suunnitellut vaihtoehdot toteutua?
- **Komponenttien rajat:** Mitä komponentti tekee, jos viestien vastaanottaminen tai lähettäminen tai toiminnon suorittaminen epäonnistuu?
- **Näkyvyysrajat:** Onko oliorakenne suunniteltu oikein siten, että tarpeellisia toimintoja pystytään kutsumaan ja ne, joihin ei saa suoraan viitata, on piilotettuja.
- **Syntaksivirheet:** Onko luokat, oliot ja muuttujat nimetty oikein. Onko ohjelmakoodissa kirjoitusvirheitä? Onko hyviä ohjelmointikäytäntöjä noudatettu?

3.1.2 Integraatiotestaus

Yksikkötestauksen jälkeen seuraava työvaihe on integraatiotestaus, jossa varmistetaan yksikkötestattujen komponenttien yhteensopivuus. Ohjelmistotuotannossa yksikkötestaus ja integraatiotestaus tavallisesti vuorottelevat siten, että uusi komponentti ensin yksikkötestataan ja sen jälkeen integraatiotestataan osaksi olemassaolevaa järjestelmää. Painopisteenä testauksessa on varmistaa järjestelmän rajapintojen toimivuus. (Kasurinen 2013: 54; Haikala & Mikkonen 2011: 207–208.)

Integroinnissa sovelletaan useimmiten niin sanottua 'bottom up' etenemistapaa, jossa komponenttien integrointi aloitetaan alimmalta tasolta edeten vaiheittain ylemmäs. Muita lähestymistapoja ovat 'top down', jossa edetään edellä mainittuun nähden päinvastaisessa järjestyksessä ja 'sandwich', jossa integraatio etenee molemmista suunnista yhtäaikaisesti ja yhteensovittaminen tapahtuu keskellä. (Kasurinen 2013: 55.)

3.1.3 Järjestelmätestaus

Järjestelmätestauksessa pyritään varmistamaan järjestelmän toimivuus kokonaisuutena, ja testauksen tuloksia verrataan vaatimusmäärittelyyn. Tässä vaiheessa yksikkötestauksessa ja integraatiotestauksessa käytetyt testikomponentit, kuten tyngät ja mock-oliot, poistetaan ja järjestelmää kokeillaan kokonaisena testiympäristössä. Järjestelmätestauksen suorittajan tulisi olla kehitystyöstä riippumaton testaaja. (Kasurinen 2013: 56–57; Haikala & Mikkonen 2011: 208–209.)

Järjestelmätestaus ei edellytä minkään tietyn testaustavan käyttöä vaan työvaiheessa voidaan toteuttaa hyvin monenlaisia toiminnallisia ja ei-toiminnallisia ominaisuuksia testaavia menetelmiä, kuten esimerkiksi käyttäjätestaus, kuormitustestaus, luotettavuustestaus ja niin edelleen (Kasurinen 2013: 56–57; Haikala & Mikkonen 2011: 208–209). Järjestelmätestauksen testitapaukset suoritetaan musta laatikko- ja lasilaatikkotestauksella, koska virheitä voidaan vielä etsiä myös yksittäisistä komponenteista toisin kuin hyväksymistestauksessa (Kasurinen 2013: 57).

3.1.4 Hyväksymistestaus

Hyväksymistestaus on viimeinen V-mallin testaustaso, jossa kehitettävää järjestelmää testataan sen varsinaisessa kohdeympäristössä mieluiten asiakkaan tai loppukäyttäjän toimesta. Järjestelmän ominaisuuksia verrataan sekä alkuperäiseen vaatimusmäärittelyyn että asiakkaan nykyisiin vaatimuksiin. Tavoitteena on osoittaa, että järjestelmä on riittävän korkealaatuinen täyttääkseen vaatimusmäärittelyn mukaiset tarpeet. Onnistuneen hyväksymistestauksen päätteeksi valmis tuote siirtyy asiakkaan omistukseen. (Kasurinen 2013: 57; Badgett, Myers & Sandler 2012: 131.)

3.2 Testausmenetelmät

Ohjelmistotuotannossa voidaan soveltaa erilaisia testausmenetelmiä riippuen siitä, missä tuotannon vaiheessa ollaan. Menetelmät voidaan tämän periaatteen mukaisesti jakaa esituotannon ja kehitysvaiheen aikaisiin sekä ennen julkaisua tehtäviin testausmenetelmiin. (Kasurinen 2013: 4, 62.)

3.2.1 Regressiotestaus

Regressiotestaus tarkoittaa yksinkertaistettuna uudelleentestaamista. Luokituksestaan huolimatta se ei ole oma itsenäinen menetelmänsä eikä sitä tarvitse suorittaa millään tietyllä testaustasolla. Regressiotestaus tarkoittaa yleisesti uudelleentestaamista tilanteessa, jossa jotakin aiemmin testattua järjestelmän osaa muutetaan ja kokonaisuuden toimivuus muutosten jälkeen halutaan varmistaa. (Kasurinen 2013: 68–69.)

Regressiotestauksen perusoletus on, että tehtyjen muutosten jälkeen järjestelmän virheet sijoittuvat todennäköisimmin uusiin komponentteihin tai niitä suoraan hyödyntäviin muihin komponentteihin. Myös ohjelmistotuotannossa osatavoitteeseen pääsemisen yhteydessä tehtävää kaikkien toimintojen tai komponenttien testausta pidetään regressiotestauksena. Tällöin testataan uudelleen myös vanhoissa versioissa olleet toiminnot, jotta varmistutaan, että uudet muutokset eivät ole rikkoneet vanhoja. (Kasurinen 2013: 69.)

3.2.2 Testausautomaatio

Testausautomaatiolla tarkoitetaan testausta, jossa hyödynnetään automaattisia testaustyökaluja toistuvien testitapauksien tekemistä varten. Tavoitteena on vapauttaa testaajat rutiininomaisesta testaustyöstä muihin tehtäviin. Tämä sekä nopeuttaa testausprosessia että vähentää virhealttiutta testaustyössä. (Kasurinen 2013: 76.)
Testausautomaatiota voidaan hyödyntää esimerkiksi kehitystyössä virheiden löytämiseksi järjestelmän moduuleista. Tällöin keskittymiskohteita on yleensä rajapinnat ja moduulien yksikkötestauksen tarkastukset. Muita käyttökohteita on päivittäisversioiden

(daily build) perustestien tekeminen ja käyttöliittymän tarkastukset esimerkiksi toimenpidesarjan toteuttavalla toimintojennauhoitusohjelmalla. (Kasurinen 2013: 76.)

Testausautomaation heikkous on sen vaativuus resurssien suhteen. Automatisoitujen testitapausten rakentaminen on usein yhtä vaativaa kuin varsinaisen ohjelmiston kehittäminen ja vaatii sekä aikaa että resursseja. Kaikissa tapauksissa testausautomaation käyttö ei ole kannattavaa. Testausautomaation käyttö ei myöskään poista tarvetta käsin testaamiselle vaan se on ainoastaan yksi mahdollinen testauksessa hyödynnettävä menetelmä, jonka tarkoitus on vähentää käsin tehtävän testauksen määrää. (Kasurinen 2013: 76–77.) Testausautomaatio soveltuu parhaiten olemassa olevien järjestelmien toimivuuden varmistamiseen (regressiotestaus), kun taas käsin testaaminen soveltuu parhaiten etsimään uusia tapoja olemassa olevien toiminnallisuuksien rikkomiseen (Ramler & Wolfmaier 2006: 88).

Kasurisen, Smolanderin ja Taipaleen (2009: 14) tekemän tutkimuksen mukaan yritysten testiorganisaatiot käyttävät testausautomaatiota vain 26 %:ssa testitapauksistaan, mikä on huomattavasti vähemmän kuin teoreettisten sovellutusten määrä testauksessa. Tulos osoittaa, että testausautomaation soveltaminen käytännössä vaatii yrityksiltä oletettua enemmän resursseja. Lisäksi selkeiden kehitystavoitteiden puute ja tuotesuunnittelun sekä ylläpidon suurempi painottaminen organisaatiossa heikentävät testausautomaation painoarvoa. (Kasurinen, Smolander & Taipale 2009: 14.)

Berner, Keller ja Weber (2005: 574) puolestaan totesivat tutkimuksessaan, että huono testiautomaatiostrategia tai ohjelmistoarkkitehtuuri, jota ei ole suunniteltu ottamaan testattavuutta huomioon, johtaa usein testausautomaation huonoon kustannustehokkuuteen. Sen sijaan liian pieni testitapausten toistomäärä on harvoin esteenä testiautomaation hyödyntämiselle. (Berner, Keller & Weber 2005: 574.)

Mikäli testitapausten odotettavissa oleva toistomäärä on vähintään kymmenen kertaa, tulisi automaation hyödyntämistä harkita. Bernerin, Kellerin ja Weberin (2005: 574) tutkimus osoitti, että lähes kaikki testitapaukset ajetaan vähintään 5–20 kertaa projektin ai-

kana. Heidän mukaansa parhaita kandidaatteja automaation hyödyntämiseen ovat savutestit (smoke test), yksikkötestit ja integraatiotestit, joita usein toteutetaan toistuvasti myös osana regressiotestausta.

Tutkimuksen kokemukset myös osoittivat, että komentosarjapohjaisen testiautomaation soveltamisessa graafisiin käyttöliittymiin tulee noudattaa suurta varovaisuutta. Niiden toteutus ja ylläpito on poikkeuksellisen vaativaa ja johtaa siksi testausmateriaalin heikkoon suunnitteluun. (Berner, Keller & Weber 2005: 574.)

4 TYÖN SUUNNITTELU

Tutkimus suoritettiin ohjelmointityönä ohjelmistoyrityksen tiloissa. Tutkimuksen suunnittelu aloitettiin rajaamalla diplomityön aihe laajuudeltaan sopivan kokoiseksi. Rajauksessa otettiin huomioon sekä toteutettavan järjestelmän tarpeet että yrityksen ja tutkimustyön tekijän toiveet. Rajauksen jälkeen työlle asetettiin tavoiteaikataulu ja laadittiin suunnitelma siitä, minkälaisia lopputuloksia tutkimukselta odotetaan. Tämän jälkeen sovittiin kehitystyön vaiheista ja tavoista, joilla työn etenemistä seurataan. Tutkimussuunnitelmana toimi työn alussa tehty diplomityön alkuraportti.

Perustana työssä tehdyille teknisille ratkaisuille toimivat sekä kirjallisista lähteistä kerätty tieto että suullinen tieto ja osaaminen, joita on saatu ohjelmistoyrityksen työntekijöiltä esimerkiksi palaverien ja kahdenkeskisten keskustelujen kautta. Kirjallisina tietolähteinä tutkimuksessa toimivat ohjelmistoarkkitehtuurin aihe-alueet, kuten ohjelmistorajapinnat, plug-in-kehykset ja REST-arkkitehtuurimalli. Ohjelmistotestauksen suhteen tutkimuksessa perehdyttiin erityisesti testiautomaatioon ja regressiotestaukseen.

4.1 Työvaiheet ja toteutus

Tutkimustyö aloitettiin suunnittelupalavereilla, joissa käytiin läpi rajapinnan tärkeimmät vaatimukset. Suunnittelu tehtiin yhdessä työn ohjaajan ja ohjelmistoyrityksen asiantuntijoiden kanssa. Laitteistorajapinnan vaatimusten perustana toimivat ohjaajan visio järjestelmän halutuista toiminnoista ja rajapintaan ensimmäisenä sovellettavan asiakasprojektin toiminnalliset vaatimukset. Palaverien lopputuloksena suunniteltiin työssä sovellettavat arkkitehtuurimallit ja teknologiat. Samalla annettiin myös suositukset toteutuksessa käytettävistä työkaluista.

Suunnittelun jälkeen perehdyttiin tutkimusaiheen vaatimiin teoria-alueisiin, joita olivat ensisijaisesti toteutuksen kannalta olennaiset arkkitehtuurimallit ja ohjelmistotestauksen aiheet. Teoriaan perehtymisen ohella aloitettiin ohjelmistokehitystyö, jossa tavoitteena

oli edetä ketterän kehityksen menetelmiä noudattaen pyrähdyksinä. Jokaisessa pyrähdyksessä toteutettaisiin laajuudeltaan rajattu osa rajapintaa, kuten esimerkiksi yksittäinen toiminto. Tavoitteena oli laatia jokaisen pyrähdyksen alussa suunnitelma tavoitteista, pitää vähintään viikoittain palaveri, jossa työn edistymistä seurattaisiin, ja lopuksi arvioida pyrähdyksen aikana tehdyt tuotokset ja laatia seuraavan pyrähdyksen suunnitelma.

Työtehtävien hallinnassa hyödynnettiin Kanban-menetelmää, jonka mukaisen työjonon ja sen priorisoinnin hallinnasta vastasi työn ohjaaja. Ominaisuuksien toteutuksessa hyödynnettiin mahdollisuuksien mukaan yrityksen jo olemassa olevia teknologioita ja toteutuksia. Kehitetyt toiminnot yksikkötestattiin ja integraatiotestattiin toteutuksen yhteydessä.

Työn aikana toteutettiin yksi prototyyppiesittely ohjelmistoyrityksen asiakkaalle. Tilaisuudessa esiteltiin testiautomaatiojärjestelmän toimintaa kokonaisuutena, jolloin myös laitteistorajapinnan toiminta ja sen tarjoamat mahdollisuudet esiteltiin.

Kun rajapinnan toiminnalliset vaatimukset oli saatu toteutettua, rajapinnan rakenne ja laatu arvioitiin ohjelmistoyrityksen asiantuntijoiden toimesta. Saadun palautteen pohjalta tehtiin muutamia rakenteellisia muutoksia rajapinnan arkkitehtuuriin. Alustavan hyväksynnän jälkeen siirryttiin järjestelmätestaukseen, jossa järjestelmän toimivuus testattiin simuloidussa ympäristössä. Rajapinta katsottiin valmiiksi, kun se täytti annetut laatuvaatimukset, läpäisi järjestelmätestauksen ja sitä kautta pystyttiin ottamaan käyttöön suunnitellussa kohdeympäristössä.

Käyttöönoton jälkeen työn tulokset ja matkan varrella tehdyt havainnot arvioitiin. Niiden pohjalta muodostettiin tutkimuksen johtopäätökset ja suositukset mahdollisiksi jatkotoimenpiteiksi.

4.2 Asiakasprojektin esittely

Tämän diplomityöprojektin lopputuotteen vastaanottavalla ohjelmistoyrityksellä on teollisuusalalla toimiva asiakas, joka kehittää pääasiallisen liiketoimintansa ohessa teollisuustuotteensa ohjaukseen ja hallintaan tarkoitettua sovellusta. Sovelluksesta on kehitetty omat versiot ainakin kolmelle yleisimmin käytössä olevalle käyttöjärjestelmäalustalle (Android, iOS ja Windows). Sovellus ohjaa teollisuustuotetta kommunikoimalla siihen integroidun sulautetun järjestelmän kanssa, joka välittää komennot tuotteelle. Tähän sulautettuun järjestelmään viitataan työssä joko nimellä *laite* tai *asiakaslaite*.

Asiakaslaitteen kehittyessä ja sovelluksen uusien versioiden julkaisun myötä laitteen ja sovelluksen yhteensopivuus keskenään pyritään säilyttämään johdonmukaisella regressiotestauksella. Omien kustannustensa minimoimiseksi asiakas on ulkoistanut laitteen ja sitä ohjaavan sovelluksen regressiotestauksen ohjelmistoyritykselle, joka loi tarkoitusta varten räätälöidyn järjestelmän. Luodussa järjestelmässä ohjelmistoyritys hyödynsi lisensoitua kolmannen osapuolen kehittämää testiautomaatiojärjestelmää, johon yhdistettiin itse kehitettyjä tapauskohtaisia ominaisuuksia ja asiakkaan omia testausta ja laitehallintaa varten luomia apusovelluksia. Kommunikaatio eri järjestelmien välillä toteutettiin pääasiassa ohjelmistoyrityksen toimesta kolmannen osapuolen palveluiden ja itse kehitettyjen tarkoitukseen sopivien ohjelmien ja komentosarjojen avulla.

Vaikka järjestelmä toimiikin kyseisen projektin edellyttämässä ympäristössä, sen heikkous pitkällä aikavälillä on joustamattomuus muutoksissa. Mikäli testattavien laitteiden määrää lisätään merkittävästi tai niiden ominaisuudet muuttuvat, voidaan koko järjestelmä joutua suunnittelemaan uudestaan. Sama ongelma pätee myös mahdollisiin uusiin asiakkaisiin ja heidän vaatimuksiinsa.

Diplomityön aloittamishetkellä suurin yksittäinen puute ja rajoite järjestelmän toiminnassa oli rajapinta testijärjestelmän ja asiakaslaitteiston välillä. Rajapinta koostui aluksi pääasiassa Java- ja Python-kielisistä ohjelmista ja Windows-komentosarjoista, jotka oli ’kovakoodattu’ antamaan tiettyjä komentoja tietyille tietokoneeseen kytketyille laitteille.

Jotta järjestelmä saataisiin tulevaisuudessa skaalautumaan myös suuremmille laitemäärille, uusille asiakkaille ja uusille vaatimuksille, tulisi järjestelmään kehittää plug-in-arkkitehtuuria hyödyntävä laitteistorajapinta. Plug-in-moduuleista koostuva rajapinta mahdollistaisi tulevaisuudessa uusien asiakaslaitteiden käytön osana järjestelmää, mikäli rajapinnan vaatimat toiminnot toteutettaisiin plug-in-komponentissa tapauskohtaisella tavalla.

4.3 Asiakaslaitteet ja niiden ominaisuudet

Kehitystyön lopputuloksena on tarkoitus kyetä laitteistorajapinnan avulla hallinnoimaan ja antamaan tiettyjä komentoja asiakkaan laitteille, jotka ovat prototyyppivaiheessa olevia sulautettuja järjestelmiä. Laitteita on kahta eri tyyppiä, joihin tässä työssä viitataan nimillä *laite A* ja *laite B*. Ne on suunniteltu toimimaan ohjaimina asiakkaan suunnittelemassa järjestelmässä yhdessä asiakkaan valmistaman tuotteen ja sähkömoottorin kanssa. Järjestelmän toiminnan simuloimiseksi asiakas on myös kehittänyt simulointipiirin *laite C*, joka voidaan kytkeä suoraan laite A:han tai B:hen. Tällöin laitteille voidaan luoda keinotekoisesti sama vaste kuin jos ne olisivat kiinnitettyinä asiakkaan tuotteeseen ja sähkömoottoriin.

Laitteet A ja B ovat sulautettuja järjestelmiä, jotka vastaanottavat komentoja Ethernet-yhteyden välityksellä. Molemmat laitteet tarvitsevat toimiakseen ulkoisen virtalähteen. Asiakkaan järjestelmissä virtalähde on sisäänrakennettuna, mutta koska kehitettävässä testiautomaatiojärjestelmässä ohjainlaitteet toimivat erillisinä yksiköinä, virransaanti ratkaistiin kytkemällä laitteet virransaantia ohjaavaan releohjainpiiriin, jonka avulla laitteiden virransaantia pystyttiin ohjaamaan rajapinnasta. Releen kytkeminen päälle tai pois oli siten myös ainut tapa käynnistää tai sammuttaa laite.

Molemmissa laitetyypeissä käytetään kahta ohjelmistotyyppiä: laitteen oma laiteohjelmisto ja kommunikointisovellus, joka vastaanottaa komentoja käyttäjän mobiililaitteelleen tai tietokoneelleen asentamasta sovelluksesta. Laiteohjelmisto on laitteessa A erilai-

nen kuin laitteessa B, mutta kommunikointisovellus on molemmissa identtinen. Kommunikointisovellus voidaan päivittää laitteelle lataamalla se tarkoitusta varten suunniteltua ohjelmaa käyttäen tietokoneelta Ethernet-yhteyden välityksellä. Laiteohjelmiston päivitykseen tarvitaan tarkoitusta varten suunnitellut komentosarjat ja sarjaporttiyhteys laitteeseen.

Yksi työn haasteista oli se, että laitteilla ei ole järjestelmätasolla mitään yksilöllistä tunnistetta, jonka perusteella voitaisiin varmentaa, mikä laite on kyseessä. Ainut tunniste, jonka perusteella laitetta voidaan etsiä, on sen IP-osoite, joka voidaan kuitenkin tietyllä komennolla vaihtaa uuteen. Mahdollisuus vaihtaa IP-osoitetta kuitenkin helpottaa laitteiden käyttöä järjestelmässä, koska kaikkien laitteiden oletus-IP-osoite on alussa sama.

5 RAJAPINNAN TOTEUTUS

Ohjelmistokehitystyö kesti noin yhdeksän kuukautta, minkä aikana ketteriä menetelmiä sovellettiin muun muassa pitämällä vähintään kerran viikossa tapaaminen (weekly meeting), jossa käytiin läpi työn edistymistä. Alkuperäisestä suunnitelmasta poiketen työn tekemisessä ei sovellettu Scrumin mukaisia pyrähdyksiä, vaan työtehtävät toteutettiin viikoittain päivitettävää Kanban-työlistaa käyttäen. Scrumin seuraaminen osoittautui käytännössä haastavaksi tiimin pienuuden, työn ohjaajan kiireiden ja kesälomakauden takia.

Työn tekemisessä käytettiin Java-ohjelmointikieltä ja Play-ohjelmistokehystä IntelliJ IDEA -ohjelmointiympäristössä. Play perustuu MVC-arkkitehtuuriin, jota voidaan tästä syystä pitää laitteistorajapinnan arkkitehtuurisena lähtökohtana. Play asetti myös suuntaviivat toimintojen tekniselle toteutustavalle.

Jokaisen uuden ominaisuuden toteutuksen yhteydessä laadittiin tarvittavat yksikkötestitapaukset, joilla ohjelmakoodin laatu varmistettiin. Rajapinnan toimintojen yksikkötestauksessa käytettiin Java-yksikkötestaukseen suunniteltua JUnit-ohjelmistokehystä. Luotujen toimintojen integraatiotestaus puolestaan toteutettiin Postman-rajapintatestaustyökalulla.

Muita työssä käytettyjä apuvälineitä olivat muun muassa Swagger, jota käytettiin rajapinnan dokumentointiin, ja Drawio-mallinnusohjelmisto, jolla mallinnettiin rakenteita ja luotiin työssä tarvittuja kuvaajia.

Laitteistorajapinnan kanssa samanaikaisesti suunniteltiin testiautomaatiojärjestelmälle myös mobiilirajapintaa, joka mahdollistaa testien ajamisen mobiililaitteilla, ja graafista käyttöliittymää, joka hyödyntää kehitettävien rajapintojen tarjoamia toimintoja.

5.1 Vaatimusten määrittely

Rajapinnan vaatimusten hahmottelemiseksi pidettiin työn alussa palavereita ohjelmistoyrityksen toimesta. Palavereissa tehtiin alustava suunnitelma toiminnoista ja ominaisuuksista, jotka suunniteltavan rajapinnan tulee toteuttaa.

Järjestelmän ensisijainen ja tärkein tavoite on tuotteistaa sovelluksen ja sen kanssa käytettävän laitteen regressiotestaus digitaaliseksi palveluksi, jota ohjelmistoyritys pystyisi markkinoimaan asiakkailleen. Ohjelmistot ja sovellukset, joiden toimintaa ja kommunikaatiota testataan yhdessä fyysisen laitteen kanssa (esimerkiksi sulautettu järjestelmä), edellyttävät usein yksilöllisiä ratkaisuja, jotka on luotu kyseistä testiympäristöä varten.

Kaikkien asiakaslaitteiden osalta tavoitteena on pystyä automatisoidusti valmistelemaan laitteet sovelluksen regressiotestausta varten. Ongelmana kuitenkin on, että jokainen asiakaslaitetyyppi vaatii erilaisen kommunikointiprotokollan. Loogisin tapa toteuttaa ehdot täyttävä rajapinta on hyödyntää modulaarista rajapinta-arkkitehtuuria, joka mahdollistaa asiakaskohtaisten plug-in-komponenttien lisäämisen järjestelmään.

Koska asiakas saattaa haluta käyttää omia laitteitaan fyysisesti erillään varsinaisesta DPC-järjestelmästä esimerkiksi omalla toimipisteellään, täytyy testiautomaatiojärjestelmän projektikohtainen rajapinta rakentaa toimimaan erillisellä palvelimella, joka voidaan tarvittaessa sijoittaa fyysisesti erilleen muusta järjestelmästä. Projektikohtainen rajapinta tulee pystyä yhdistämään internet-yhteyden avulla helposti muuhun järjestelmään. (Filander 2019a.)

Koska asiakasprojektista riippuen järjestelmän kautta hallittavat laitteet voivat olla ominaisuuksiltaan erilaisia, tulee myös järjestelmän arkkitehtuurissa ottaa vaihtelevat ominaisuudet huomioon. Projektin alussa tehdyn vaatimusmäärittelyn mukaan laitteistorajapinnan kautta tulee olla mahdollista nähdä, onko laite varattu/vapaa tai valmis/ei-valmis testaukseen, riippumatta laitteen ominaisuuksista (Filander 2019b). Lisäksi rajapinnan tulee mahdollistaa ainakin kytkettynä olevien ja vapaiden laitteiden hakeminen (Filander 2019b).

Alustava esitys toiminnoista, jotka laitteistorajapinnan tulee toteuttaa (Filander 2019b):

- Discover – järjestelmään kytkettynä olevien laitteiden tarkistaminen
- Connect – laitteen yhdistäminen järjestelmään
- PowerUp – laitteen käynnistäminen
- ShutDown – laitteen sammuttaminen
- Restart – laitteen uudelleenkäynnistys
- Update – järjestelmäpäivityksen asentaminen laitteen muistiin
- IsReady – laite valmiina testaukseen
- Wait – laitteen sijoittaminen jonoon odottamaan testausta
- IsFree – laite on vapaa eli sitä ei ole varattu mihinkään testiin
- Reserve – laitteen varaaminen testaukseen
- Release – laitteen vapauttaminen testauksesta

Mikäli testiautomaatiojärjestelmän käyttäjä haluaa tietoa tietyn laitteen ominaisuuksista, järjestelmä suorittaa kyselyn projektikohtaiselle rajapinnalle, joka palauttaa pyydetyn laitteen tiedot, mikäli käyttäjällä on siihen oikeus. Yleisten ominaisuuksien lisäksi laitteilla voisi olla myös laitekohtaisia ominaisuuksia, joita rajapinnan käyttäjä voisi myös halutessaan hyödyntää.

Laitteistorajapinnan ei-toiminnalliset vaatimukset:

- Arkkitehtuuri mahdollistaa useiden rinnakkaisten projektirajapintojen olemassaolon.
- Kaikkien projektirajapintojen tulee toteuttaa yleisen rajapinnan määrittelemät laitteistorajapinnan toiminnot.
- Rajapinnan abstraktiotasojen tulee pystyä toimimaan erillisillä palvelimilla, jotka kommunikoivat internet-yhteyden välityksellä.

Laitteistorajapinnan toiminnalliset vaatimukset:

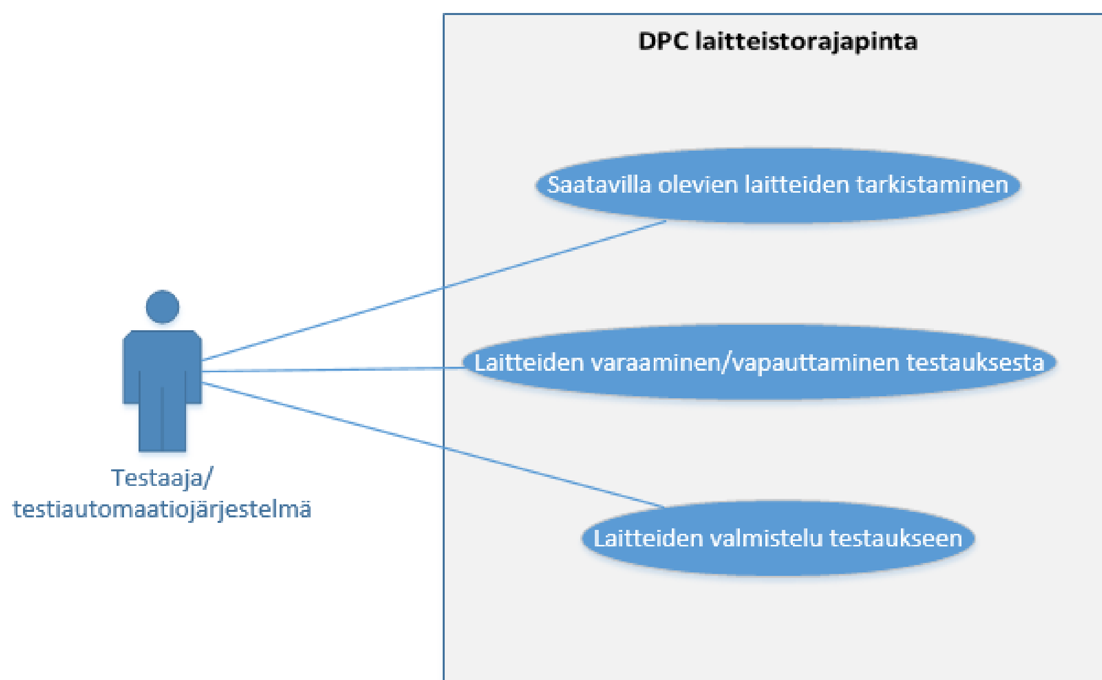
- Testiautomaatiojärjestelmän tulee mahdollistaa kaikkien asiakaslaitteiden osalta:
  - Mahdollisuus nähdä, onko laite vapaa vai varattu testaukseen.
  - Mahdollisuus nähdä, onko laite valmisteltu testaukseen vai ei ja halutessa valmistella se laitekohtaisella tavalla.
  - Mahdollisuus hakea järjestelmään kytkettyjä laitteita tai vapaita laitteita.

- Kehitettävän projektikohtaisen rajapinnan tulee lisäksi toteuttaa nykyistä asiakasprojektia varten kehitetyn järjestelmän toiminnot, jotka ovat:
  - Laitteen ohjelmiston päivittäminen
  - Laitteen käynnistys, sammutus ja uudelleenkäynnistys
  - Laitteen IP-osoitteen muuttaminen

Käyttäjätarinat ja käyttötapaukset:

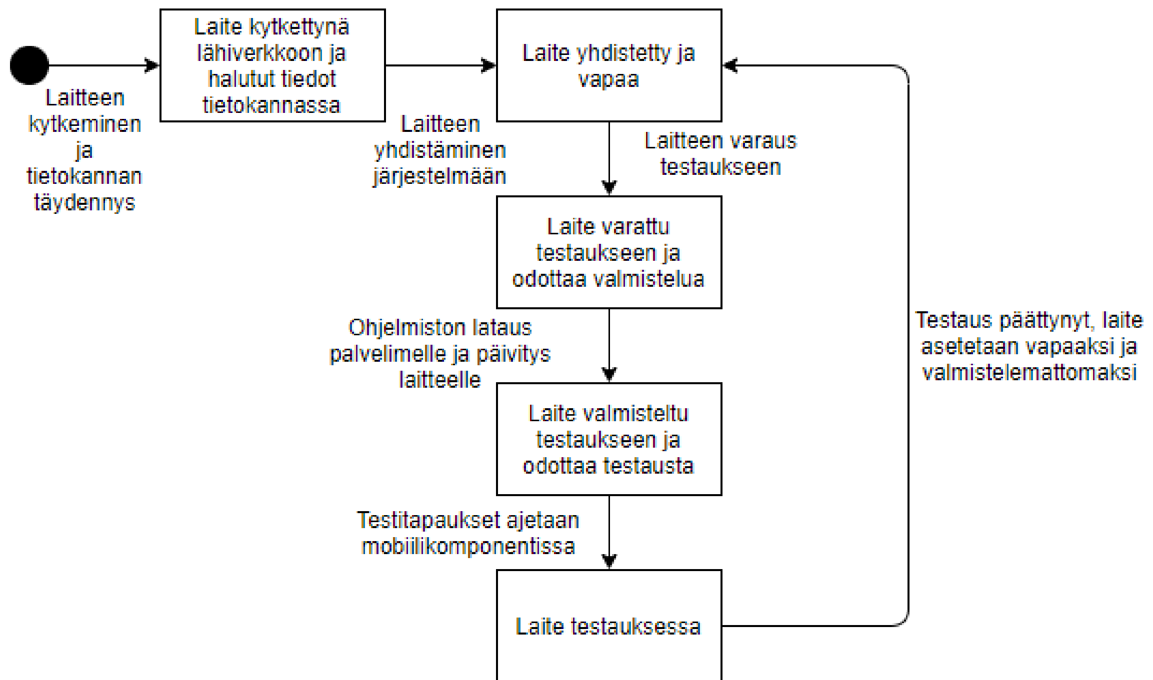
Järjestelmän toiminnallisuuden ymmärtämiseksi annettujen rajapintakomentojen pohjalta laadittiin lista käyttäjätarinoista ja eri käyttötapauksista, joita rajapinnan käyttäjällä olisi. Suunnitelmassa päädyttiin kolmeen käyttäjätarinaan (alla), joiden pohjalta myös vastaavat käyttötapaukset määräytyivät (kuva 7).

1. DPC-laitteistorajapinnan käyttäjänä haluan nähdä kaikki testausta varten saatavilla olevat laitteet ja niiden ominaisuudet.
2. DPC-laitteistorajapinnan käyttäjänä haluan pystyä varaamaan haluamani laitteet testaukseen.
3. DPC-laitteistorajapinnan käyttäjänä haluan pystyä automaattisesti päivittämään oikean ohjelmiston varatuille laitteille testausta varten.



Kuva 7. Testiautomaatiojärjestelmän laitteistorajapinnan käyttötapauskaavio.

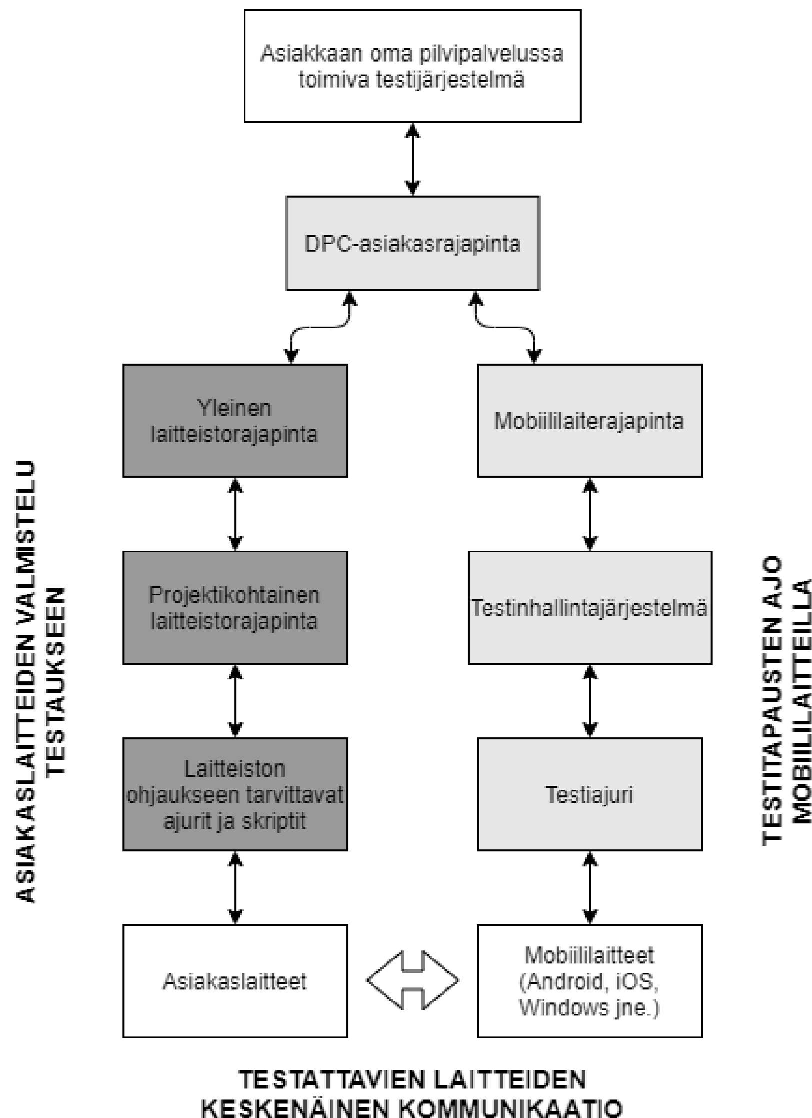
Käyttötapauskaavion lisäksi laitteen eri tiloja kuvaamaan laadittiin tilakaavio (kuva 8), joka havainnollistaa laitteen mahdolliset tilat testiautomaatiojärjestelmässä.



Kuva 8. Tilakaavio asiakaslaitteen mahdollisista tiloista testauksen eri vaiheissa.

5.2 Ohjelmistoarkkitehtuurin suunnittelu

Laitteistorajapinnan keskeisimpänä tavoitteena on mahdollistaa tiettyjen komentojen välittäminen järjestelmään kytketyille laitteille niiden erilaisista teknisistä ominaisuuksista huolimatta. Helpoin tapa toteuttaa rajapinnan joustavuus muutoksissa on hyödyntää plug-in-kehystä arkkitehtuurissa. Tällöin rajapinta koostuu kahdesta erillisestä abstraktiotasosta, joille annetaan tässä työssä nimet *yleinen rajapinta* ja *projektikohtainen rajapinta*. Kuvassa 9 on hahmotelma testiautomaatiojärjestelmän suunnitellusta rakenteesta. Kuvassa näkyvät laitteistorajapinnan komponentit (tummanharmaalla värillä) ja niiden sijoittelu suhteessa muihin DPC-testiautomaatiojärjestelmän komponentteihin (vaaleanharmaalla värillä).



Kuva 9. DPC-testiautomaatiojärjestelmän komponentit (Perustuu osittain: Filander 2019c).

Yleinen rajapinta pysyy testiautomaatiojärjestelmässä muuttumattomana. Sen tehtävänä on vastaanottaa ja palauttaa tietoa testiautomaatiojärjestelmälle fyysisiltä laitteilta. Yleisestä rajapinnasta toteutetaan vain yksi ilmentymä, joka sijoitetaan testiautomaatiojärjestelmän palvelimelle osaksi muuta järjestelmää. Projektikohtainen rajapinta toimii rajapinnan plug-in-komponenttina, josta voidaan toteuttaa useita ilmentymiä. Lähtökohtaisesti jokaisen ilmentymän tulee toteuttaa yleisen rajapinnan toiminnot erilaisilla, projektikohtaisilla tavoilla, jotka ovat riippuvaisia siihen kytkettyjen laitteiden teknisistä ominaisuuksista. Toteutettuaan yleisen rajapinnan toiminnot, projektikohtaisen rajapinnan tulee

myös palauttaa kyselyjen vastaukset oikeassa, yleisen rajapinnan edellyttämässä muodossa takaisin testiautomaatiojärjestelmälle.

Yleisten laiteominaisuuksien lisäksi projektikohtainen rajapinta voi myös käsitellä ja palauttaa tietoja projektikohtaisista ominaisuuksista. Näiden osalta yleisen rajapinnan ei tarvitse osata prosessoida laitekohtaista tietoa vaan toimia välittäjänä projektikohtaisen rajapinnan ja käyttäjän välillä.

Jotta projektikohtainen rajapinta voidaan eriyttää fyysisesti muusta järjestelmästä, rajapinnan abstraktiotasot tulee rakentaa erillisille, osittain toisistaan riippumattomille palvelimille. Kommunikaatio komponenttien välillä voidaan toteuttaa verkkosovellusarkkitehtuuria, kuten esimerkiksi HTTP:hen perustuvaa REST-arkkitehtuuria hyödyntämällä. REST:n käytön etuna on paitsi sen pienet suorituskyvylliset vaatimukset, myös sen yleisyys verkkopohjaisessa kommunikaatiossa. REST-rajapintakutsumallin seuraaminen helpottaa myös integraatiota osaksi laajempaa testiautomaatiojärjestelmäkokonaisuutta.

Toisistaan fyysisesti erilliset järjestelmän komponentit ja abstraktiotasot edellyttävät myös erillisten, tasokohtaisten tietokantojen käyttämistä. Tämä tarkoittaa sitä, että kun käyttäjä haluaa tiedon esimerkiksi tietyssä testiprojektissa käytettävien laitteiden määrästä ja ominaisuuksista, järjestelmä hakee tiedon projektikohtaiselta logiikkatasolta. Testiautomaatiojärjestelmä ei toisin sanoen tallenna laitetietoja omalle palvelimelleen. Tämä mahdollistaa laajennettavuuden ja tekee järjestelmästä modulaarisen helpottaen muutosten tekemistä tulevaisuudessa.

Koska työ keskittyy erityisesti rajapinnan kehittämiseen asiakaslaitteiden ja testiautomaatiojärjestelmän välille, ei varsinaista MVC-mallin mukaista näkymä-luokkaa tarvitse sisällyttää arkkitehtuuriin. Siten rajapinnan arkkitehtuurin voidaan sanoa vastaavan MVC-I-arkkitehtuuria (ks. 2.5 & kuva 2), jossa ohjain ja näkymä sulautetaan yhtenäiseksi komponentiksi.

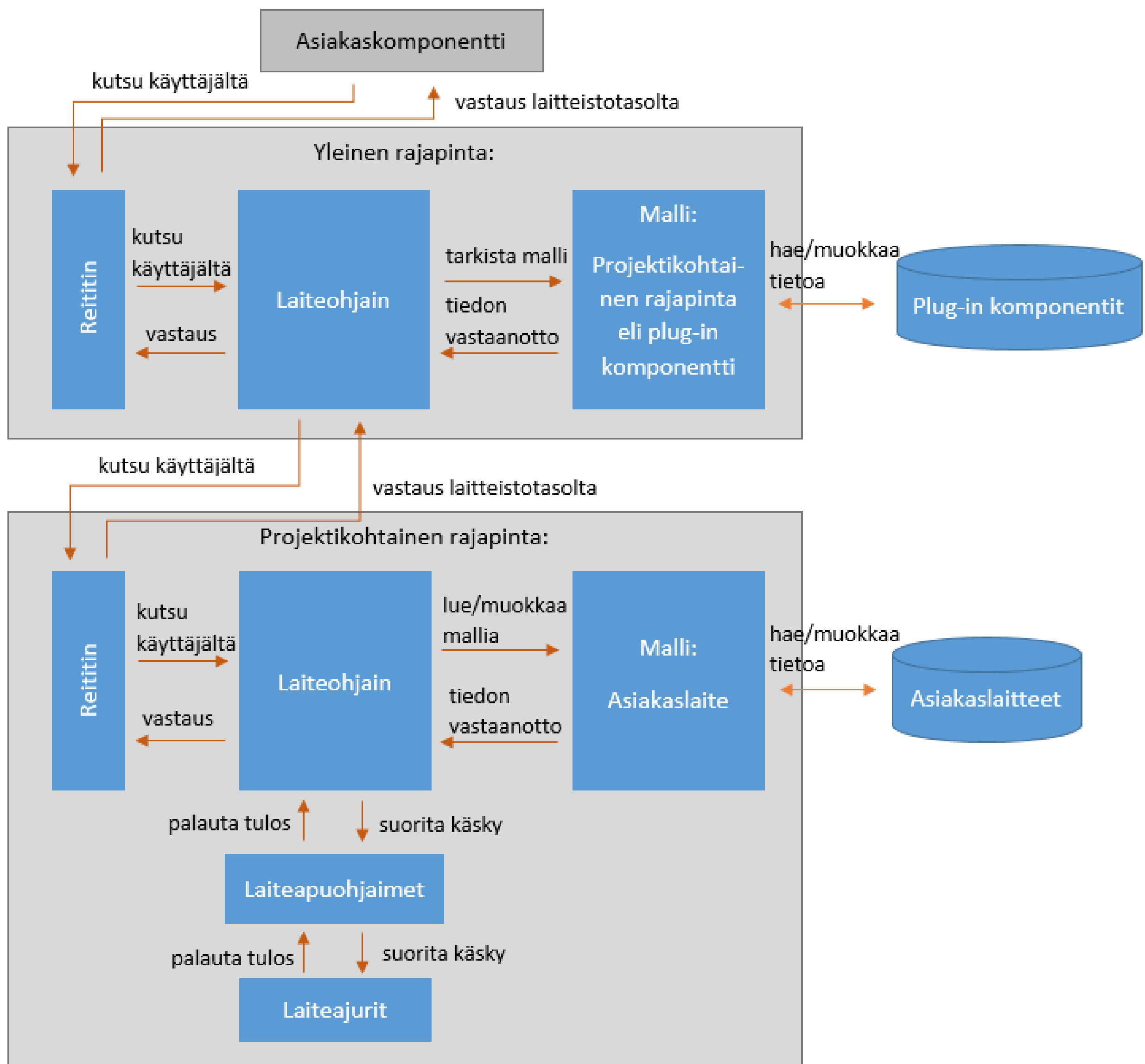
Tiivistetysti esitettynä laitteistorajapinta koostuu MVC-I-arkkitehtuurilla rakennetusta yleisestä rajapinnasta ja sen toteuttavasta plug-in-komponentissa toimivasta vastaavalla

rakenteella tehdystä projektikohtaisesta rajapinnasta. Molemmissa ohjain-näkymä-luokat toimivat varsinaisina tietoa välittävinä rajapintoina. Komponenttien välinen kommunikointi ja resurssien käsittely noudattavat REST-mallin yleisiä periaatteita.

5.3 Projektipohjan luominen

Koska järjestelmän tärkeimpiä vaatimuksia ovat laajennettavuus ja hajautettavuus aloitettiin rajapinnan kehitystyö luomalla kaksi erillistä Play-ohjelmistokehyksen ja MVC-arkkitehtuurin toteuttavaa sovellusprojektia, joista ensimmäiseen rakennettiin yleinen rajapinta ja toiseen projektikohtainen rajapinta. Molempien sovellusprojektien tuli toteuttaa samat toiminnot ja siksi kummankin ohjain-luokan tuli toteuttaa toisiaan vastaavat abstraktit rajapintaluokat. Yleisen rajapinnan sisältävä projekti toteuttaa toiminnot siten, että se välittää asiakaskomponentin pyynnöt projektikohtaisen rajapinnan sisältävälle projektille, odottaa paluuviestinä saatavaa vastausta ja välittää sen takaisin asiakaskomponentille.

Molemmista projektipohjista poistettiin MVC-mallin mukaiset näkymäluokat, koska rajapinnalle ei ole tarvetta suunnitella erillistä graafista käyttöliittymää. Käyttäjäsyötteet välitetään rajapintaan Play-ohjelmistokehyksen reitittimen kautta. Varsinainen käyttäjänäkymä suunnitellaan tulevaisuudessa osaksi laitteistorajapintaa hyödyntävää testiautomaatiojärjestelmää. Rajapintatasojen arkkitehtuuri ja komponenttien välinen kommunikaatio on esitetty kuvassa 10.



Kuva 10. Laitteistorajapinnan arkkitehtuurimalli.

Yleisen rajapinnan sisäisessä arkkitehtuurissa HTTP-kutsut välitetään reitittimeltä *Laite-ohjain*-nimiselle sovelluksen ohjainluokalle, joka välittää tietoa sekä kyseisen abstraktiotason mallin että alemman tason komponenttien välillä. Laiteohjain tarkistaa kyseisen malliluokan olemassaolon tietokannasta ja välittää sitten tiedon eteenpäin.

Yleisen rajapinnan malliluokkana toimii tässä tapauksessa projektikohtaisen rajapinnan ilmentymä eli plug-in-komponentti. Kun malliluokan olemassaolo on tarkistettu, ohjain

hyödyntää sen osoitetietoja ja välittää käyttäjäkutsun malliluokan edustamalle projektikohtaiselle rajapinnalle hyödyntäen Play-ohjelmistokehyksen WS-kirjastoa, joka mahdollistaa palvelimien väliset epäsynkroniset HTTP-kutsut (Lightbend Inc. 2020a).

Projektikohtaisessa rajapinnassa kutsut välitetään reitittimeltä laiteohjain-luokalle, joka toteuttaa kaikki yleisen rajapinnan metodit. Ohjain käsittelee kutsun ja tarpeen mukaan hakee tietoa, muokkaa mallia tai välittää komennon alemman tason komponenteille. Projektikohtaisessa rajapinnassa malli kuvaa asiakaslaitetta. Laiteohjaimen lisäksi projektikohtaiseen rajapintaan sisältyvät myös niin sanotut laiteapuohjain-luokat, jotka suorittavat laiteohjaimen alemman abstraktiotason toimintoja, kuten esimerkiksi välittävät käskyjä laiteajureille.

Molemmille rajapintaprojekteille luotiin omat toisistaan riippumattomat tietokannat, joihin tallennettiin rajapinnan mallit. Koska rajapinnat toimivat erillään, yleinen rajapinta pääsee käsiksi projektikohtaisiin laitetietoihin ainoastaan suorittamalla kyselyn oikeaan projektikohtaiseen rajapintaan tietokannasta löytyvien tietojen perusteella.

5.4 Rajapinnan ominaisuuksien toteuttaminen

Kun projektipohjat molemmille rajapintakomponenteille oli luotu, alkoi vaatimusmäärittelyn mukaisten toimintojen toteuttaminen. Testattavuuden helpottamiseksi jokainen rajapintaan lisätty toiminto toteutettiin ensin projektikohtaiseen rajapintaan ja sen jälkeen yleiseen rajapintaan. Tällöin jokainen toiminto voitiin heti integraatiotestata asiakaslaitteen kanssa ja tunnistaa mahdolliset toteutusteknisen haasteet aikaisessa vaiheessa.

Kun toiminto oli toteutettu projektikohtaiseen rajapintaan, sen toimivuus testattiin yksikkötestauksella ja integraatiotestauksella. Kun uusi toiminto toimi ja se täytti annetut vaatimukset, luotiin vastaava välittäjänä toimiva kutsu yleiseen rajapintaan. Myös sen toimivuus varmistettiin vastaavasti yksikkötestauksen ja integraatiotestauksen avulla.

Toteutettavien ominaisuuksien priorisoinnista huolehti tuotteen omistajana toiminut työn ohjaaja. Priorisointi tehtiin helpoin-ensin ja tärkein-ensin -periaatteita noudattaen. Rajapinnan toimintojen toteutusjärjestys ja lyhyet kuvaukset niiden ominaisuuksista löytyvät taulukosta 1. Toimintojen yksityiskohtaiset kuvaukset löytyvät liitteestä 3.

Taulukko 1. Laitteistorajapinnan toiminnot listattuna toteutusjärjestyksessä.

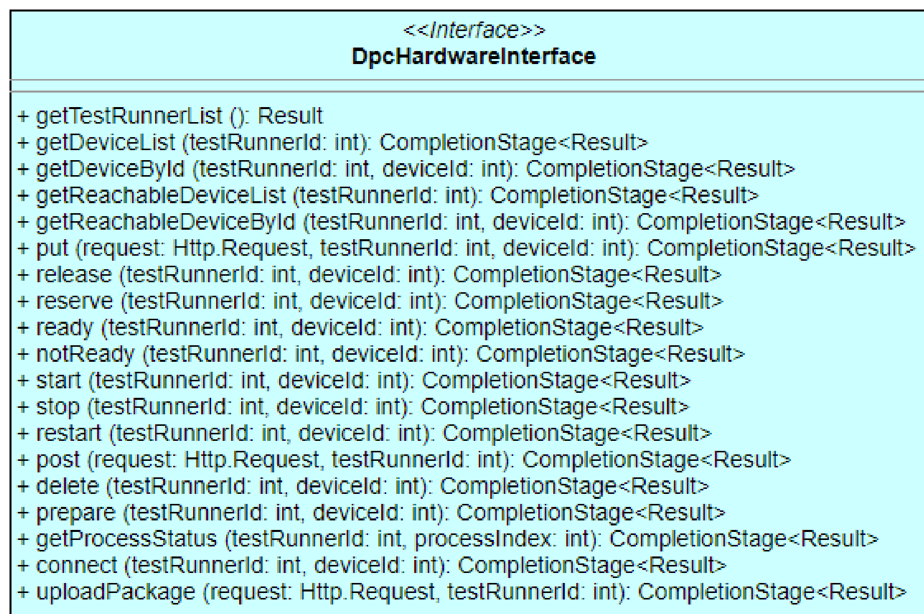
Järjestys-numero	Toiminnallisuus	Kuvaus
1	Hae tietokannassa olevia laitteita tai plug-in-komponentteja	Rajapinnan malliluokkien haku molemmista abstraktiotasoista. Sekä mallilistan että yksittäisten mallien haku tunnisteen perusteella mahdollista.
2	Hae saatavilla olevia laitteita	Testiautomaatiojärjestelmän lähiverkkoon kytkettynä olevien laitteiden hakeminen IP-osoitteen perusteella.
3	Muokkaa laitteen tietoja	Laitteen kaikkien ominaisuuksien vapaa muokkaaminen.
4	Muuta laitteen varaustilaa tai valmiustilaa	Laitteen yksittäisten tila-arvojen muuttaminen. Mahdollistaa laitteen varaamisen tai vapauttamisen ja asettamisen valmiiksi testaukseen tai takaisin 'epävalmiiksi'.
5	Laitteen virranhallinta	Laitteen sammuttaminen, käynnistäminen tai uudelleenkäynnistäminen rajapinnan kautta.
6	Lisää tai poista laite	Laitteen lisääminen tietokantaan tai poistaminen tietokannasta rajapintakutsulla.
7	Päivitä laitteen ohjelmisto ja seuraa päivitysprosessin etenemistä	Laitteen sisäisen ohjelmiston päivitys ja päivitysprosessin seuraaminen rajapinnasta prosessitunnisteen avulla.
8	Vaihda laitteen IP-osoite	Vaihtaa laitteen IP-osoitteen tietokantaan tallennetun osoitteen mukaiseksi. Sisällytettiin myöhemmin osaksi toimintoa nro 9.
9	Valmistele laite testaukseen	Toiminnon nro 7 paranneltu versio, joka suorittaa laitetyypin mukaisen päivityssarjan, joka asentaa molemmat halutut ohjelmistot laitteeseen.
10	Yhdistä laite testiautomaatiojärjestelmään automaattisesti	Toiminnon nro 8 paranneltu versio. Tarkistaa, onko laitteen IP-osoite oletusarvossa ja vaihtaa tarvittaessa tilalle tietokannasta löytyvän IP-osoitteen.
11	Lataa uusi ohjelmistopaketti	Uuden laitteelle asennettavan ohjelmistopaketin lataaminen laitteistorajapintaan.

Taulukossa listatut toiminnot kuvaavat joko tiettyä yksittäistä rajapinnan mahdollistamaa rajapintakomentoa (toiminnot 2, 3, 8–11) tai niiden sarjaa (toiminnot 1, 4–7). Tämän li-

säksi toimintojen suhde rajapintakutsuihin muuttui laadun arvioinnin jälkeen jonkin verran. Lopullisessa rajapinnan toteutuksessa osa tässä listatuista toiminnoista jakautuu useampaan rajapintakutsuun ja osa toiminnoista on koottu yhteen kutsuun.

Viimeisen toiminnon lisäämisen jälkeen rajapinnan rakennetta siistittiin poistamalla ylimääräisiksi arvioituja toimintoja tai ominaisuuksia, jotka oli rajapinnan kehittyessä havaittu tarpeettomiksi. Esimerkiksi erilliset ohjelmistojen päivitystoiminnot jätettiin pois, koska ne eivät toteuta yleisen rajapinnan päivitystoiminnon vaatimuksia. Yleisen rajapinnan ei voida olettaa huomioivan laitekohtaisia ominaisuuksia. Sen sijaan tarkoitus on hoitaa *laitteen valmistelu testaukseen* -toiminnon sisällä kaikki tarvittavat projektikohtaiset työvaiheet. Myös IP-osoitteen vaihtamiseen tarkoitettu metodi ja *laitteen virranhallinta* -toimintoon sisältynyt virransaannin tilan tarkistamiseen tarkoitettu metodi jätettiin pois, koska molempien tarjoama toiminnallisuus voidaan toteuttaa osana muita rajapintakutsuja.

Rajapinnan ensimmäisen version toiminnot toteuttavat metodit on esitetty kuvan 11 rajapintaluokassa. Metodien nimien lisäksi luokasta näkyvät myös metodien sisääntulomuuttujat ja niiden tyypit sekä palautettavien ulostulojen tyypit.



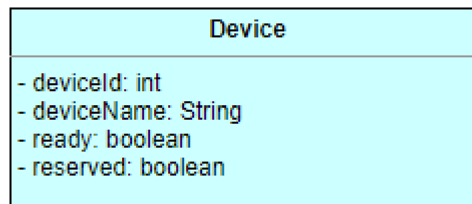
Kuva 11. Yleisen rajapinnan ensimmäisen version abstrakti rajapintaluokka.

Rajapinnan mahdollistamat HTTP-kutsut on puolestaan listattu taulukkoon 2. Jokaista taulukon 2 rajapintakutsua vastaa kuvan 11 mukainen metodi, joka käsittelee asiakaskomponentin antamat syötteet. Toiminnot on esitetty kuvassa ja taulukossa samassa järjestyksessä.

Taulukko 2. Yleisen rajapinnan ensimmäisen version HTTP-kutsut.

Tyyppi	URI	Kuvaus
GET	/test-runners	Näyttää listan kaikista testiautomaatiojärjestelmän projektikohtaisista rajapinnoista.
GET	/test-runners/{testRunnerId}/devices	Näyttää listan kaikista projektikohtaisen rajapinnan laitteista.
GET	/test-runners/{testRunnerId}/devices/{deviceId}	Näyttää laitetunnisteen mukaisen laitteen kaikki tiedot.
GET	/test-runners/{testRunnerId}/devices/reachable	Näyttää listan kaikista lähiverkkoon kytketyistä laitteista.
GET	/test-runners/{testRunnerId}/devices/{deviceId}/reachable	Näyttää, onko laitetunnisteen mukainen laite kytkettynä lähiverkkoon.
PUT	/test-runners/{testRunnerId}/devices/{deviceId}	Laitetietojen muokkaus laitetunnisteen perusteella.
GET	/test-runners/{testRunnerId}/devices/{deviceId}/release	Vapauttaa laitteen testauksesta laitetunnisteen perusteella.
GET	/test-runners/{testRunnerId}/devices/{deviceId}/reserve	Varaa laitteen testaukseen laitetunnisteen perusteella.
GET	/test-runners/{testRunnerId}/devices/{deviceId}/ready	Asettaa laitteen valmis-tilaan valmistelun jälkeen (prepare).
GET	/test-runners/{testRunnerId}/devices/{deviceId}/not-ready	Asettaa laitteen ei-valmis-tilaan testauksen jälkeen.
GET	/test-runners/{testRunnerId}/devices/{deviceId}/start	Käynnistää laitteen laitetunnisteen perusteella.
GET	/test-runners/{testRunnerId}/devices/{deviceId}/stop	Sammuttaa laitteen laitetunnisteen perusteella.
GET	/test-runners/{testRunnerId}/devices/{deviceId}/restart	Uudelleenkäynnistää laitteen laitetunnisteen perusteella.
POST	/test-runners/{testRunnerId}/devices	Lisää uuden laitteen projektikohtaiselle palvelimelle.
DELETE	/test-runners/{testRunnerId}/devices/{deviceId}	Poistaa laitteen palvelimelta laitetunnisteen perusteella.
GET	/test-runners/{testRunnerId}/devices/{deviceId}/prepare	Valmistelee laitteen testausta varten päivittämällä sen ohjelmiston.
GET	/test-runners/{testRunnerId}/process-status/{processId}	Näyttää prosessin tilan annetun prosessitunnistenumeron perusteella.
GET	/test-runners/{testRunnerId}/devices/{deviceId}/connect	Yhdistää laitteen testiautomaatiojärjestelmään.
POST	/test-runners/{testRunnerId}/upload-file/{fileId}	Lataa projektikohtaiselle palvelimelle uuden päivityspaketin päivitystunnisteella.

Testiautomaatiojärjestelmän toimivuuden kannalta laitteistorajapinnan laitteilla tulee olla tietyt yhteiset ominaisuudet. Nämä ominaisuudet näkyvät kuvan 12 asiakaslaite-malliluokassa. Laitteistorajapinnan asiakaslaitteen tulee siis sisältää ainakin tunniste, nimi, valmiustila ja varaustila-attribuutit. Nämä ominaisuudet mahdollistavat laitteen yksilöinnin ja seurannan testiautomaatiojärjestelmässä.



Kuva 12. Asiakaslaitemallin yleiset ominaisuudet laitteistorajapinnan ensimmäisessä versiossa.

5.5 Yksikkö- ja integraatiotestaus

Laitteistorajapinnan komponenttien yksikkötestauksessa käytettiin Javalle suunniteltua JUnit-yksikkötestausohjelmistokehystä. Projektiin luotiin oma yksikkötestipaketti, jonne testitapaukset tallennettiin. Jokaiselle projektin luokalle luotiin oma testiluokka, jonka sisältämät testitapaukset kävivät läpi luokan sisällä olevat metodit.

Testiluokkien lisäksi testikansioon luotiin myös mock-luokkia simuloimaan testattavan komponentin kanssa vuorovaikutuksessa olevia komponentteja. Esimerkiksi komponenttien toiminnan testaus virhetilanteissa testattiin luomalla virheellistä tai poikkeavaa tietoa testattavalle komponentille välittävä mock-luokka.

Yksikkötestien tulosten vertailu toteutettiin JUnit-kehyksen Assert-metodilla. Esimerkiksi: ”assertEquals(viesti, odotettu_ulostulo, todellinen_ulostulo)”, joka vertaa saatua tulosta käyttäjän antamaan odotettuun tulokseen sekä ”assertTrue(ehto)” ja ”assertFalse(ehto)”, jotka vertaavat saatua tulosta määrättyyn totuusarvoon.

Yksikkötestimetodin tarkoitus on palauttaa testin tulos totuusarvona. Mikäli testi palauttaa myönteisen totuusarvon, palautettava arvo vastaa kehittäjän odotuksia ja testi on onnistunut. Jokaisesta testattavasta komponentista pyrittiin luomaan vähintään yksi virhekoodin palauttava ja yksi onnistuva testitapaus.

Mock-luokkien luomisessa käytettiin apuna Play-ohjelmistokehyksen Guice-moduulia ja niin kutsuttua riippuvuusinjektiota (dependency injection), jossa komponentteihin lisättävien sidosmääreiden avulla voidaan kuvata kaikki kyseisen komponentin luomiseen tarvittavat osat. Tämä nopeuttaa simuloitujen komponenttien luomista ja samalla pienentää inhimillisen virheen riskiä testauksessa. (Lightbend Inc. 2020b.)

Yksittäisten komponenttien lisäksi toteutuksen aikana suoritettiin myös rajapinnan toimintojen integraatiotestaus, jossa testattiin, vastasivatko tietynlaisen rajapintakutsun paluuarvot odotuksia. Testit suoritettiin käsin, listattuja testitapauksia noudattaen. Useimmissa testitapauksissa käytettiin myös asiakaslaitteita rajapintaan kytkettyinä.

5.6 Laitteistorajapinnan käyttö Docker-säiliössä

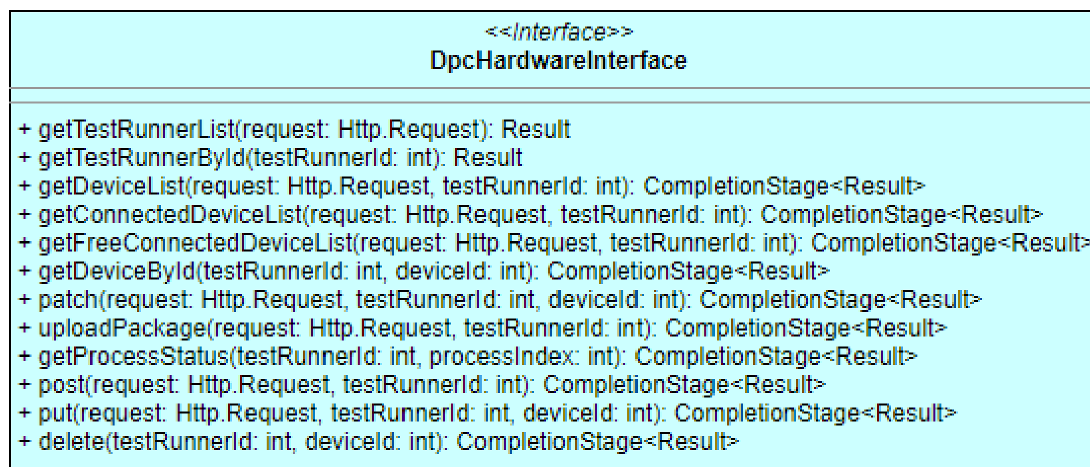
Tutkielman ohjaajan aloitteesta haluttiin työn alkuvaiheessa tutkia rajapinnan käyttöä Docker-virtuaaliympäristössä, koska se mahdollistaisi testiautomaatiojärjestelmän komponenttien helpon asennuksen uusille palvelinalustoille. Työn aikana sekä yleinen laitteistorajapinta että projektikohtainen rajapinta integroitiin toimimaan Docker-säiliön sisällä siten, että rajapintojen tietokannat toimivat kontin ulkopuolella.

Projektikohtaisen rajapinnan sisältämät riippuvuudet osoittautuivat ongelmaksi Docker:n käytössä. Koska Docker-säiliö käyttää oletusarvoisesti Linux-käyttöjärjestelmän karsittua versiota, se ei mahdollista monia käyttöjärjestelmän kehittyneempien ominaisuuksien tai Windows-käyttöjärjestelmäriippuvaisten ohjelmien käyttöä. Myöskään tietokoneen fyysisten yhteyksien, kuten USB-liitinten, käyttö ei ole mahdollista säiliöidyssä ympäristössä.

Koska sekä Windows-käyttöjärjestelmäriippuvuudet että tietokoneen fyysiset yhteydet ovat tutkimuksessa toteutetun projektikohtaisen rajapinnan kannalta välttämättömiä ei projektikohtaisen rajapinnan ajaminen Docker-säiliössä ole mahdollista. Lisäksi on syytä olettaa, että suurin osa myös tulevaisuudessa järjestelmään liitettävistä projektikohtaisista rajapinnoista sisältää vastaavanlaisia riippuvuuksia. Näin ollen säiliöintiä ei voida lähtökohtaisesti pitää kannattavana myös muillekaan projektikohtaisille rajapintatoteutuksille. Yleisen rajapinnan ajaminen säiliössä sen sijaan on teknisesti mahdollista, mutta todennäköisesti ei tarpeellista, koska siitä ei arkkitehtuurisuunnitelman mukaisesti ole tarvetta luoda useita ilmentymiä kuten projektikohtaisesta rajapinnasta.

5.7 Laadun arviointi

Kun vaatimusmäärittelyn mukaiset toiminnallisuudet oli toteutettu rajapintaan, arvioitiin toteutetun arkkitehtuurin laatu vertaamalla rajapintakutsujen toteutusta REST arkkitehtuurin yleisiin periaatteisiin. REST-mallin mukaista suunnitelmaa seuraamalla arkkitehtuuri voidaan pitää helppolukuisena ja mahdollistaa rajapinnan helppo sovellettavuus tuleviin projekteihin. Myös yleinen yhteensopivuus muiden REST rajapintakomponenttien kanssa helpottuu. Rajapinnan laadun arvioinnin suoritti yrityksen asiantuntija. Muutosten tavoitteena oli entistä resurssikeskeisempi lähestyminen rajapinnan esitykseen. Arvioinnin jälkeen rajapinnasta luotiin uusi versio, jonka rakenne muokattiin vastaamaan paremmin REST-arkkitehtuurimallia. Yleisen rajapinnan uuden version abstrakti rajapintaluokka on esitetty kuvassa 13.



Kuva 13. Yleisen rajapinnan abstrakti rajapintaluokka tehtyjen muutosten jälkeen.

Tärkeimpänä muutoksena rajapintaan toteutettiin PATCH-kutsu, joka korvasi vanhan rajapinnan varaustilan ja valmiustilan asettamiseen sekä virranhallintaan tarkoitetut kutsut. Myös laitteen yhdistäminen ja valmistelu testaukseen toteutetaan parannuksien myötä PATCH-kutsulla, joka muokkaa tarkoituksen mukaisia resurssiattribuutteja. Käyttäjän antamat muutokset hyväksytään vain, jos kyseinen muutettava kenttä on projektikohtaisessa rajapinnassa erikseen sallittu. Tuntemattomat kentät rajapinta jättää huomiotta. Tämä mahdollistaa eteenpäin yhteensopivuuden, koska rajapinta ei palauta virhettä esimerkiksi tilanteessa, jossa asiakaskomponentti on päivitetty uudempaan versioon.

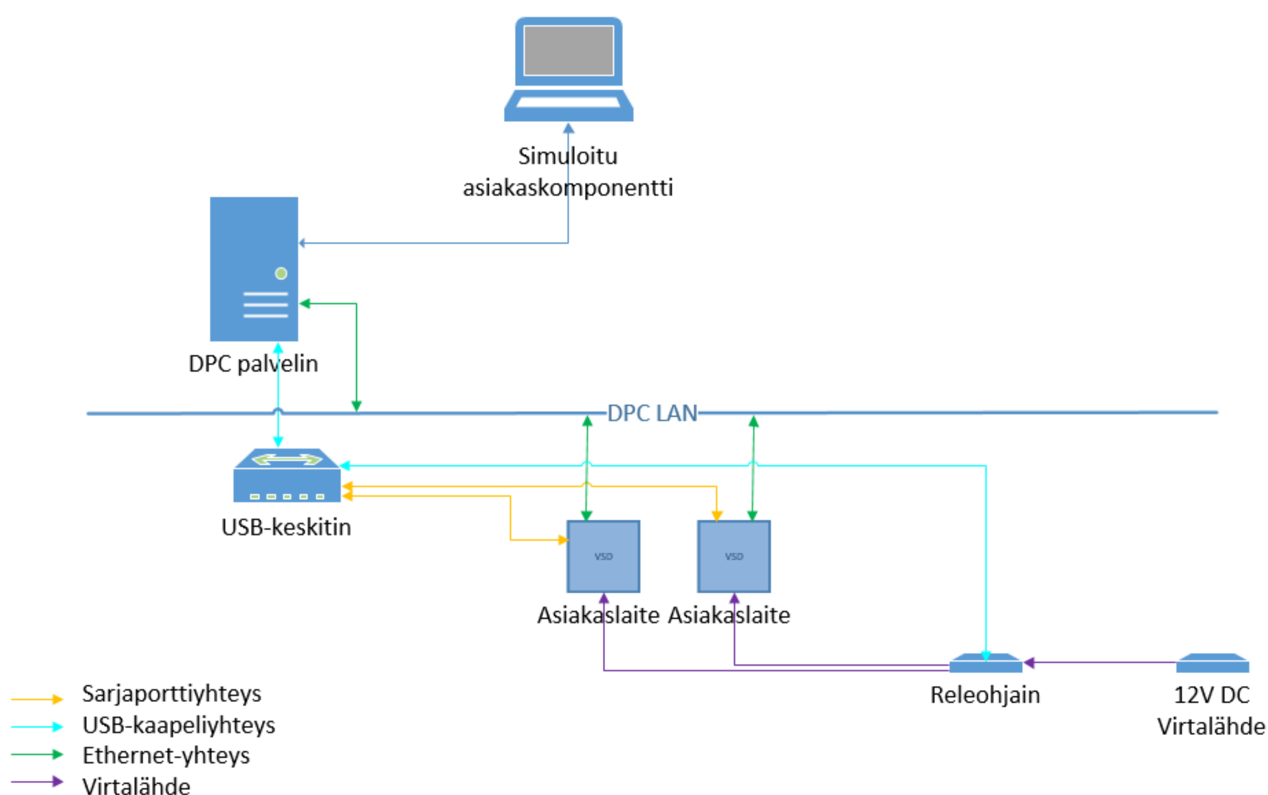
REST-mallin mukaisesti asiakaslaitteiden hakeminen tiettyjen kriteerien perusteella tulisi suorittaa antamalla kaikkien laitteiden listan hakevalle metodille hakukriteereitä vastaava yksi tai useampi kyselyparametri. Käytännön toteutuksen helpottamiseksi ja nopeuttamiseksi rajapinnan ensimmäiseen versioon haku kuitenkin toteutettiin luomalla kullekin etukäteen tiedossa olevalle yleiselle hakurajaukselle oma, niin sanottu kovakoodattu metodinsa.

REST-mallin mukaisesti yleiseen rajapintaan lisättiin kutsu, jolla voidaan hakea yksittäisen projektikohtaisen rajapinnan tiedot sen tunnisteluvun perusteella. Laitteen lisäämisen, poistamisen ja muokkaamisen mahdollistavat metodit säilytettiin rajapinnassa, vaikka

niiden käyttö vaatimusmäärittelyn mukaisissa testaustilanteissa ei ole tarpeellista. Toiminnot voidaan kuitenkin sisällyttää rajapintaan osana järjestelmävalvojan oikeuksiin sisältyviä toimintoja. Lisäksi yksittäisen laitteen saatavuustiedon palauttava kutsu poistettiin, koska sama tieto saadaan myös laitteen kaikki tiedot hakemalla.

5.8 Järjestelmätestaus

Laitteistorajapinnan paranteluvaiheen jälkeen sen komponentit asennettiin testiympäristönä toimivalle palvelintietokoneelle järjestelmätestauksen suorittamiseksi (kuva 14). Laitteistorajapintaan kytkettiin kaksi asiakaslaitetta (yksi laite A ja yksi laite B), joiden alustamista testaukseen kokeiltiin simuloidun asiakaskomponentin avulla.



Kuva 14. Laitteistorajapinnan testiympäristö järjestelmätestauksessa (Perustuu: Nguyen 2019).

Testitapauksia laadittaessa havaittiin, että paras tapa niiden kuvaamiseen on sekvenssikaavio, jossa suunniteltu komponenttien välinen kommunikaatio kuvataan siinä sillä tavalla, kuin sen odotetaan tapahtuvan. Kaikki testitapaukset laadittiin sekvenssikaavioina, jotka kuvasivat asiakaskomponenttina toimivan testiautomaatiojärjestelmän kutsut laitteistorajapinnalle ja sen takaisin asiakaskomponentille antamat vasteet. Järjestelmätestaukseen valittiin viisi testitapausta:

1. Onnistunut asiakaslaitteen valmistelu (laite B).
2. Onnistunut asiakaslaitteen valmistelu (laite A).
3. Valmistelu epäonnistuu: laiteohjelmiston päivitys epäonnistuu, koska asiakaskomponentti lataa vääränlaisen tiedoston rajapintaan (laite A).
4. Valmistelu epäonnistuu: asiakaskomponentti yrittää ladata tiedoston rajapintaan, mutta rajapintakutsu ei sisällä tiedostoa (laite A).
5. Valmistelu epäonnistuu: asiakaskomponentti yrittää aloittaa valmistelun, mutta päivitystiedostot puuttuvat rajapinnasta (laite B).

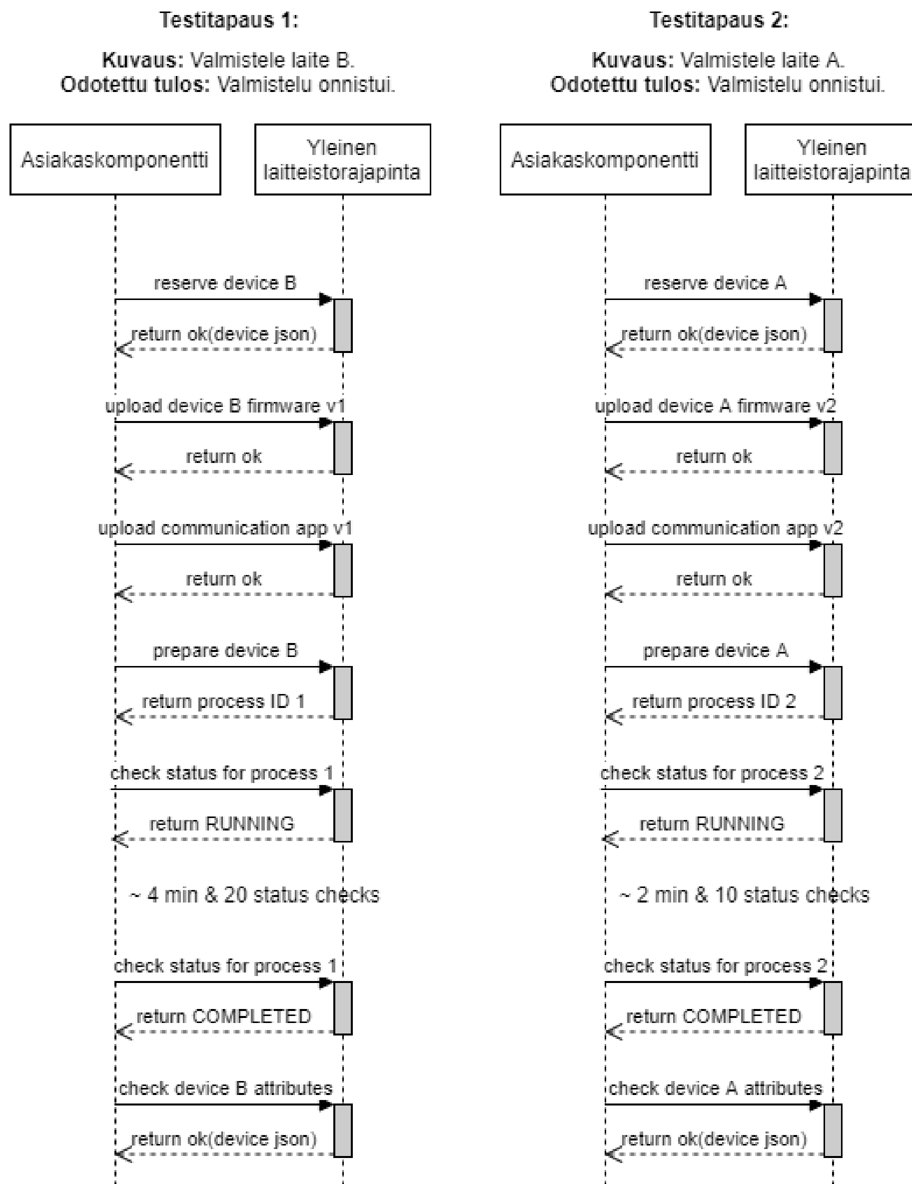
Testitapauksiin haluttiin sisällyttää kaksi onnistunutta testitapausta (yksi kummallakin testilaitteella). Tämän lisäksi haluttiin testata muutama tapaus, joissa prosessi ei etene odotetulla tavalla tai rajapintaa kutsutaan virheellisesti.

Testitapauksista kaksi ensimmäistä testasivat järjestelmän toiminnan normaaliolosuhteissa. Kolmannessa testissä laitteelle A yritetään ladata laitteen B laiteohjelmisto. Neljännessä testissä tiedoston lataus rajapintaan on virheellinen ja viidennessä yritetään käynnistää laitteen valmistelu, vaikka rajapinta ei sisällä päivitystiedostoja.

Testeissä laitetta B testattiin laiteohjelmistojen ensimmäisillä versioilla, joiden mukaan laitteistorajapinta on suunniteltu. Laitetta A testattiin puolestaan uudemmalla versiolla, jossa käytetään hieman erilaista tiedostorakennetta.

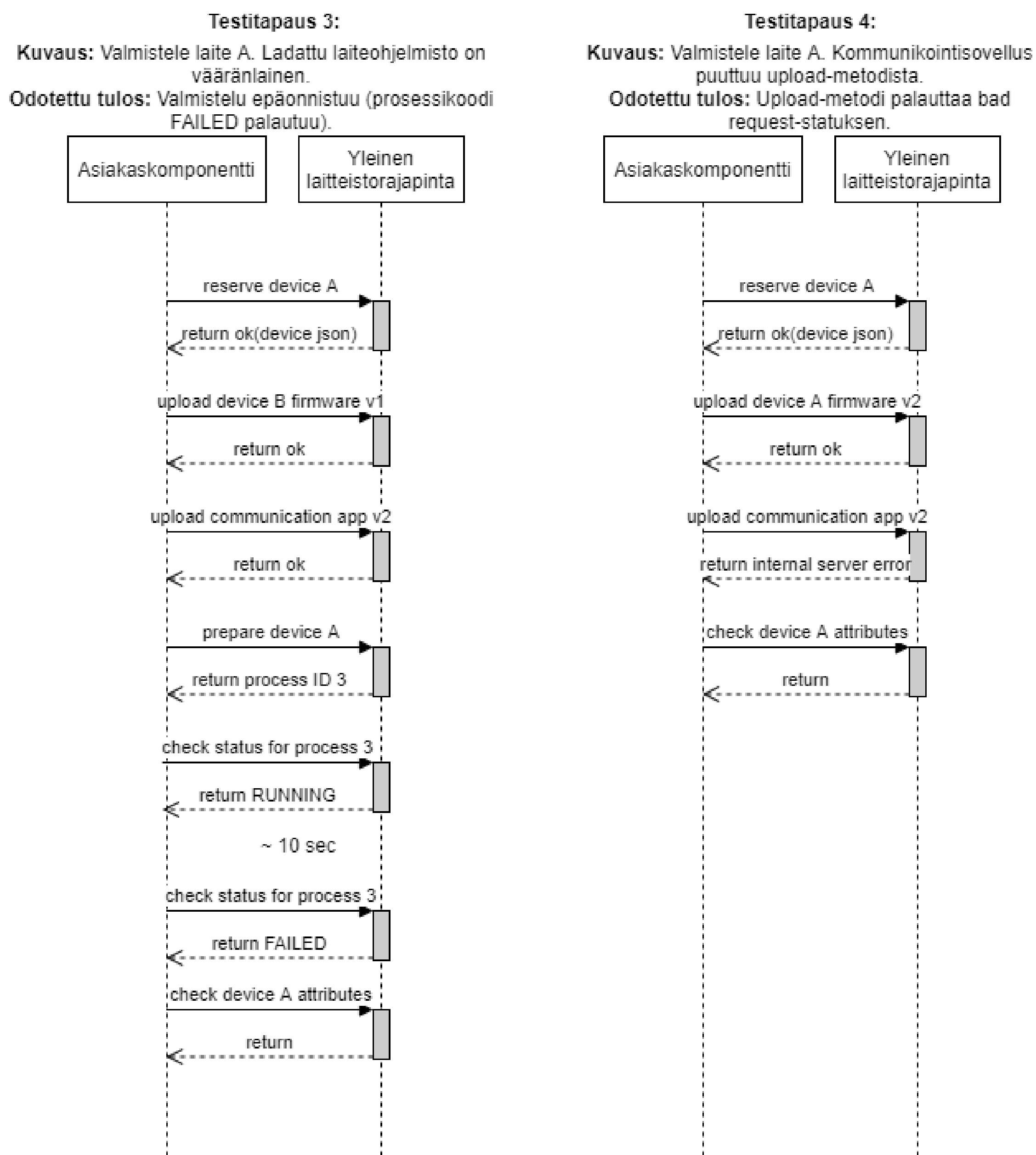
Simulointi suoritettiin siten, että asiakaskomponentti ajoi sekvenssikaavion mukaisesti laaditun rajapintakutsuja lähettävän komentosarjan. Testin aikana komentosarja suoritti

rajapintakomennot määrätyssä järjestyksessä ja palautti laitteistorajapinnan paluuviestit asiakaskomponentille. Testitapausten 1 ja 2 sekvenssikaaviot näkyvät kuvassa 15.



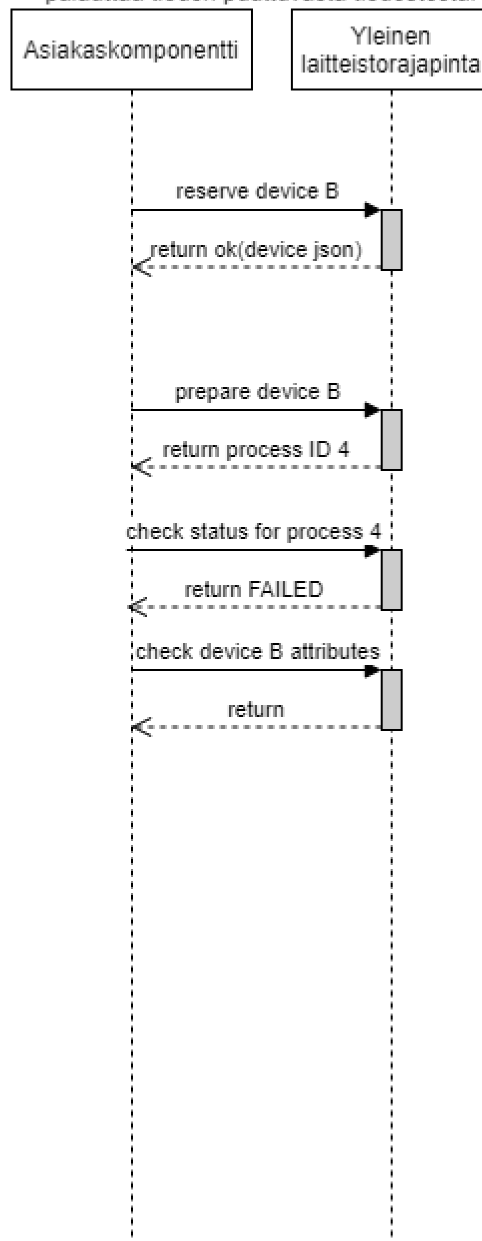
Kuva 15. Onnistuvaa asiakaslaitteen valmistelua simuloivat testitapaukset 1 ja 2.

Testitapausten 3 ja 4 sekvenssikaaviot näkyvät kuvassa 16. Testeissä arvioidaan rajapinnan kykyä tunnistaa aliprosessin virhetila tai puutteellinen asiakaskomponentin kutsu.



Kuva 16. Virhetilanteita simuloivat testitapaukset 3 ja 4.

Kuvassa 17 esitetyssä viimeisessä testitapauksessa 5 testataan laitteistorajapinnan kykyä tunnistaa järjestelmän sisäinen virhe eli päivitystiedoston puuttuminen. Koska päivitystiedosto on järjestelmän ulkopuolinen tiedosto, sen puuttuminen on muita komponentteja todennäköisempää.



Kuva 17. Virhetilannetta simuloiva testitapaus 5.

Järjestelmätestien tulokset näkyvät taulukossa 3. Testitapaukset 1, 4 ja 5 palauttivat odotetut tulokset. Onnistuneet testitapaukset 1 ja 5 suoritettiin laitteen B vanhemmalla laiteohjelmistolla, jonka toimintaperiaatetta käyttäen laitteistorajapinnan päivitystoiminto rakennettiin. Myös testitapaus 4 onnistui, koska se ei edennyt testaukseen asti, vaan palautti odotusten mukaisesti virhekoodin puutteellisesta rajapintakutsusta.

Taulukko 3. Järjestelmätestien tulokset.

Testinro	**Kuvaus**	**Laite**	**Laiteohjelmiston versio**	**Kommunikointisovelluksen versio**	**Odotettu tulos**	**Todellinen tulos**
1	Onnistunut asiakaslaitteen valmistelu.	B	Laite B:n versio 1	Laite B:n versio 1	Prosessikoodi: *COMPLETED*. Laite varattu ja valmisteltu.	Prosessikoodi: *COMPLETED*. Laite varattu ja valmisteltu.
2	Onnistunut asiakaslaitteen valmistelu.	A	Laite A:n versio 2	Laite A:n versio 2	Prosessikoodi: *COMPLETED*. Laite varattu ja valmisteltu.	Prosessikoodi: *FAILED*. Prepare-komentosarja palauttaa tiedon puuttuvasta tiedostosta.
3	Valmistelu epäonnistui: laiteohjelmiston päivitys epäonnistuu, koska asiakaskomponentti lataa vääränlaisen tiedoston rajapintaan.	A	Laite B:n versio 1	Laite A:n versio 2	Prosessikoodi: *FAILED*. Laite varattu mutta ei valmisteltu.	Prosessikoodi: *COMPLETED*. Laite varattu ja valmisteltu.
4	Valmistelu epäonnistuu: asiakaskomponentti yrittää ladata tiedoston rajapintaan, mutta rajapintakutsu ei sisällä tiedostoa.	A			Upload-metodi palauttaa *bad request*-statuksen.	Upload-metodi palauttaa *bad request*-statuksen.
5	Valmistelu epäonnistuu: asiakaskomponentti yrittää aloittaa valmistelun, mutta päivitystiedostot puuttuvat rajapinnasta.	B			Prosessikoodi: *FAILED*. Prepare-komentosarja palauttaa tiedon puuttuvasta tiedostosta.	Prosessikoodi: *FAILED*. Prepare-komentosarja palauttaa tiedon puuttuvasta tiedostosta.

Odotusten vastaisesti testitapaus 2 epäonnistui ja palautti tiedon puuttuvasta tiedostosta. Testitapauksessa 2 käytettävän laitteen A päivityspaketit oli vaihdettu uudempiin versioihin, joiden sisäinen rakenne oli muuttunut aiemmista versioista. Lähemmässä tarkastelussa ongelman aiheuttajaksi paljastui ladattavan päivityspaketin ulkopuolinen tiedosto, joka sisälsi riippuvuuden päivityspaketin sisäiseen kansiorakenteeseen. Koska uuden päivityspaketin sisäinen rakenne oli erilainen, päivityksen käynnistävä tiedosto ei enää löytänyt tarvittavaa tiedostoa halutusta sijainnista vaan päivitys epäonnistui.

Testitapauksessa 3 oli tarkoitus ladata laitteeseen A laitteen B laiteohjelmisto. Testin tarkoituksena oli tarkistaa, havaitseeko laitteistorajapinta valmisteluprosessin epäonnistumisen, jos prosessi käynnistyy, mutta epäonnistuu ajossa. Odotusten vastaisesti prosessi kuitenkin onnistui ja väärä ohjelmisto päivittyi asiakaslaitteelle, joka toimi päivityksen jälkeen normaalisti. Prosessin epäonnistuminen kesken ajon saatiin kuitenkin testattua, koska testitapaus 2 osoitti rajapinnan tunnistavan epäonnistuvan aliprosessin.

6 TULOKSET

Työn lopputuloksena syntyi kahdelle abstraktiotasolle toteutettu Play-ohjelmistokehyksessä rakennettu laitteistorajapinta. Sen arkkitehtuuri noudattaa MVC-I-mallia ja REST-määritelmän mukaisia viestintäperiaatteita. Arkkitehtuurin yksityiskohtainen kuvaus löytyy liitteestä 2.

Rajapinnan arkkitehtuurinen rakenne täyttää etukäteen annetut vaatimukset skaalautuvuudesta ja modulaarisuudesta. Molemmat rajapintatasot toteuttavat samat yleisesti määritellyt toiminnot. Myös projektikohtaiset toiminnallisuusvaatimukset täyttyivät, vaikkakin päivitystoiminnon joustavuudessa havaittiin järjestelmätestauksen aikana puutteita. Mikäli asiakas muuttaa päivityspakettiensa rakennetta merkittävästi, rajapinta ei välttämättä ole niiden kanssa enää yhteensopiva.

Työn lopuksi projektikohtaisesta toteutuksesta luotiin tyhjä mallirajapinta, jota voidaan käyttää pohjana suunniteltaessa uusia plug-in komponentteja uusille asiakkaille ja uusille erilaisilla ominaisuuksilla varustetuille laitteille.

6.1 Työn eteneminen ja havainnot

Tutkimustyön eteneminen noudatti etukäteen laadittua suunnitelmaa. Työ eteni ketterien menetelmien käytäntöjä noudattaen ja edistymistä seurattiin säännöllisesti. Vaikka rajapinnan toimintoja ei toteutettu pyrähdyksissä, tehtävälistaa hallittiin ja tarvittaessa muutoksiin reagoitiin.

Alkuperäisen vaatimusmäärittelyn mukaista ”wait”-toimintoa ei lopulliseen laitteistorajapintaan toteutettu, koska työn toteutuksen aikana syntyi käsitys siitä, että jonotustoiminto ja laitteiston käytön hallintaan liittyvä logiikka tulisi laitteistorajapinnan sijaan sijoittaa testiautomaatiojärjestelmän ydinlogiikkaan. Lisäksi lopulliseen laitteistorajapintaan sisällytettiin toimintoja ja ominaisuuksia, joita ei alkuperäisessä suunnitelmassa ol-

lut, mutta ne arvioitiin toteutuksen aikana kuitenkin hyödyllisiksi tai jopa välttämättömiksi. Näitä ominaisuuksia ovat esimerkiksi laitteen IP-osoitteen vaihtaminen ja uuden päivityspaketin lataaminen rajapintaan.

Ohjaajan toiveesta myös Docker-säiliöinnin mahdollisuutta laitteistorajapinnan käytön yhteydessä tutkittiin. Säiliöinti ei kuitenkaan osoittautunut kannattavaksi toteuttaa rajapintaan, koska projektikohtainen rajapinta sisälsi riippuvuuksia käyttöjärjestelmän edistyneempiin toiminnallisuuksiin, joita ei ollut säiliöinnin sisällä mahdollista toteuttaa. Yleisen rajapinnan osalta se olisi kyllä mahdollista toteuttaa, mutta nykyisessä käyttötarkoituksessa ja kuormituksessa se ei ole tarpeellista.

Ohjelmistokehityksen lopussa tehty rajapinnan laadullinen arviointi osoittautui lopputuloksen kannalta tärkeäksi, koska se paransi rajapinnan luettavuutta. Tämä vuorostaan helpottaa rajapinnan käyttöä ja edelleenkehittämistä tulevaisuudessa. Samalla ymmärrys REST-mallin ominaisuuksista kehittyi työn aikana.

Varsinkin toteutuksen loppuvaiheessa alkoivat ilmetä laitteistorajapinnan roolin epäselvyydet suhteessa muuhun DPC-järjestelmään. Muun muassa laitteistorajapintaa käyttävän komponentin tarkka rooli ja luonne jäi työn toteutuksen aikana epäselväksi, mikä vaikutti järjestelmätestaukseen ja teki testitapausten suunnittelemisesta yhdessä testaajan kanssa haastavaa.

Toteutettu järjestelmätestaus osoitti rajapinnan täyttävän työn alussa asetetut toiminnalliset ja ei-toiminnalliset vaatimukset. Päivityspaketin lataamistoiminnosta tosin paljastui heikkous, joka saattaa aiheuttaa laitteen valmistelun epäonnistumisen, mikäli päivityspaketin rakenne muuttuu merkittävästi. Tämä ongelma kuitenkin johtuu pääasiassa epäselvyyksistä päivityspakettien rakenteessa ja voitaisiin ratkaista esimerkiksi perehtymällä rakenteeseen tarkemmin tai sopimalla asiakkaan kanssa vakiokäytännöistä.

6.2 Yleinen laitteistorajapinta

Yleinen laitteistorajapinta toimii DPC-testiautomaatiojärjestelmän kommunikaatioväylänä asiakaslaitteisiin. Se sisältää rajapinnan määrittelemät metodit, jotka sekä yleisen että projektikohtaisen rajapinnan tulee määritellä. Myös metodeja vastaavat URI-kutsut tulee toteuttaa molemmissa rajapinnoissa.

Projektikohtaista rajapintakerrosta kuvataan yleisessä rajapinnassa nimellä ”test-runner”. Käyttämällä yleisen rajapinnan ”test-runner” -mallien tietokantaa voidaan rajapintakutsut välittää oikealle plug-in-komponentille, joka toteuttaa yleisen rajapintamäärittelyn mukaisen kutsun projektikohtaisella tavalla. Yleinen laitteistorajapinta käyttää WS-kirjastoa kutsujen lähettämiseen ja vastaanottamiseen projektikohtaiselta rajapinnalta. Lista rajapinnan tarjoamista toiminnoista on esitetty taulukossa 4. Rajapintakutsujen tarkka Swagger 2.0-mallinen YAML-kielellä kirjoitettu kuvaus löytyy liitteestä 1.

Taulukko 4. Yleisen laitteistorajapinnan tarjoamat toiminnot.

Tyyppi	URI	Kuvaus
GET	`/test-runners`	Näyttää listan kaikista testiautomaatiojärjestelmän projektikohtaisista rajapinnoista.
GET	`/test-runners/{testRunnerId}`	Näyttää tunnisteen mukaisen projektikohtaisen rajapinnan tiedot.
GET	`/test-runners/{testRunnerId}/devices`	Näyttää listan kaikista projektikohtaisen rajapinnan laitteista.
GET	`/test-runners/{testRunnerId}/connected-devices`	Näyttää listan kaikista verkkoon kytketyistä laitteista.
GET	`/test-runners/{testRunnerId}/free-connected-devices`	Näyttää listan kaikista vapaista verkkoon kytketyistä laitteista.
GET	`/test-runners/{testRunnerId}/devices/{deviceId}`	Näyttää laitetunnisteen mukaisen laitteen kaikki tiedot.
PATCH	`/test-runners/{testRunnerId}/devices/{deviceId}`	Laitteen tilan muokkaus laitetunnisteen perusteella.
POST	`/test-runners/{testRunnerId}/device-files`	Lataa projektikohtaiselle palvelimelle uuden päivityspaketin päivitystunnisteella.
GET	`/test-runners/{testRunnerId}/processes/{processId}`	Näyttää prosessin tilan annetun prosessitunnistenumeron perusteella.
POST	`/test-runners/{testRunnerId}/devices`	Lisää uuden laitteen projektikohtaiselle palvelimelle.
PUT	`/test-runners/{testRunnerId}/devices/{deviceId}`	Laitetietojen muokkaus laitetunnisteen perusteella.
DELETE	`/test-runners/{testRunnerId}/devices/{deviceId}`	Poistaa laitteen palvelimelta laitetunnisteen perusteella.

6.3 Projektikohtainen laitteistorajapinta

Projektikohtaisesta rajapinnasta luotiin työn lopussa mallikappale, josta karsittiin pois kaikki projektikohtaiset toiminnot. Mallikappale määrittelee projektikohtaisen rajapinnan yleiset ominaisuudet, jotka jokaisen tulevan plug-in-komponentin tulisi toteuttaa.

Projektikohtaisen rajapinnan malli sisältää ohjain-luokan rajapinnan sekä itse ohjain-luokan (kuva 18), joka sisältää rajapintametodit ja kuvaukset niiden odotetusta toiminnallisuudesta. Rajapinta määrittelee myös metodien odotetut sisään- ja ulostulot. Myös alemman tason apuohjain-luokista on luotu mallikappaleet. Projektikohtaisen rajapinnan laitemalli (kuva 19) tarjoaa esimerkin asiakaslaitteesta ja sen vähimmäisominaisuuksista, jotka yleinen rajapinta vaatii toimiakseen.

DeviceController

- deviceUtils: DeviceUtilsInterface
- processUtils: ProcessUtilsInterface
- actorSystem: ActorSystem

+ getDeviceQuery(request: Http.Request): Query<Device>
+ getDeviceList(request: Http.Request): Result
+ getConnectedDeviceList(request: Http.Request): Result
+ getFreeConnectedDeviceList(request: Http.Request): Result
+ getDeviceById(deviceId: int): Result
+ patch(request: Http.Request, deviceId: int): Result
+ uploadPackage(id: String): Result
+ getProcessStatus(processIndex: int): Result
+ changeIp(deviceId: int)
+ post(request: Http.Request): Result
+ put(request: Http.Request, deviceId: int): Result
+ delete(deviceId: int): Result

Kuva 18. Projektikohtaisen rajapinnan ohjain-luokka.

Device

- deviceId: int
- reserved: boolean
- prepared: boolean
- connected: boolean

Kuva 19. Rajapinnan asiakaslaite-luokka ja siltä vaaditut vähimmäisominaisuudet.

Alkuperäisestä asiakaslaitteen mallista poiketen nimiattribuutti on jätetty pois, koska sen olemassaolo ei ole järjestelmän toiminnan kannalta välttämättömänä. Malliin lisättiin sen tilalle laitteen yhteystilasta kertova attribuutti, koska yhdistettyjen laitteiden hakeminen järjestelmästä oli yksi vaatimusmäärittelyn mukaisista halutuista ominaisuuksista. Lisäksi laitteella tulee olla alkuperäisen suunnitelman mukaisesti yksilöllinen tunniste sekä varaustilan ja valmiustilan ilmoittavat attribuutit laitteen seurannan mahdollistamiseksi.

Uutta asiakaslaitetyyppiä luotaessa laitteelle voidaan toteuttaa myös muita, asiakaskohtaisia ominaisuuksia, jotka laitteistorajapinnan käyttäjä näkee ja pystyy myös hyödyntämään haluamallaan tavalla. Esimerkiksi laitteen käynnistäminen tai sammuttaminen rajapinnan kautta on mahdollista lisäämällä virranhallintaa kuvaava resurssi ja toteuttamalla PATCH-metodiin sitä vastaava tapauskohtainen toiminnallisuus. Tyypillisempi käyttötarkoitus on todennäköisesti kuitenkin laitteen hakeminen tietyn ominaisuuden perusteella.

7 JOHTOPÄÄTÖKSET

Työn tavoitteena oli suunnitella plug-in-komponenteilla laajennettavissa oleva laitteistorajapinta testiautomaatiojärjestelmän ja asiakaslaitteiden välille. Laitteistorajapinnan tärkeimpiä vaatimuksia olivat yleisten rajapintakutsujen määrittely ja mahdollisuus käyttää plug-in-komponentteja fyysisesti erillään muusta järjestelmästä.

Kehitystyön lopputuloksena syntynyt laitteistorajapinta suunniteltiin REST-arkkitehtuurimallia noudattaen. Rajapinnan modulaarisuus toteutettiin käsittelemällä plug-in-komponentteja oliokokoelmana arkkitehtuurissa. Tiettyyn plug-in-olioon viittaamalla rajapintakutsut on mahdollista välittää asiakaslaitteille. REST-mallin seuraaminen mahdollistaa myös laitteistorajapinnan helpon yhteensovittamisen muihin järjestelmän komponentteihin tulevaisuudessa. Toteuttamalla yleinen laitteistorajapinta projektikohtaisesta laitteistorajapinnasta erilliseksi ilmentymäksi mahdollistettiin komponenttien eriyttäminen fyysisesti toisistaan tarvittaessa.

Yksi kehitetyn laitteistorajapinnan mahdollisista puutteista on sen sisältämien toimintojen vähäinen automatisointi. Rajapinta jättää suurimman vastuun sen toimintojen oikeanlaisesta käytöstä asiakaskomponentille. Käynnistettyä päivitysprosessia ei esimerkiksi pysty suoraan yhdistämään tiettyyn laitteeseen vaan vastuu prosessinhallinnasta jätetään asiakaskomponentille. Myös useampien laitteiden samanaikainen valmistelu saattaa aiheuttaa ongelmia ainakin päivitettävän laiteohjelmiston hallinnassa ja laitteiden varaamisessa.

Järjestelmätestauksessa suurimmaksi rajapinnan puutteeksi paljastui päivityksenhallintakomentosarjojen yhteensopivuusongelmat päivityspakettien kanssa, mikäli päivityspakettien rakenne muuttuu paljon. Toisena päivitystoiminnon heikkoutena voidaan pitää myös sen yksinkertaisuutta, koska tehdyn päivityksen tietoja ei taltioida mihinkään, vaan rajapinta luottaa asiakaskomponentin tietävän, mikä ohjelmisto asiakaslaitteeseen on kulloinkin ladattu. Tulevissa projekteissa toimintoa voi olla tarpeen laajentaa esimerkiksi tallentamaan tiedon ladatusta ohjelmaversiosta tai hallinnoimaan useita vaihtoehtoisia versioita.

Syitä päivitystoiminnon yhteensopivuusongelmiin ja yksinkertaisuuteen ovat muun muassa teknisen osaamisen rajoitteet, työhön käytettävän ajan rajaaminen yksinkertaistamalla toimintoja ja rajapinnan asiakasvaatimusten ajoittainen epäselvyys ja epätarkkuus. Ilmenneitä rajoitteita ei laadun arvioinnissa pidetty rajapinnan ensimmäisen version kannalta kriittisinä, koska toistaiseksi siihen kytkettävien laitteiden määrä on pieni. Jatkokehityksen kannalta ne on kuitenkin syytä huomioida.

Käytännön tasolla merkittävimmät käyttöönoton estävät puutteet laitteistorajapinnassa ovat tietoturvan ja käyttäjävarmennuksen puuttuminen. Molemmat ominaisuudet oli kuitenkin rajattu tämän työn ulkopuolelle.

Koska tutkimuksessa toteutettiin vain yksi projektikohtainen rajapintakomponentti, ei rajapinnan määrittelyn yleispätevyyttä ole mahdollista todentaa täysin luotettavasti. Tulevat projektit saattavat osoittaa, että nykyistä määrittelyä on tarvetta vielä laajentaa. Nykyinen toimiva malli antaa kuitenkin viitteitä siitä, että sama rajapintamäärittely olisi mahdollista toteuttaa myös toisenlaisille asiakaslaitteille. Vaikka toimintojen tarkka toteutus vaihtelee, rajapintametodien tavoitteen tulisi pysyä samana. Siksi työssä toteutettu projektikohtaisen laitteistorajapinnan tarkka kuvaus ja dokumentointi on työn tavoitteen ja testiautomaatiojärjestelmän tulevaisuuden kannalta ensiarvoisen tärkeää.

Suurin ongelma rajapinnan toteutuksessa oli vaatimusten epäselvyys, joka johtui itse testiautomaatiojärjestelmän määrittelyn puutteista. Laitteistorajapinnan ja sitä hyödyntävän testiautomaatiojärjestelmän jatkokehityksen kannalta onkin tärkeää seuraavaksi määritellä koko järjestelmän vaatimukset ja nämä vaatimukset täyttävä arkkitehtuuri. Tämä mahdollistaa tulevaisuudessa myös laitteistorajapinnan vaatimusten tarkentamisen ja edelleenkehittämisen. Tämän jälkeen olisi mahdollista lisätä järjestelmän toimintaan myös toinen projektikohtainen toteutus, joka toteutuessaan mahdollistaisi laitteistorajapinnan laajennettavuuden luotettavan arvioinnin. Myös järjestelmäkokonaisuuden tuotteistaminen mahdollistuu, kun sen arkkitehtuurisista ominaisuuksista syntyy varmuus.

LÄHDELUETTELO

Badgett, Tom, Glenford J., Myers & Corey, Sandler (2012). *The Art of Software Testing*. John Wiley & Sons, Inc. Third Edition. ISBN 978-1-118-03196-4.

Bass, Len, Paul, Clements & Rick, Kazman (2013). *Software architecture in practice*. Third Edition. Upper Saddle River, NJ: Addison-Wesley. ISBN 978-0-321-81573-6.

Berner, Stefan, Rudolf K., Keller & Roland, Weber (2005). *Observations and Lessons Learned from Automated Testing*. Proceedings of the 27th International Conference on Software Engineering. St. Louis, MO, USA. s. 571–579.

Biehl, Matthias (2015). *API Architecture: The Big Picture for Building APIs*. Volume 2. API-University Press. ISBN 978-1-5086-7664-5.

Bourque, Pierre & Richard E., Fairley (2014). *Guide to the Software Engineering Body of Knowledge v3.0 SWEBOK*. IEEE Computer Society. ISBN 978-0-7695-5166-1.

Chatley, Robert, Susan, Eisenbach & Jeff, Magee (2003). *Modelling a Framework for Plugins*. Proceedings of the Workshop on Specification and Verification of Component-Based Systems. Department of Computing, Imperial College London.

Chaufournier, Lucas, Prateek, Sharma, Prashant, Shenoy & Y. C., Tay (2016). *Containers and Virtual Machines at Scale: A Comparative Study*. Middleware '16: Proceedings of the 17th International Middleware Conference, 1 s. 1–13.

Docker Inc. (2019). What is a Container? [verkkodokumentti]. [22.11.2019]. Saatavissa: https://www.docker.com/resources/what-container.

dotCloud (2019). *About the dotCloud Platform* [verkkodokumentti]. [22.11.2019]. Saatavissa: https://web.archive.org/web/20140702231323/https:// www.dotcloud.com/about.html.

Fielding, Roy T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine. Doctor of Philosophy in Information and Computer Science.

Fielding, Roy T. & Richard N., Taylor (2000). *Principled Design of the Modern Web Architecture*. Proceedings of the 2000 International Conference on Software Engineering, Limerick, Ireland. s. 407–416.

Filander, Mika (2019a), Managing Director. Devatus Oy. Arkkitehtuurisuunnittelupalaveri, Vaasa, 2.4.2019.

Filander, Mika (2019b), Managing Director. Devatus Oy. Rajapintasuunnittelupalaveri, Vaasa, 11.3.2019.

Filander, Mika (2019c), Managing Director. Devatus Oy. Viikkopalaveri, Vaasa, 20.6.2019.

Haikala, Ilkka & Tommi, Mikkonen (2011). *Ohjelmistotuotannon käytännöt*. 12. painos. Helsinki: Talentum. ISBN 978-952-14-1754-2.

IEEE-SA Standards Board (2000). *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE Standard 1471-2000. ISBN 0-7381-2519-9.

Kasurinen, Jussi (2013). *Ohjelmistotestauksen käsikirja*. 1. painos. Jyväskylä: Docendo. ISBN 978-952-5912-99-9.

Kasurinen, Jussi, Kari, Smolander & Ossi, Taipale (2009). *Software Test Automation in Practice: Empirical Observations*. Advances in Software Engineering, Special Issue on Software Test Automation. Hindawi Publishing.

Koskimies, Kai & Tommi, Mikkonen (2005). *Ohjelmistoarkkitehtuurit*. Helsinki: Talentum. ISBN 952-14-0862-6.

Krasner, Glenn E. & Stephen T., Pope (1988). *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*. Journal of Object-Oriented Programming, 1 (3) s. 26–49.

Lightbend Inc. (2019a). *Play Framework makes it easy to build web applications with Java & Scala* [verkkodokumentti]. [18.11.2019]. Saatavissa: https://www.playframework.com/.

Lightbend Inc. (2019b). *What is Play?* Play 2.7.x documentation [verkkodokumentti]. [18.11.2019]. Saatavissa: https://www.playframework.com/documentation /2.7.x/Introduction.

Lightbend Inc. (2020a). *Calling REST APIs with Play WS* [verkkodokumentti]. [15.1.2020]. Saatavissa: https://www.playframework.com/documentation /2.8.x /JavaWS.

Lightbend Inc. (2020b). *Dependency Injection* [verkkodokumentti]. [24.1.2020]. Saatavissa: https://www.playframework.com/documentation/2.8.x/JavaDependencyInjection.

Nguyen, Hoang (2019). *DPC_setup*. Slack-viesti Jani Lehtisalolle 13.12.2019.

Qian, Kai, Xiang, Fu, Lixin, Tao, Chong-Wei, Xu & Jorge L., Díaz-Herrera (2010). *Software architecture and design illuminated*. Sudbury, Mass: Jones and Bartlett cop. ISBN 978-0-7637-5420-4.

Ramler, Rudolf & Klaus, Wolfmaier (2006). *Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost*. Proceedings of the 2006 International Workshop on Automation of Software Testing, Shanghai, China. p. 85–19.

Restfulapi.net (2020). *REST Resource Naming Guide*. REST API Tutorial [verkkodokumentti]. [7.2.2020]. Saatavissa: https://restfulapi.net/resource-naming/.

Richard-Foy, Julien (2014). *Play Framework Essentials*. Packt Publishing 2014. ISBN 978-1-78398-240-0.

LIITTEET

LIITE 1. Laitteistorajapinnan yksityiskohtainen kuvaus


```
swagger: "2.0"
info:
  title: DPC Hardware Interface
  description: DPC common hardware interface layer that defines all functions that pro-
ject specific layer should implement. Requests with 'Tester' tag are prioritized as they
are used for preparing devices for testing. Requests with 'Admin user' tag are needed
less often. They are reserved for administrative usage because they enable altering the
system and may reveal sensitive information.
  version: '1.0'
host: localhost
basePath: /
schemes:
  - http
paths:
  /test-runners:
    get:
      summary: Shows a list of test runners.
      description: Shows a list of all test runners in the system. Only accessible for
admin-istrative users for security reasons.
      tags:
        - Admin user
      responses:
        200:
          description: The found test runners and their details are given in json form.
          schema:
            $ref: '#/definitions/TestRunner'
  /test-runners/{testRunnerId}:
    parameters:
        - $ref: '#/parameters/testRunnerParameter'
    get:
      summary: Shows test runner details by ID number.
      tags:
        - Tester
      responses:
        200:
          description: The found test runner and all of its details are given in json
form.
          schema:
            $ref: '#/definitions/TestRunner'
        404:
          description: Test runner with given ID not found.
  /test-runners/{testRunnerId}/devices:
    parameters:
        - $ref: '#/parameters/testRunnerParameter'
    get:
      summary: Returns a list of devices.
      description: Returns a list of all project devices. In future version list can be
filtered with given device parameter value.
      tags:
        - Tester
      responses:
        200:
          description: The found devices and their details are given in json form.
          schema:
            $ref: '#/definitions/Device'
    post:
      summary: Adds a new device.
      description: Adds new device to project. Common hardware layer requires only de-
vice ID number, 'reserved', 'prepared' and 'connected' fields to be included with the
device object. Project layer has its own requirements for data fields. Once the require-
ments for both layers are fulfilled post will be successful. All fields required by com-
mon layer are pre-set. User should only be able to modify project specific fields of the
device.
      tags:
        - Admin user
      consumes:
```

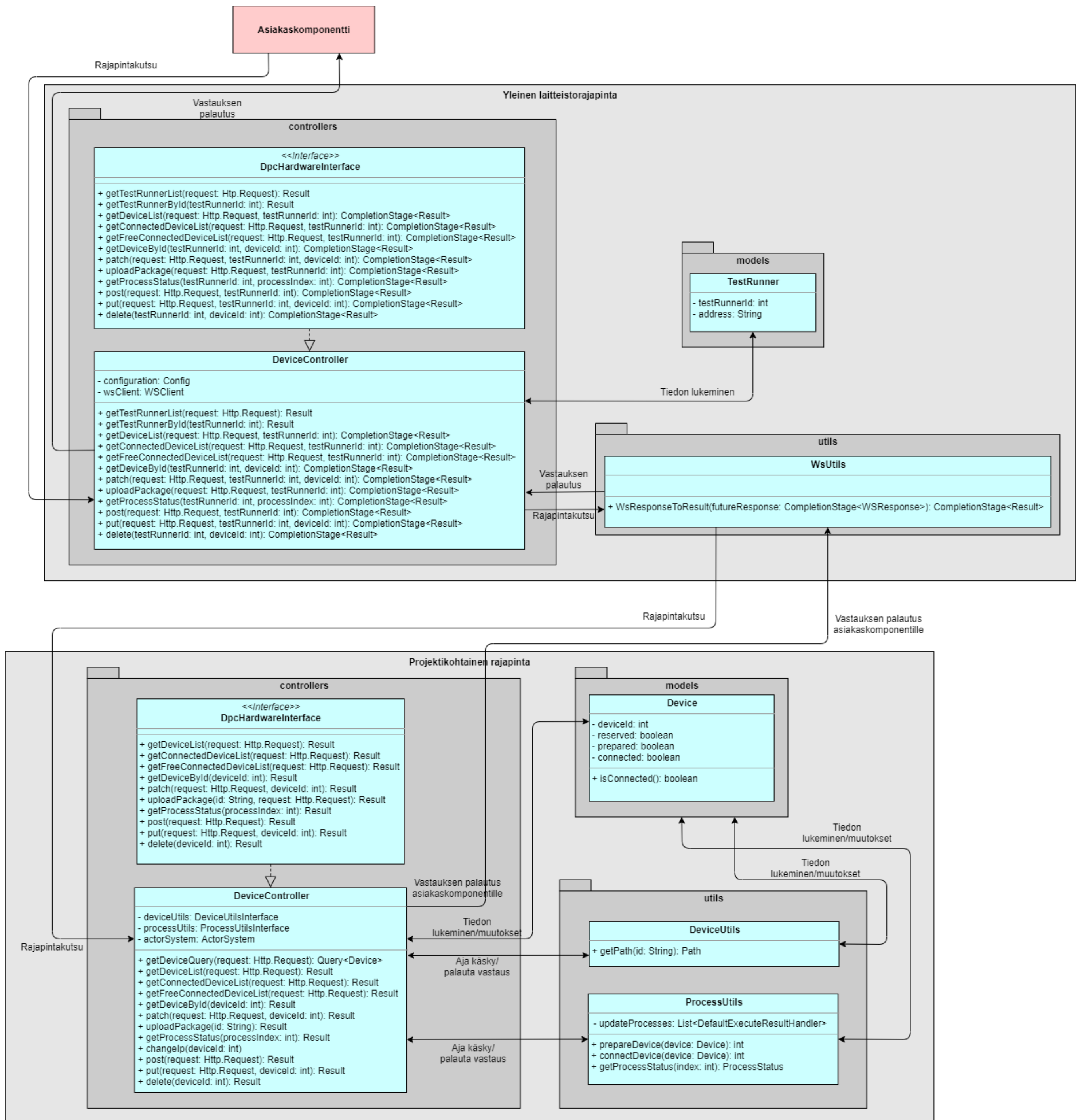
```
        - application/json
      parameters:
        - in: body
          name: Device
          description: Add new device.
          required: true
          schema:
            type: object
      responses:
        200:
          description: Post request is accepted and device is saved to database. Returns
de-vice json.
          schema:
            $ref: '#/definitions/Device'
        400:
          description: If requestBody is missing some required data, 'bad request' is
thrown.
        403:
          description: If some of the data fields are out of acceptable bounds, 'forbid-
den' status is thrown.
  /test-runners/{testRunnerId}/connected-devices:
    get:
      summary: Returns a list of connected devices.
      description: Returns a list of all devices connected to the DPC local network.
      tags:
        - Tester
      parameters:
         - $ref: '#/parameters/testRunnerParameter'
      responses:
        200:
          description: The found devices and their details are given in json form.
          schema:
            $ref: '#/definitions/Device'
  /test-runners/{testRunnerId}/free-connected-devices:
    get:
      summary: Returns a list of free and connected devices.
      description: Returns a list of all devices connected to local network that are not
reserved to any test.
      tags:
        - Tester
      parameters:
         - $ref: '#/parameters/testRunnerParameter'
      responses:
        200:
          description: The found devices and their details are given in json form.
          schema:
            $ref: '#/definitions/Device'
  /test-runners/{testRunnerId}/devices/{deviceId}:
    parameters:
        - $ref: '#/parameters/testRunnerParameter'
        - $ref: '#/parameters/deviceParameter'
    get:
      summary: Shows device details by ID number.
      description: Shows all device details with given ID number.
      tags:
        - Tester
      responses:
        200:
          description: The found device and all of its details are given in json form.
          schema:
            $ref: '#/definitions/Device'
        404:
          description: Device or test runner with given ID not found.
    patch:
      summary: Changes device state.
      description: Change state of 'reserved', 'prepared' or 'connected' booleans. De-
pending on project it is also possible to power up or shut down device.
      tags:
        - Tester
      consumes:
        - application/json
      parameters:
```

```
        - in: body
          name: changeParameter
          description: Field name and its new value. Request body must contain at least
one parameter.
          required: true
          schema:
            type: object
            properties:
              reserved:
                type: boolean
                example: true
      responses:
        200:
          description: Device state changed successfully. Returns device json.
          schema:
            $ref: '#/definitions/Device'
        400:
          description: Request body does not fulfill given requirements.
        404:
          description: Device or test runner with given ID not found.
    put:
      summary: Modifies device data fields by ID number.
      description: Allows modification of data fields from device with given ID number.
Re-quest body must contain at least device ID number and one data field.
      tags:
        - Admin user
      consumes:
        - application/json
      parameters:
        - in: body
          name: modifiedDevice
          description: New device details to replace old data.
          required: true
          schema:
            type: object
            required:
              - deviceId
            properties:
              deviceId:
                type: integer
              attributeToBeChanged:
                type: string
      responses:
        200:
          description: Modification succeeded and new data is updated to data-base. Re-
turns the new device data as json.
          schema:
            $ref: '#/definitions/Device'
        400:
          description: Request body does not fulfill given requirements.
        404:
          description: Device or test runner with given ID not found.
    delete:
      summary: Deletes device by ID number.
      description: Deletes device with given ID number from the database.
      tags:
        - Admin user
      responses:
        200:
          description: Device successfully deleted from the database.
        404:
          description: Device or test runner with given ID not found.
  /test-runners/{testRunnerId}/device-files:
    post:
      summary: Uploads a new device file to test runner server.
      description: Uploads new update package file to test runner server. New uploaded
file replaces the old one in location determined by fileId.
      tags:
        - Tester
      consumes:
         - multipart/form-data
      parameters:
```

```
        - in: formData
          name: updateFile
          type: file
          required: true
          description: The file to upload.
        - $ref: '#/parameters/testRunnerParameter'
        - in: path
          name: fileId
          type: string
          required: true
          description: Specify which update file type will be uploaded.
      responses:
        200:
          description: Upload successful. Replaces the old file in the destination.
        400:
          description: Missing file.
        404:
          description: Test runner with given ID not found.
        500:
          description: Update package upload failed.
  /test-runners/{testRunnerId}/processes/{processId}:
    get:
      summary: Checks sub-process status.
      description: Checks process status for sub-process with given process ID number.
      tags:
        - Tester
      parameters:
        - $ref: '#/parameters/testRunnerParameter'
        - in: path
          name: processId
          type: integer
          required: true
          description: Specify process ID number for checking its status.
      responses:
        200:
          description: Returns status for given process ID number. Currently running
process returns 'RUNNING'. If process is completed within given timeframe status changes
to 'COMPLET-ED'. If process takes longer to finish status changes to 'FAILED'. 'NO_PRO-
CESS' usually means process startup failed or process start wasn't requested.
        404:
          description: Test runner with given ID not found.
parameters:
  testRunnerParameter:
    in: path
    name: testRunnerId
    type: integer
    required: true
    description: Specify the associated test runner.
  deviceParameter:
    in: path
    name: deviceId
    type: integer
    required: true
    description: Specify the associated device.
definitions:
  Device:
    type: object
    properties:
      deviceId:
        type: integer
      reserved:
        type: boolean
        description: Indicates whether or not the device is reserver for test run.
      prepared:
        type: boolean
        description: Indicates whether or not the device has been prepared for testing.
User is not allowed to change this value directly. Changes back to false when 'reserved'
turns false after testing.
      connected:
        type: boolean
        description: Indicates whether or not the device is currently connected to local
net-work and responds. User is not allowed to change this value directly.
```

```
      powered:
        type: boolean
        description: Indicates whether or not the device is currently powered up. User
is not allowed to change this value directly. Not required by common hardware interface.
  TestRunner:
      type: object
      properties:
        testRunnerId:
          type: integer
        address:
           type: string
```

LIITE 2. Laitteistorajapinnan arkkitehtuuri luokkakaaviona



LIITE 3. Laitteistorajapinnan ominaisuuksien toteutuksen kuvaus

Lista sisältää kaikki rajapintaan toteutetut toiminnot tai toimintokokonaisuudet ja niiden tarkat kuvaukset numeroituna toteutusjärjestyksen mukaan. Joidenkin toimintojen lopullinen muoto ja käyttötapa laitteistorajapinnassa muuttui laadun arvioinnin jälkeen, mutta niiden sisäinen rakenne pysyi pääpiirteittäin samanlaisena.

1. **Hae tietokannassa olevia laitteita tai plug-in komponentteja**

   Ensimmäisenä luoduille projektipohjille toteutettiin metodit, joiden avulla tietokannasta voidaan hakea tiedot olemassa olevista asiakaslaitteista. Tähän sisältyvät kutsu, jolla voidaan hakea rajapinnan tietokannasta lista kaikista sille tallennetuista asiakaslaitteista, ja vastaava, jolla voidaan hakea tiedot yksittäisestä asiakaslaitteesta tunnisteen perusteella. Aluksi metodit toteutettiin vain projektikohtaiseen rajapintaan, mutta myöhemmin myös yleiseen rajapintaan luotiin vastaavat metodit, joiden tehtävä on kutsua alkuperäisiä metodeja. Myös projektikohtaisten rajapintojen eli plug-in-komponenttien listan palauttava metodi lisättiin yleiseen rajapintaan.

2. **Hae saatavilla olevia laitteita**

   Ensimmäisenä rajapinnan vaatimusmäärittelyyn liittyvistä toiminnoista toteutettiin metodi, joka tarkistaa onko tietty asiakaslaite kytkettynä DPC:n lähiverkkoon. Metodi tarkistaa tietyn IP-osoitteen saatavuuden lähiverkossa ja palauttaa tuloksen totuusarvona. Toiminto toteutettiin niin, että tieto saatavuudesta voidaan palauttaa sekä laitetietoihin että erilliseen rajapintakutsuun, jolla saatavuus tarkistetaan laitetunnisteen perusteella. Seuraavaksi toimintoa laajennettiin siten, että projektikohtaisen rajapinnan tietokannasta voidaan hakea kaikki lähiverkkoon kytkettynä olevat laitteet.

3. **Muokkaa kaikkia laitteen tietoja**

   Laitteen varaustilanne- ja testivalmius-määreiden tilaa haluttiin pystyä muuttamaan rajapinnan kautta tapahtuvilla komennoilla. Tätä varten rajapintaan kehitettiin kaksi vaihtoehtoista lähestymistapaa, joista ensimmäinen on REST:n PUT-kutsua käyttävä

metodi. PUT mahdollistaa asiakaslaitteen minkä tahansa tietokentän muokkaamisen vapaasti. Varaustilanne- ja testivalmius-määreiden tilanmuokkauksen lisäksi PUT mahdollistaa myös muiden laitemääreiden muokkaamisen, mutta jättää käyttöliittymälle isomman roolin oikeanlaisen toiminnallisuuden rakentamiseen ja voi jopa mahdollistaa rajapinnan väärinkäytökset, koska se mahdollistaa periaatteessa myös koko resurssin korvaamisen uudella.

4. **Muuta laitteen varaustilaa tai valmiustilaa**

Toinen kehitetty tapa toteuttaa varaustilan ja testivalmiuden muuttaminen oli neljän metodin sarja, jossa halutulle laitteelle välitettiin tietty komento osoitekentässä, kuten ”laite/{tunniste}/varaa” tai ”laite/{tunniste}/vapauta”. Metodit mahdollistavat vain yksittäisen totuusarvomuuttujan tilan vaihtamisen, pitäen rajapinnan turvallisempana. Lähestymisen heikkoutena on, etteivät kyseiset toiminnot noudata REST-rajapinnan määritelmää yhtä hyvin kuin PUT.

5. **Laitteen virranhallinta**

Koska testiautomaatiojärjestelmän alkuperäinen, suppea rajapinta mahdollisti asiakaslaitteen käynnistämisen tai sammuttamisen suoraan pilvipalvelun testiputkesta, haluttiin vastaava ominaisuus toteuttaa myös uuteen rajapintaan. Koska asiakaslaite yksin ei mahdollista virran kytkemistä päälle tai pois, toiminnon toteuttaminen tarkoitti käytännössä asiakaslaitteen virransaannin ohjaamista relepiirin välityksellä. Tarkoitusta varten luotiin komentosarjatiedosto, jonka välityksellä rajapinta pystyy ohjaamaan relettä. Koska laitteen uudelleenkäynnistykseen yhteydessä tulee asiakkaan mukaan sisällyttää kymmenen sekunnin viive sammutuksen ja käynnistyksen välille, toteutettiin käynnistys- ja sammutuskomentojen lisäksi myös erillinen uudelleenkäynnistys. Virranhallintakomentojen lisäksi luotiin samalla myös mahdollisuus tarkistaa laitteen virransaannin tila releen avulla.

6. Lisää tai poista laite

Kehitetty rajapinta ei tässä vaiheessa vielä mahdollistanut uuden asiakaslaitteen lisäämistä rajapinnan kautta järjestelmään, vaan edellytti joko käsin tapahtuvaa lisäämistä suoraan tietokantaan tai niin sanotun *kippausdata*-metodin käyttöä, joka alustaa tietokannan valmiiksi syötetyillä tiedoilla. Tästä syystä rajapintaan päätettiin lisätä laitteen lisäämisen mahdollistava metodi. Metodi vastaanottaa lisättävän laitteen tiedot JSON-muodossa ja tarkistaa, että ne täyttävät tietyt projektikohtaisen rajapinnan vaatimat reunaehdot. Tämän jälkeen uusi laite tallennetaan tietokantaan ja sille luodaan yksilöllinen tunniste. Tallennetut tiedot palautetaan käyttäjälle. Myös laitteen poistamisen tunnisteluvun perusteella mahdollistava metodi luotiin.

7. Päivitä laitteen ohjelmisto ja seuraa päivitysprosessin etenemistä

Rajapinnan toiminnoista toteutuksen kannalta vaativin oli asiakaslaitteen ohjelmistopäivityksen toteutus. Koska laitteistorajapinnan laitteet ovat asiakasyrityksen kehittämiä prototyyppivaiheessa olevia sulautettuja järjestelmiä (ks. 4.1) niiden sisältämien kahden erillisen ohjelmiston päivittäminen vaati usean tietokoneohjelman ja komentosarjapaketin asentamista laitteistorajapintapalvelimelle. Tämän lisäksi kumpikin asiakaslaitetyyppi vaati erilaisen toimintosarjan päivityksen toteuttamiseksi onnistuneesti.

Ensimmäiseksi rajapintaan laadittiin kaksi erillistä päivitystoimintoa; ensimmäinen laiteohjelmiston päivitykselle ja toinen kommunikointisovelluksen päivitykselle (ks. 4.1). Käyttäjäkutsun yhteydessä toiminnot tarkistavat tietokannasta asiakaslaitteen tyypin ja suorittavat sen perusteella komentosarjatiedoston, joka toteuttaa päivityksen laitteen edellyttämällä tavalla.

Päivityksen kesto riippuu laitetyypistä ja kestää kaikissa tapauksissa niin kauan, että prosessi ei ehdi palauttaa vastausta REST-kutsun tekevälle käyttäjälle ennen sen vanhenemista. Tästä syystä molemmat päivitysmetodit toteutettiin siten, että komentosarjatiedoston käynnistyessä sille luodaan prosessitunniste ja niin sanottu tarkkailija, joka

ilmoittaa pyydettäessä prosessin tilan. Päivitysprosessin käynnistämisen jälkeen käyttäjälle palautetaan päivitysprosessin tunnistenumero. Tämän jälkeen järjestelmään luotiin erillinen toiminto aliprosessien tilan seuraamista varten. Toiminto palauttaa käyttäjälle tiedon prosessin tilasta (esimerkiksi: *kesken* tai *valmis*) parametrina annettua prosessitunnistetta vastaan.

8. Vaihda laitteen IP-osoite

Osana päivitystoimintojen rakentamista kehitettiin myös laitteen IP-osoitteen vaihtamisen mahdollistava toiminto osaksi päivityskomentosarjaa. Tämä oli välttämätöntä, koska asiakaslaitteet palauttavat niille määritetyn osoitteen takaisin oletusarvoonsa päivityksen yhteydessä. IP-osoitteen vaihtamistoiminto lisättiin osaksi päivitystoimintoja, jotta osoitteenvaihdos voidaan tehdä automaattisesti. Se kuitenkin lisättiin myös osaksi projektikohtaista rajapintaa, mikäli käyttäjälle tulee tarve muuttaa osoitetta myös käsikäyttöisesti.

9. Valmistele laite testaukseen

Kun projektikohtaisen rajapinnan erilliset päivitystoiminnot oli saatu toimimaan halutulla tavalla, toteutettiin seuraavaksi laitteen testaukseen valmisteleva metodi, joka toteutettiin projektikohtaisessa rajapinnassa yhdistämällä erillisten päivitystoimintojen logiikka uuteen komentosarjatiedostoon. Tällöin metodia kutsuttaessa aliprosessina ajettava komentosarja suorittaa molempiin päivityksiin vaadittavat toiminnot oikeassa järjestyksessä. Kuten aiemmissa päivitystoiminnoissa, myös laitteen valmisteleva metodi palauttaa käyttäjälle prosessitunnisteen, jonka avulla valmisteluprosessin tilaa voidaan seurata.

10. Yhdistä laite testiautomaatiojärjestelmään automaattisesti

Yleisen rajapinnan näkökulmasta metodin tarkoitus on automatisoida laitteen yhdistäminen testiautomaatiojärjestelmään soveltaen projektikohtaista tapaa, jolla laitteet on mahdollista kytkeä. Tämän asiakasprojektin tapauksessa toiminto automatisoi IP-osoitteen vaihtamisen. Koska kaikkien asiakaslaitteiden IP-osoite on oletusarvoisesti

ennen järjestelmään kytkemistä sama, suunniteltiin yhdistämistoiminto siten, että asiakaslaitteen fyysisen järjestelmään kytkemisen jälkeen tietokantaan tulee syöttää laitteen halutut tiedot, kuten haluttu IP-osoite. Tämän jälkeen syöttämällä yhdistämistoiminnolle laitteen tunnistenumero se muuttaa automaattisesti IP-osoitteen tietokannasta löytyvän osoitteen mukaiseksi. Myös tämä toiminto käyttää tarkoitusta varten suunniteltua komentosarjatiedostoa, ja prosessin tilaa voidaan seurata prosessitunnisteen avulla.

11. Lataa uusi ohjelmistopaketti

Viimeisenä toteutettavana vaatimusmäärittelyn mukaisena toimintona ennen rajapinnan arviointia toteutettiin uuden ohjelmistopaketin lataaminen laitteistorajapintapalvelimelle. Koska asiakasprojektissa laitteilla käytettäviä ohjelmistoja on kolmea eri tyyppiä; laitteiden A ja B laiteohjelmistot sekä kommunikointisovellus (ks. 4.3), joka on molemmille laitetyypeille sama; tulee ladattavat ohjelmistopaketit pystyä erittelemään toisistaan. Luotu toiminto mahdollistaa paketin lataamisen samassa yhteydessä annettavan tiedostotunnisteen mukaiseen sijaintiin.