Jari Nykänen

SIMULAATIO OSANA OHJELMISTOKEHITYSTÄ

Opinnäytetyö
Tietojenkäsittely

2018

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Jari Nykänen	Tradenomi (AMK)	Syyskuu 2018
Opinnäytetyön nimi Simulaatio osana ohjelmistokehitystä		45 sivua 3 liitesivua
Toimeksiantaja Observis Oy		
Ohjaaja Janne Turunen		

Tiivistelmä

Tässä opinnäytetyössä tutkitaan, kuinka simulaatioita voidaan hyödyntää ohjelmistokehityksen ja -testaamisen työkaluna. Tämän tutkimuksen pohjalta kehitetään myös toimiva laiteintegraatio, laitesimulaatio ja simulaation hallintanäkymä. Opinnäytetyö toteutettiin osana Observis Oy:n ObSAS-tilannekuvaohjelmiston kehitystä. Työssä käsitellään Java-ohjelmointikielellä tehtävissä ja suoritettavissa simulaatioissa tarvittavia ohjelmia ja tekniikoita.

Opinnäytetyön teoriaosuudessa selvitetään, mitä on ohjelmistotestaaminen ja miten siihen voidaan liittää simulaatioita. Lisäksi tutkitaan, mitä etuja simulaatiot tuovat ja milloin niitä on järkevää käyttää. Teoriaosuudessa kerrotaan myös, miten simulaatio ja emulaatio eroavat toisistaan. Teoriaosuus pohjustaa myös käytännön toteutuksessa oleellisessa osassa olevat tekniikat ja laitteiston.

Opinnäytetyön käytännön osiossa kuvaillaan käytetty ohjelmistoarkkitehtuuri yleisellä tasolla, jotta tämän työn lukijalle muodostuisi selkeä kuva siitä, mitä käytännön osiolla pyritään saavuttamaan. Käytännön osiossa myös kerrotaan, kuinka oikea laite - tässä työssä Vaisalan WXT536-sääasema - pystytään liittämään osaksi olemassa olevaa ohjelmistoa. Lisäksi esitellään, kuinka tämän oikean laitteen pohjalta pystytään luomaan laitetta jäljittelevä simulaatio ja kuinka luotua simulaatiota pystytään hallitsemaan graafisen käyttöliittymän kautta. Lisäksi käytännön osiossa on esitelty ja demonstroitu, kuinka yksikkötestejä pystytään hyödyntämään sovelluskehityksen aikana.

Opinnäytetyössä toteutettu laiteintegraatio, -simulaatio ja simulaation hallinta täyttivät toimeksiantajan asettamat tavoitteet. Käytännön toteutuksen luotiin toimiva pohja tuleville simulaation hallintanäkymille.

Asiasanat

simulointi, integraatio, Java, tilannekuva

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Jari Nykänen	Bachelor of Business Administration	September 2018

Thesis title

Simulation as a part of software development

45 pages
3 pages of appendices

Commissioned by

Observis Oy

Supervisor

Janne Turunen

Abstract

The objective of the thesis was to research how software simulations could be used as a part of software development and testing. A working device integration, device simulation and a simulation control view were developed based on this study. Thesis was done during the development of ObSAS Situation Awareness Software and the main programming language was Java.

Chapters two and three introduces what simulations were and what were benefits and downsides of using simulations. These chapters also defined the differences between simulations and emulations with examples. All used software, frameworks and techniques were introduced in these chapters.

Chapter four was the practical part of the thesis. First, it gives an overview of the ObSAS software architecture. After that the chapter had a short introduction about the Vaisala WXT536 weather station which was the main device used in this practical part. The objective was to first integrate this device to the ObSAS software, then to create a simulation imitating the behavior of the actual device and finally to create a usable simulation control page. Testing was also a major part of this chapter and how to use unit tests during the development process.

All the set objectives were accomplished, and the solution created in the practical part is currently in use in Observis Oy. This solution was a useful tool during the development of the ObSAS software and it will be the foundation for future simulation control views developed in the company.

Keywords

simulation, integration, Java, situational awareness

SISÄLLYS

1 JOHDANTO

Tämän opinnäytetyön tavoitteena on tutkia ja kehittää, kuinka toteutetaan laitteen integrointi eli sulauttaminen osaksi jo olemassa olevaa ohjelmistoa. Lisäksi tavoitteena oli tutkia, miten kustannustehokkailla Raspberry Pi –mikrotietokoneilla suoritettavia laitesimulaatioita pystytään hyödyntämään osana ohjelmistokehitystä ja -testaamista. Opinnäytetyön toimeksiantaja oli Observis Oy, joka on erikoistunut tilannekuvajärjestelmiin. Opinnäytetyössä kehitetyn ratkaisun tuli mahdollistaa oikean laitteen liittäminen osaksi yrityksen ohjelmistoa, sekä tarjota mahdollisuus oikean laitteen simuloinnille ja sen hallinnalle. Lisäksi ratkaisun tuli toimia pohjana tuleville simulaation hallintaohjelmille.

Luvussa kaksi kerrotaan lyhyesti ohjelmistotestaamisesta ja sen hyödyistä. Luvussa käsitellään myös yleisellä tasolla simulaatioita, sen tuomia etuja sekä rajoituksia. Luvussa on myös käytännön esimerkki siitä, millaisessa tilanteessa simulaation tekemistä kannattaa suunnitella ja mitä sen tekemissä tulee ottaa huomioon. Luvussa esitellään myös emulaatiot ja se, miten ne eroavat simulaatioista.

Luvussa kolme on esitelty käytännön ratkaisun kannalta oleelliset ohjelmistot ja viitekehykset. Nämä ovat Apache Felix, JUnit, Maven sekä OSGi. Luvussa on myös esittely käytännön toteutuksen fyysinen puoli ja siinä merkittävimmässä roolissa olivat Raspberry Pi –mikrotietokoneet. Tässä luvussa on esitelty kyseisen laitteen yleiset tiedot, mallit sekä perustelut sille, miksi se valittiin käytännön toteutuksen alustaksi.

Luvussa neljä käsitellään WXT536-sääaseman integrointi ohjelmistoon, simulaation ja sen hallintanäkymän toteuttaminen sekä käytännön toteutuksen arkkitehtuuri yleisellä tasolla. Luvussa esitellään vaiheittain, kuinka oikea, ennestään tuntematon laite saadaan toimimaan jo olemassa olevaan järjestelmän kanssa. Tämän jälkeen esitellään, kuinka oikean laitteen ja sen käyttöohjeiden pohjalta luodaan laitetta matkiva simulaatio ja hallintanäkymä. Käytännön toteutuksen esimerkkilaitteena oli Vaisalan WXT536-sääasema.

2 TESTAAMINEN JA SIMULAATIOT

Uusien ohjelmistopohjaisten ratkaisujen kehittäminen vaatii toimivaa testaamisen integraatiota osaksi tuotantoprosessia (Effective Software Testing 2006, 4–28). Testaamisen esteeksi voivat kuitenkin muodostua fyysiset rajoitukset, jos testattavana kohteena on ohjelmiston ja fyysisen laitteen yhtenäinen integraatio. Tämän integraation testaamiseen voidaan - oikeiden laitteiden sijaan - käyttää ohjelmallista laitesimulaatiota, joka jäljittelee oikean laitteen toimintaa. Tässä luvussa tutustutaan testaamiseen, simulointiin sekä sen tuomiin etuihin ja rajoituksiin.

2.1 Ohjelmistotestaaminen

Ohjelmistotestaus on työtä, jolla pyritään varmistamaan, että tuotettu ohjelmisto toimii juuri niin kuin sen olisi tarkoituskin. Homes (2013, 7) määrittelee testaamisen joukoksi aktiviteetteja, joilla on selkeästi määrätty tavoite. Näiden aktiviteettien tarkoituksena on tunnistaa ohjelmiston tai järjestelmän virheitä ja arvioida sen laatua käyttäjätyytyväisyyden saavuttamiseksi. Tästä huolimatta testaamiseen ei yleensä kiinnitetä riittävän paljon huomiota, vaan se on yleensä toimenpide, joka tehdään kiireellä ohjelmistokehityksen loppuvaiheessa. Testaaminen on myös hyvin yleisesti se osa ohjelmistokehityksessä, josta aikataulullisista tai kustannuksellisista syistä joudutaan leikkaamaan. Tämä leikkaaminen voi kuitenkin pidemmällä tähtäimellä nostaa kuluja jopa monikymmenkertaiseksi, jos jo käytössä olevaan ohjelmistoon joudutaan tekemään korjauksia, jotka olisi voitu välttää kunnollisella testaamisella kehitysvaiheessa. (Kasurinen 2013, 10-17.)

Riittävän aikaisessa vaiheessa tehdyillä ja hyvin suunnitelluilla testeillä voidaan tuottaa jopa 80 % vähemmän bugeja eli virheitä sisältäviä ohjelmia. Bugien korjaaminen kehitysvaiheessa vähentää ohjelman julkaisun jälkeisiä korjaustoimenpiteitä ja nostaa tuottavuutta jopa 45%. (IBM 2006, 28.) Voidaankin siis todeta, että kehitysvaiheen tehokkaalla ja hyvin suunnitellulla testaamisella voidaan säästää suuria summia kokonaiskustannuksissa. Tämä käy myös ilmi vuonna 2002 tehdystä tutkimuksessa, jonka mukaan puutteellinen testaaminen voi nostaa yhdysvaltalaisten ohjelmistotalojen tappiot 59,5 miljardiin dollariin vuodessa, kun otetaan huomioon sekä ohjelmistotaloille että

ohjelmistojen käyttäjille aiheutuneet haitat. (Tassey 2002, 23; Kasurinen 2013, 11.)

Homes (2011, 5) painottaa, että testaamisen tulisi olla kiinteä osa ohjelmistokehitystä aina aloituksesta ohjelman ylläpitoon ja sen käytöstä poistamiseen asti. Kasurinen (2013, 12) ja Homes (2011, 12) molemmat ovat myös sitä mieltä, että mitä aikaisemmassa vaiheessa ohjelmiston virheet huomataan, sitä halvempaa niiden korjaaminen on. Kasurisen (2013, 12) mukaan ohjelmiston virheen korjaaminen jo kehitysvaiheessa maksaisi vain 1–2 prosenttia niistä kuluista, mitä se maksaisi julkaisun jälkeen.

2.2 Simulaatiot ja niiden tuomat edut sekä haasteet

Simulaatioilla pyritään mallintamaan ja ennen kaikkea jäljittelemään todellisia asioita ja tapahtumia matemaattisen tai loogisen mallintamisen avulla. Räsänen (2004, 5) määrittelee simuloinnin seuraavasti: ”Simuloinnilla tarkoitetaan jonkin tuotteen, prosessin tai järjestelmän olennaisten osien tai kokonaisuuden jäljittelyä.” Simulointeja voidaan käyttää muun muassa opetustilanteissa, asiakkaiden käyttäytymisen mallintamisessa tietyillä parametreillä tai oikeiden laitteiden korvaamisessa ohjelmallisesti. (Räsänen 2004, 5-10.)

Simulaatioita voidaan tehdä monella eri tavalla. Esimerkiksi sairaanhoitoa opetettaessa käytetään nukkeja, joilla jäljitellään ihmisen anatomiaa ja ihmiskehon käyttäytymistä (JAMK 2015). Niitä suunniteltaessa ja toteutuksessa voidaan myös käyttää visuaalisia elementtejä, kuten animointeja. Vaarana niissä kuitenkin on se, että simulaation tekijä tai analysoija saattaa keskittyä enemmän animaatioihin kuin syntyneisiin tuloksiin. (Räsänen 2004, 13; van der Aalst & Voorhoeve 1995, 10.)

Simulointien käyttöön on monia hyviä syitä. Tietokoneiden prosessointitehon kasvun myötä simulaatioilla pystytään mallintamaan lähes mitä tahansa. (Dubitzky ym. 2011,13.) Matemaattisilla malleilla ja simulaatio-ohjelmilla pystytään esimerkiksi mallintamaan asiakasvirtojen käyttäytymistä tiettyinä kellonaikoina tai parantamaan tuotantoprosessin toimintaa. Tämä joustavuus tarjoaa myös mahdollisuuden samojen testien toistettavuuden. (Kasurinen 2013, 10-11; van der Aalst & Voorhoeve 1995, 2; Xiannong 2002.)

Yksi merkittävimmistä simuloinnin eduista on sen kustannustehokkuus. Oikeassa ympäristössä toteutettu testaaminen vaatii usein lisähenkilöstöä, oikeita laitteita, tai testit voivat olla niin massiivisia, ettei niitä pystytä edes toteuttamaan. Simuloinnilla voidaan puolestaan testata ja analysoida eri vaihtoehtoja ja tarvittaessa muuttaa simulaatioita halutunlaiseksi muokkaamalla simulaatioissa käytettyjä parametrejä. Tarvittaessa jopa koko systeemin simulointi on mahdollista. (Räsänen 2004, 11; van der Aalst & Voorhoeve 1995, 4; Xiannong 2002.) Räsänen (2004, 11) nostaa myös esille simulaatioiden ajallisen näkökulman. Niillä voidaan säästää aikaa, ja tietokoneella tehtäviä simulointeja voidaan tehdä ajasta riippumatta.

Ohjelmallisia simulaatioita voidaan myös hyödyntää opetuskäytössä. Hyvänä esimerkkinä voidaan pitää lentokonesimulaattoreita, joilla voidaan kouluttaa lentäjiä asettamatta kuitenkaan ketään vaaraan. Simulaatiota voidaan käyttää myös kriittisten laitteiden ja laitekokonaisuuksien, kuten esimerkiksi rautateiden kiskojen hallintalaitteiden, testaamisessa ja turvallisuuden varmistamisessa. (Räsänen 2004, 11.)

Vaikka simulaatiot ovat oivallisia ja monipuolisia työkaluja testaamisen avuksi, on niilläkin omat haittansa ja rajoitteensa. Van der Aalst ja Voorhoev (1995, 5) sekä Räsänen (2004, 11) nostavat molemmat esille ajan. Vaikka simulaatiolla säästettäisiin aikaa, menee itse simulaattorin tekemiseen käyttötarkoituksesta ja monimutkaisuudesta riippuen aikaa ja rahaa. Lisäksi simulaation loppuun suorittaminen ja luotettavien tulosten saaminen saattaa monimutkaisemmissa tapauksissa viedä pitkään. Näiden tulosten arvioinnissa ja tulkinnassa tulee myös olla erittäin tarkkana, jotta ollaan täysin varmoja tulosten oikeellisuudesta. (Van der Aalst & Voorhoev 1995, 5; Xiannong 2002.)

2.3 Milloin simulointi on järkevää?

Ennen simuloinnin tekemistä on hyvä miettiä, onko järkevää käyttää resursseja simulaation tekemiseen varsinkin silloin, jos sen kohde on helposti päätteltävissä tai laskettavissa ilman erillisiä simulaatioita (Banks & Gibson 1997, 1). Kuten aikaisemmin mainittu, prosessina simulaatioiden tekeminen on hyvin samanlainen kuin ohjelmistokehitys, jossa tarvitaan useita välivaiheita. Näiden

välivaiheiden läpikäyminen vaatii resursseja ja osaamista, jotka saattavat nostaa simulaation kustannukset liian suuriksi suhteessa saatavaan hyötyyn. Varsinkin monimutkaisempia simulaatioita suunnitellessa tulee olla selvillä se, onko simulaatioon saatavilla projektin onnistumisen kannalta riittävästi oikeanlaista dataa. Tämä onkin Banksin ja Gibsonin (1997, 3) mukaan yksi simulaation kriittisimmistä osista, ja toteutusvaihetta ei missään nimessä kannata aloittaa ennen kuin on varmistettu datan saatavuus. Pelkkä datan saatavuus ei kuitenkaan riitä. Simulaation tuottaman datan tulee myös olla oikeaa ja virheetöntä. Tämä korostuu varsinkin silloin, jos oikeita laitteita korvataan kehitysympäristössä ohjelmallisilla simulaatioilla. Virheellinen data saattaa pahimmillaan aiheuttaa jopa ihmishenkiä vaaraan, jos simulointi ei toimi täsmälleen oikein.

Simulaation teko kannattaa silloin, jos sillä saavutettavat tulokset ja säästöt ovat suurempia kuin sen tekemiseen menevät resurssit. Säästöjä miettiessä tulee myös ottaa huomioon projektin mahdolliset takarajat. Simulaation tarpeellisuus tulee kyseenalaistaa silloin, jos sen tekemiseen ei ole riittävästi aikaa ja projektiin käytetty aika venyy. Myöhästyminen tarkoittaa hyvin usein tappioita niin ohjelmistoa ja simulaatiota tekevälle yritykselle kuin asiakkaallekin. (Banks & Gibson 1997, 2-3.)

Kuitenkin jos kohteena on esimerkiksi iso tuotantoketju tai järjestelmä voi simulaatio olla järkevä ratkaisu kuluista huolimatta. Simuloinnilla voidaan pienentää monimutkaisen järjestelmän tuomia riskejä paljastamalla mahdolliset ongelmakohdat. Näiden ongelmakohtien huomaaminen simulaatiossa voi tarkoittaa huomattavia säästöjä projektin myöhemmässä vaiheessa. Simulaatio toimii työkaluna myös järjestelmän suorituskyvyn testaamisessa ja analysoinnissa. (Banks & Gibson 1997, 2-3; Wardman & Kim 1997; Xiannong 2002.)

2.4 Simulaatiot osana ohjelmistokehitystä ja testausta

Tämän opinnäytetyön pääpisteenä on tietokoneilla suoritettavat simulaatio-ohjelmat. Räsäsen (2014, 13) mukaan simulaatioiden tuotantoprosessi on hyvin samankaltainen kuin ohjelmistotuotannossa. Tuotantoprosessissa voidaan käyttää esimerkiksi vesiputousmallia, mutta prosessiin voidaan soveltaa myös muitakin tuotannon malleja. Yhteinen lähtökohta tuotantomallista riippumatta

on idea, jonka pohjalta voidaan tehdä määrittely, suunnittelu, toteutus, testaus ja käyttöönotto. (Räsänen 2004, 13.)

Simulaatio-ohjelman käyttö on hyvä vaihtoehto varsinkin silloin, kun tarvitaan ohjelmiston ja laitteen välinen integraatio. Tämän integraation toteutuksen testaaminen voi osoittautua hankalaksi, jos tarvittavat laitteet ovat suurikokoisia, kalliita tai hankalasti saatavilla. Myös Räsänen (2004, 11) toteaa, että ”- - monet oikeasti toteutettavat tilanteet voivat olla suuruusluokaltaan hankalia tai suorastaan mahdottomia toteuttaa.” Ohjelmallinen simulaatio nostaa arvoaan myös silloin, jos samaa laitetta tarvitaan useassa paikassa. Varsinkin kalliimpien laitteiden kanssa tämä tulisi hyvin nopeasti kannattamattomaksi kehitys- ja testiympäristön kannalta.

Luvussa 2.4 mainitut kohdat pätevät myös ohjelmistokehityksessä ja ohjelmistosimulaatioissa. Havainnollistetaan simulaation merkitystä esimerkin avulla. Oletetaan, että projektin alussa suunnitelmiin kuuluu sääasema, joka mittaa ilmankosteutta, -painetta ja -lämpötilaa. Tämän laitteen antama data tulisi integroida osaksi käyttöliittymää. Testiympäristössä halutaan oikean laitteen sijaan käyttää simulaattoria, koska se on taloudellisesti kannattavampaa kuin oikean sääaseman ostaminen. Tässä vaiheessa ei kuitenkaan ole tiedossa, minkälaista ja missä muodossa olevaa dataa sääasema tuottaa ja lähettää. Simulaattorin tekemistä ei siis ole järkevä aloittaa ennen kuin ollaan täysin varmoja siitä, mitä dataa joudutaan käsittelemään. Pahimmassa tapauksessa yritys tuhlaa tärkeitä resursseja simulaattorin tekoon, jota ei voida käyttää. Ja vaikka itse simulaattori toimisikin täysin oikean laitteen tavoin, mahdolliset muutokset projektin vaatimuksissa saattavat täysin poistaa sääaseman tarpeen lopullisesta tuotteesta.

Simulaatiot toimivat myös ohjelmistokehityksen testaamisen työkaluna. Jos ohjelmassa on käyttöliittymä tai toiminnallisuuksia, jotka toimivat laitteiden lähettämän datan pohjalta, on tärkeää, että kehitysympäristössä pystytään näkemään ja testaamaan tehtyjä muutoksia. Tämä testiympäristö mahdollistaa myös sen, että asiakkaalla jo olevaan tuotteeseen tulevia päivityksiä pystytään testaamaan paikallisesti ennen niiden toimittamista. Tällä voidaan varmistua, että toimitettava päivitys toimii myös oikeiden laitteiden kanssa.

2.5 Emulaatiot

Simulointien yhteydessä nousee hyvin usein esille termi emulaatio. Emulaatio ja simulaatio ovat kuitenkin kaksi eri asiaa (taulukko 1). Simuloinnissa pyritään mallintamaan jonkin oikean asian tai laitteen toimintaa niin tarkasti kuin mahdollista tiettyyn tarkkuuteen asti. Tietotekniikassa tämä tarkoittaa sitä, että järjestelmä pyrkii käyttäytymään samalla tavalla kuin oikean maailman kohde. Tämä simulointi ei kuitenkaan peri simuloitavan kohteen kaikkia piirteitä, vaan ”sääntöjä” pystytään tarvittaessa muuttamaan. Hyvänä esimerkkinä voidaan pitää jo aikaisemmin mainittua lentokonesimulaattoria, jossa kaikki tapahtuu täysin virtuaalisesti ja jonka asetuksia pystytään muuttamaan halutunlaiseksi. (Harper 2015; Räsänen 2004, 11.)

Simulaattori	Emulaattori
Järjestelmä, joka pystyy jäljittelemään toista järjestelmää tiettyyn pisteeseen asti	Järjestelmä, joka jäljittelee täysin toisen järjestelmän toimintaa
Ei peri jäljiteltävän järjestelmän kaikkia sääntöjä	Perii kaikki jäljiteltävän järjestelmän säännöt
Mallintaa ohjelmia ja tapahtumia	Kopioi toisen järjestelmän toiminnan

Taulukko 1. Simulaation ja emulaation erot

Emulaatio on puolestaan järjestelmä, joka käyttäytyy täysin samalla tavalla kuin toinen järjestelmä. Emuloinnissa peritään myös oikean maailman kohteen kaikki säännöt pienintäkin yksityiskohtaa myöten. Yksi parhaimmista esimerkeistä ovat vanhat pelikonsoliemulaatiot, jotka rakennetaan ohjelmallisesti täsmälleen samalla tavalla kuin alkuperäinen konsoli, jopa bugeja ja ongelmia myöten. Toinen esimerkki ovat virtuaaliympäristössä ajettavat käyttöjärjestelmät, kuten Android-emulaatiot, joilla voidaan testata mobiiliapplikaatioiden toimintaa ilman oikeaa laitetta. (Harper 2015.)

Tässä opinnäytetyössä keskitytään simulaatioihin, koska täsmälleen oikean laitteen toimintaa jäljittelevän emulaattorin tekeminen ei olisi järkevää eikä myöskään tarpeellista. Käytännön osuudessa keskitytään oikean tyyppisen laitteen datan mahdollisimman tarkkaan mallintamiseen eikä itse oikean laitteen emulointiin.

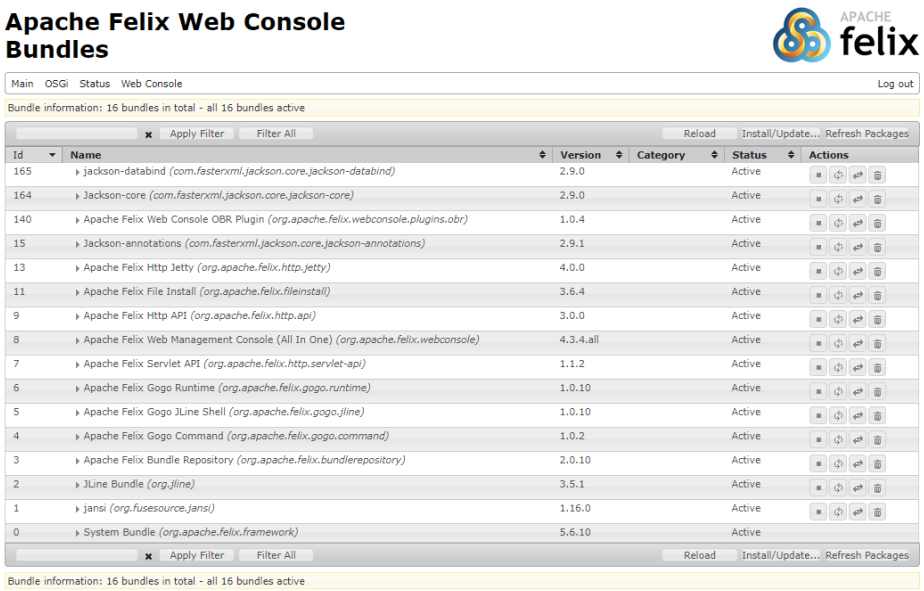
3 SIMULAATION TOTEUTTAMINEN JA TUOTANTOYMPÄRISTÖ

Tässä luvussa käydään läpi tämän opinnäytetyön käytännön osiossa käytettävät kehitysympäristöt. Ensimmäisessä ja toisessa luvussa on lyhyesti esitelty OSGi, Apache Felix, Maven sekä JUnit-testit. Näiden jälkeen esitellään Raspberry Pi –mikrotietokoneet, joita käytetään käytännön osion ohjelmistojen suorittamiseen.

3.1 OSGi ja Apache Felix

OSGi eli Open Services Gateway initiative on vuonna 1999 kehitetty avoimen lähdekoodin viitekehys, joka on kasvanut alkuperäisestä Java-teknologian kotiverkkoihin mukauttamisesta, suureksi markkinarajat ylittäväksi viitekehykseksi. Viitekehyksen suosiota kasvatti avoimiin lähdekoodeihin perustuvien ratkaisujen laaja leviäminen sekä isojen toimijoiden mukaantulo. (Gedeon 2010, 29.)

OSGi-viitekehyksen on kehitetty Java-ohjelmointikielelle. Sen tuoma etu on ajettavaan ympäristöön komponenttimallin kautta saatava dynaamisuus, jossa sen osia voidaan päivittää ja hallita erillisinä yksikköinä. OSGi:n avulla luodaan dynaamisia moduuliohjelmia, joita kutsutaan bundleiksi. Bundle on JAR eli Java Archive -kokoelma, joka sisältää Java-ohjelman tarvitsevat koodit, resurssit ja konfiguraatiot eli rakenteet. Esimerkiksi bundlen konfiguraatioissa voi olla vaatimuksena tietty Java-versio, jonka JAR-tiedosto tarvitsee toimiakseen. (Gedeon 2010, 31; Osgi Alliance 2017.)



Kuva 1. Apache Felix -hallintanäkymä

Näiden JAR-tiedostojen suorittamista varten on luotu avoimen lähdekoodin Apache Felix -ajoympäristö. OSGi-viitekehyksen lisäksi Apache Felix tarjoaa muun muassa riippuvuuksien hallintatyökalut, verkkosivupohjaisen hallintanäkymän (kuva 1) sekä bundlen lokitietojen käsittelyn. (Gedeon 2010,45-46.) Tämä lokitietojen käsittely on oleellisessa osassa tämän opinnäytetyön käytännön toteutuksessa.

Apache Felix valittiin testiympäristön alustaksi sen helppouden ja muokattavuuden takia. Käytännössä Apache Felix on Java-ohjelma, jonka avulla voidaan ajaa muita Java-ohjelmia (bundleja). Kaikki siihen asennettavat bundlet ovat tavallaan omia liitännäisiä, jotka pystyvät kommunikoimaan keskenään. Apache Felixiä voidaan myös käyttää suoraan Eclipsestä käsin. Tämä mahdollistaa Eclipsen sisäisen virheenkorjaustilan käytön JAR-tiedostojen tarkasteluun. Lisäksi Apache Felixiin oli mahdollista kehittää OSGi-bundlet päivittävä työkalu, mutta sen kuvaus ei kuulu tämän opinnäytetyön rajaukseen. Kaikki tämän opinnäytetyön käytännön toteutuksessa tehdyt JAR-paketit on suoritettu käyttämällä Apache Felix -viitekehystä.

3.2 JUnit ja Maven

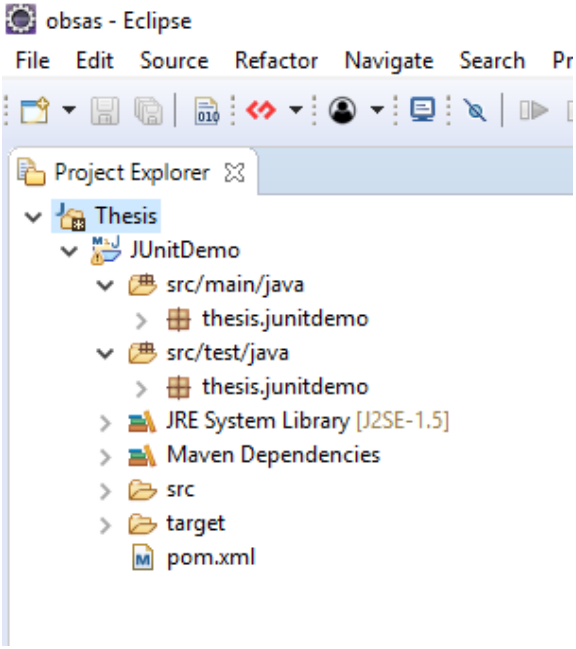
JUnit on Javalle kehitetty testauksen viitekehys, jonka avulla kehittäjät pystyvät tekemään helposti yksikkötestejä (Acharya 2014, 29). JUnittestejä voidaan

suorittaa muun muassa Eclipse kehitysympäristön avulla. Eclipse on myös päätyökalu tämän opinnäytetyön käytännön osiossa.

Yksikkötestaaminen parantaa merkittävästi koodin luotettavuutta, koska sen avulla pystytään huomaamaan mahdolliset koodivirheet jo kehitysvaiheessa. Tutorialspoint (2018) mukaan, yksikkötestien tekeminen myös pakottaa kehittäjät lähestymään ohjelmistokehitystä eri näkökulmasta, jossa ensin testataan ja sen jälkeen toteutetaan. Tätä näkökulmaa kutsutaan testivetoiseksi ohjelmistokehitykseksi (Test Drive Development). (Pietiläinen 2018, 6-8.) Tämä parantaa kokonaisuudessa ohjelmoijan tuottaman koodin laatua ja vähentää näin ollen virheiden syntymistä. (Pietiläinen 2018, 10; Tutorialspoint, 2018).

Yksikkötestaamista voidaan tehdä kahdella eri tavalla; käsin tai automatisoidusti. JUnit-viitekehys mahdollistaa testien automatisoinnin helposti osaksi Eclipseä ja Apache Maveniä. Apache Maven on projektinhallinta- ja pakkaustyökalu, jonka avulla voidaan rakentaa helposti ylläpidettävä ja uudelleenkäytettävä projektirakenne. (Acharya 2014, 90; Shah 2014, 19.)

Jotta automatisoituja JUnit-testejä voidaan suorittaa Mavenin ja Eclipsen avulla, tarvitsee Eclipse M2Eclipse-lisäosan. Tämän lisäosan avulla Mavenin komentoja pystytään suorittamaan suoraan Eclipsen omasta käyttöliittymästä. Maven puolestaa tarvitsee toimiakseen Javan kehitysympäristön JDK:n eli Java Development Kit:in. Näiden testien hyödyllisyyttä voidaan demonstroida luomalla uusi Eclipsen Maven-projekti, joka oletuksena luo kuvan 2 näköisen hakemistorakenteen.



Kuva 2. Uusi Eclipsen Maven-projekti

Tämän jälkeen luodaan uusi Java-luokka nimeltä *App* (kuva 3), johon luodaan yksinkertainen metodi *DemoMetodi(String jotainTekstia)*. Tähän metodiin lähetetään argumenttina jotain tekstiä, jonka jälkeen if- ja else-if-lauseilla tarkistetaan saadun viestin sisältö ja kirjoitetaan Eclipsen lokiin saatu viesti ja palautetaan sama viesti takaisin.

```
public class App {
    public static void main( String[] args ) {
        System.out.println( "Demo Maven Projekti" );
    }

    public String DemoMetodi(String jotainTekstia) {

        if(jotainTekstia.startsWith("tekstiä")) {
            System.out.println("Metodille lähetettiin: " + jotainTekstia);
        }

        else if (jotainTekstia.startsWith("Jotain muuta")) {
            System.out.println("Metodille lähetettiin: " + jotainTekstia);
        }

        return jotainTekstia;

    };
}
```

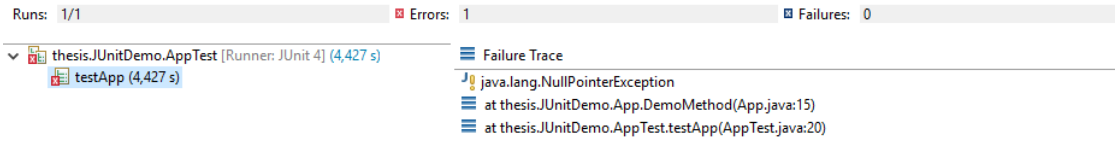
Kuva 3. JUnit-demo luokka *App*.

Seuraavaksi luodaan yksinkertainen JUnit-testi (kuva 4), jossa kuvan yksi metodiin lähetetään viestejä. Kaksi ensimmäistä viestiä lähettävät String-muotoisia viestejä, mutta kolmas lähettääkin pelkän null-viestin.

```
public class AppTest {

    public static final String VIESTI1 = "tekstiä";
    public static final String VIESTI2 = "Jotain muuta";
    public static final String VIESTI3 = null;

    @Test
    public void testApp() {

        App app = new App();

        app.DemoMetodi(VIESTI1);
        app.DemoMetodi(VIESTI2);
        app.DemoMetodi(VIESTI3);

    }

}
```

Kuva 4. JUnit-testi luokalle *App*.

Tämä testi aiheuttaa virheilmoituksen NullPointerException (kuva 5), koska kuvan yksi metodissa ei ole null-arvon tarkastavaa koodia. Tämä yksinkertainen esimerkki korostaa Tutorialspointin (2018) ajatusmaailmaa, jossa itse sovelluksen kehittämisen ja testaamisen tulisi tapahtua rinnakkain.



Kuva 5. JUnit-testin tulos.

Tämä testi suoritetaan joka kerta, kun kyseistä projektia pakataan Maven install -komennon avulla. Koko koodin pakkaus epäonnistuu (kuva 6), mikäli paketoinnin yhteydessä ajettavista testeistä jokin epäonnistuu esimerkiksi kuvan 5 tavalla.

```
Tests in error:
  testApp(thesis.JUnitDemo.AppTest)

Tests run: 1, Failures: 0, Errors: 1, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.772 s
[INFO] Finished at: 2018-08-12T17:04:26+03:00
[INFO] Final Memory: 15M/211M
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test (default-test) on project JUnitDemo: There are test failures.
```

Kuva 6. Epäonnistunut JUnit-testi maven install -komentoa suorittaessa

3.3 Raspberry Pi

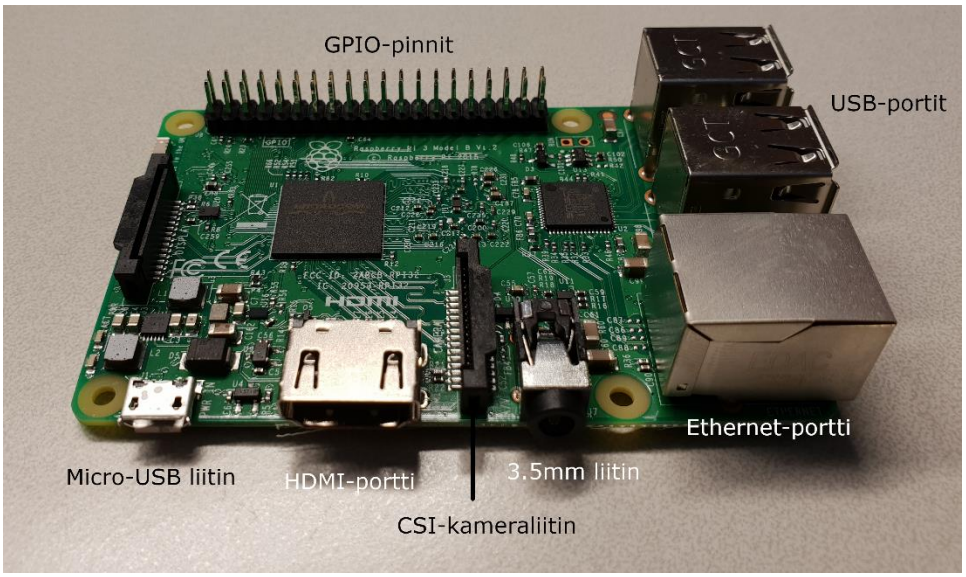
Raspberry Pi on Iso-Britannialaisen Raspberry Pi Foundationin kehittämä yhden piirilevyn mikrotietokone, jonka tarkoituksena on edistää tietotekniikan

opettamista kouluissa ja kehittyvissä maissa. Raspberry Pi:stä tuli maaliskuussa 2017 historian kolmanneksi eniten myydyin tietokone 12,5 miljoonalla myydyllä yksiköllä Nieldin (2017).

Raspberry Pi:stä on kuluttajille saatavilla viittä eri versiota: A+, B+, 2, 3 ja Zero. Nämä mallit eroavat toisistaan pääosin vain laskentatehon, USB-porttien lukumäärän ja verkkoyhteyksien osalta. Poikkeuksena Raspberry Pi Zero -versio, joka on suunniteltu mahdollisimman halvaksi muun muassa poistamalla kaikki normaalikokoiset USB-liitännät sekä Ethernet-portti. Näiden lisäksi on vielä olemassa Raspberry Pi Compute Module, joka on tarkoitettu teollisuuden käyttöön. (Upton & Halfacree 2016, 13-20.)

3.4 Raspberry Pi 3 B

Tämän opinnäytetyön teoriasisältö sekä käytännön osuus keskittyvät Raspberry Pi 3 B -malliin, joka julkaistiin helmikuussa 2016 (Martin 2016). Edeltävien versioiden tapaan Raspberry Pi 3 B tarjoaa hyvät liitäntämahdollisuudet pienestä koostaan huolimatta (kuva 7). Laitteesta löytyy muun muassa neljä USB-porttia, 40 pinninen GPIO (*General Purpose Input Output*)-liitäntä, HDMI-liitäntä ja Ethernet-portti. Laite on myös ensimmäinen Raspberry Pi, jossa on sisäänrakennettu tuki 2.4GHz langattomalle verkkoyhteydelle sekä Bluetoothille. (Upton & Halfacree 2016, 15-29.)



Kuva 7. Raspberry Pi 3 B liitännät

Tämän mallin prosessori toi ensimmäistä kertaa Raspberry Pihin tuen 64-bittiselle käyttöjärjestelmälle, verrattuna edellisiin malleihin, jotka tukivat vain 32-bittistä käyttöjärjestelmää. Uptonin ja Halfacreen mukaan (2016, 19) tämä arkkitehtuurin muutos parantaa ohjelmistojen yhteensopivuutta ja laitteen turvallisuutta.

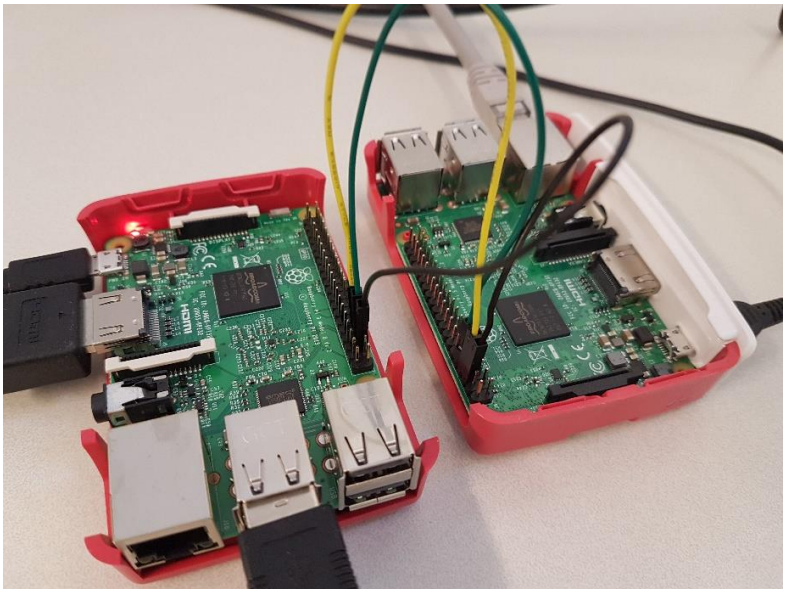
Raspberry Pi:n virallisena käyttöjärjestelmänä toimii Raspbian, joka on avoimeen lähdekoodiin perustuvan Linux-käyttöjärjestelmän muunnos. Toisin kuin suljetun lähdekoodin Windows- ja OS X -käyttöjärjestelmät, Linux tarjoaa käyttäjälle täyden vapauden muokata käyttöjärjestelmän lähdekoodeja. Laitteeseen on saatavilla myös muita Linux-ohjaisia käyttöjärjestelmiä, kuten Arch Linux, mutta myös esimerkiksi Windows 10 -käyttöjärjestelmän IoT Core -versio (Upton & Halfacree 2016, 18-23.)

3.5 Hyödyt ja mahdollisuudet

Raspberry Pi 3 on monessa suhteessa erittäin käyttökelpoinen ja kykenevä laite niin edullisena tietokoneena, palvelimena kuin työvälineenä ohjelmistokehityksessä. Tämän toteaa myös Jääskeläinen (2016), jonka mukaan ”Käytännössä vain oma mielikuvitus ja osaaminen rajoittavat sitä, mitä Raspberry Pi:llä voi tehdä.” Jääskeläinen (2016) myös mainitsee laitteen avulla tehdyistä projekteista muun muassa etäohjattavan kahvinkeittimen ja infrapunakameraa hyödyntävän mökkivahdin. Laitteen pieni koko mahdollistaakin sen käytön paikoissa, joissa tavallisen tietokoneen käyttö ei onnistuisi.

Näistä ja lukuisista muista erilaisista projekteista voidaankin päätellä, että merkittävä tekijä laitteen suosiolle on se, että se on kustannustehokas niin hankintahinnaltaan kuin käyttökuluiltaan. Raspberry Pi:n saa itselleen vain muutamalla kymmenellä eurolla, joten laitteen tuomien mahdollisuuksien hyödyntäminen ei vaadi suuria rahallisia investointeja. Uptonin ja Halfacreen (2016, 21) mukaan laite tarvitsee toimiakseen vain 5 V:n virtalähteen ja micro-USB-kaapelin. Jääskeläinen (2016) ehdottaa tarvittavan virran hankkimiseksi esimerkiksi halpaa aurinkopaneelia tai akkuja.

Kuitenkin suurin syy Raspberry Pi:n monikäyttöisyydelle on siihen sisäänrakennetut GPIO-pinnit. Näiden ohjelmoitavien pinnien avulla voidaan kommunikoida sellaisten laitteiden kanssa, joita ei normaalisti pystytä yhdistämään tavalliseen tietokoneeseen (liite 1). (Upton ym. 2016, 448.) Nämä pinnit mahdollistavat muun muassa yhteyden muodostamisen kahden Raspberry Pi –mikrotietokoneen välille (kuva 8). Yhteyden läpi on mahdollista lähettää esimerkiksi komentoja koneelta toiselle.



Kuva 8. Kahden Raspberry Pi:n välinen yhteys GPIO-liitännän kautta

Raspberry Pi:n kustannustehokkuus oli ratkaiseva tekijä sen valinnalle testiympäristön päälaitteeksi. Vaihtoehtona olisi ollut esimerkiksi virtuaalitietokoneiden käyttö. Virtuaalikoneet olisivat olleet todennäköisesti monta kertaa Raspberry Pi:tä tehokkaampia, mutta niillä olisi ollut mahdotonta testata esimerkiksi fyysistä johdon irrotusta. Myös mahdollisia ongelmatilanteita varten Raspberry Pi on järkevä valinta. Tämän nostaa esille myös Jääskeläinen (2016), jonka mukaan monesti vastaus ongelmaan löytyy nopealla google haulla. Tähän todennäköisenä syynä voidaan olettaa olevan Raspberry Pi:n ja yleisesti Linux-pohjaisten käyttöjärjestelmien verkkoyhteisön aktiivisen toiminnan ja avointen lähdekoodien saatavuuden.

3.6 Rajoitteet ja vaihtoehdot

Vaikka Raspberry Pi tarjoaa lukuisia mahdollisuuksia ja käyttötarkoituksia, silläkin on rajansa. Suurimpana rajoituksena on silkka prosessointitehon ja

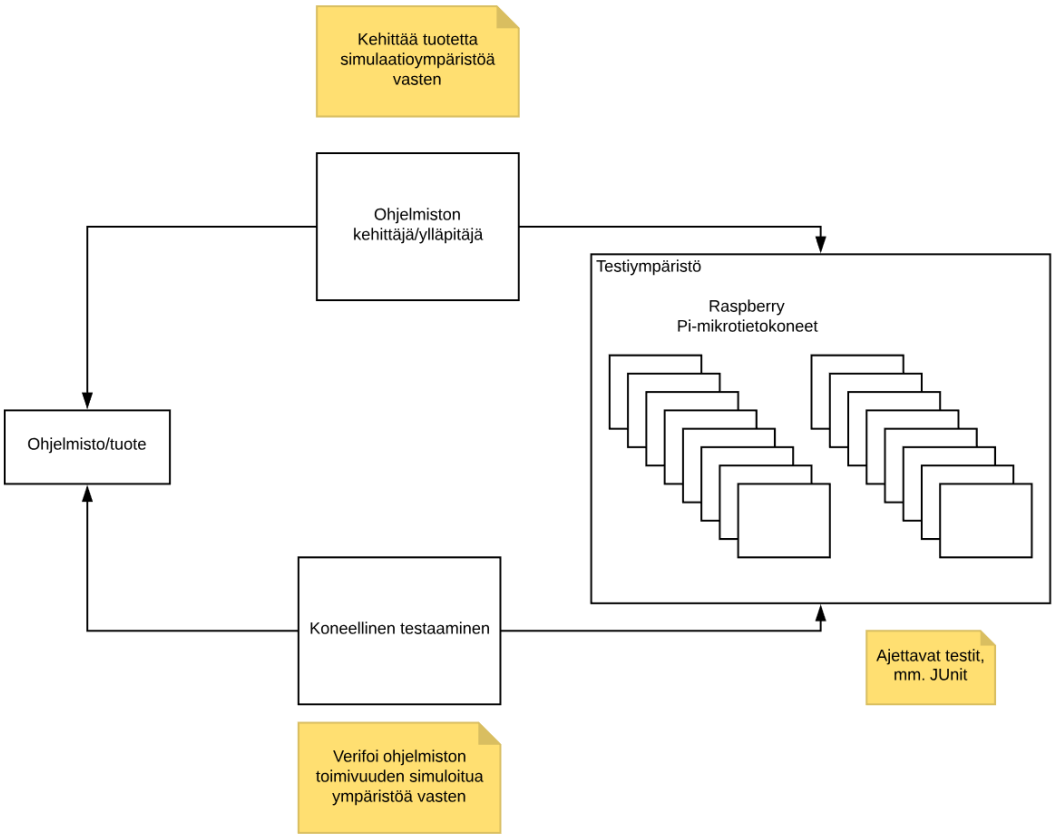
RAM- eli keskusmuistin puute. Tämä johtuu siitä, että laitteessa käytettävät osat pohjautuvat mobiiliteknologiaan. (Upton & Halfacree 2016, 20.) Tehon puute selittänee myös osan laitteen alhaisesta hinnasta. Luottokortin kokoiselta tietokoneelta ei voi odottaa täysikokoisen kannettavan tai pöytätietokoneen suoritustehoa.

Raspberry Pi:n lanseerauksen jälkeen markkinoille on ilmestynyt lukuisia kilpailijoita, kuten Orange Pi PC ja Odroid C2. Osa kilpailee hinnalla ja osa ominaisuuksilla, joita ei ole saatavilla Raspberry Pi:hin. Myyntivaltteja ovat esimerkiksi tehokkaammat prosessorit, prosessorin suuremmat kellotaajuudet ja serial ata -liitännän tarjoaminen kiintolevyn kiinnittämistä varten. Tehojen tarve riippuu kuitenkin käyttötarkoituksesta, sillä kehitysalustana käytettäessä suuremmista prosessoritehoista ei ole merkittävää hyötyä. (Jääskeläinen, 2016.)

Testiympäristössä päädyttiin kuitenkin käyttämään Raspberry Pi:tä, vaikka kilpailevia tuotteita olikin tarjolla. Jääskeläinen (2016) tuo esille kilpailijoiden selkeät puutokset käyttöjärjestelmien toimivuuden ja ongelmatilanteiden ratkaisun osalta. Nämä ongelmatilanteet haluttiin välttää mahdollisimman tehokkaasti, jotta testiympäristön kehitykseen käytetty aika menisi suurelta osalta ohjelmiston koodaamiseen eikä laite- tai käyttöjärjestelmän ongelmien selvittämiseen.

4 OBSAS LAITESIMULAATIO

Observis Oy:llä oli tarvetta kustannustehokkaalle ja helposti muokattavissa olevalle testiympäristölle ja sen hallintajärjestelmille. Testiympäristössä toteutettiin sisäverkkoon kytketyillä Raspberry Pi –mikrotietokoneilla. Näille asennettiin Apache Felix -viitekehys, jonka kautta laitesimulaatio-ohjelmat sisältäneet JAR-paketit suoritettiin. Näiden simulaatio-ohjelmien tuli vastata oikeiden laitteiden käyttäytymistä ja niiden antamaa dataa. Tämä simulaattoreiden lähettämä data palveltiin ohjelmallisesti eteenpäin käyttöliittymään. Laitesimulaatioissa tuli myös ottaa huomioon mahdollisten vikatilanteiden ilmeneminen.



Kuva 9. Observis Oy:n kehitysprosessin tulevaisuuden tavoite

Tämä laitesimulaatio- ja testiympäristö oli osa Observis Oy:n kehittämää ObSAS-tilannekuvaohjelmistoa ja sen kehitystä (Observis, 2018). Tulevaisuuden tavoitteena Observis Oy:llä on Raspberry Pi –mikrotietokoneiden, ohjelmallisten laitesimulaatioiden sekä yksikkö- ja ohjelmistotestaamisen sulauttaminen keskeiseksi osaksi yrityksen ohjelmistotuotantoa (kuva 9). Lisäksi tästä testiympäristöstä on tarkoitus jatkokehittää mahdollisimman automaattinen järjestelmä, jota vasten kaikki ObSAS-tuotteeseen kuuluvat ohjelmistot testattaisiin. Tämän opinnäytetyön tuloksena syntyvä simulaatio- ja testiympäristö toimii jatkokehityksen pohjana.

Projekti alkoi testiympäristön suunnittelemisella, jossa kartoitettiin, mitä laitteita testiympäristössä tulee simuloida. Simulaattoreiden kannalta oleellinen tieto oli myös simuloitavien laitteiden tarvitsemat yhteydet, sillä osa laitteista toimii TCP/IP-protokollan ja osa sarjaporttiyhteyden välityksellä.

Projektissa käytettiin hyvin monenlaisia laitteita, mutta tässä opinnäytetyössä käytetään esimerkkinä Vaisalan WXT536-sääasemaa, jonka integrointi ja si-

mulaation toteutus oli yksi tehtävistä. Kaikki esimerkkikoodit on otettu toiminnassa olevasta ohjelmistosta, minkä vuoksi kaikki luokat, metodit ja kommentit ovat englanniksi. Lähes kaikki käytännön osion koodeista on kirjoitettu Java-ohjelmointikielellä. Samanlainen ohjelma olisi ollut mahdollista tehdä esimerkiksi NodeJS:ää käyttäen, mutta se ei olisi noudattanut muuta ObSAS-ohjelmiston arkkitehtuuria. Tilan säästämiseksi koodeista on otettu suurin osa kommenteista pois.

4.1 WXT536 sääasema

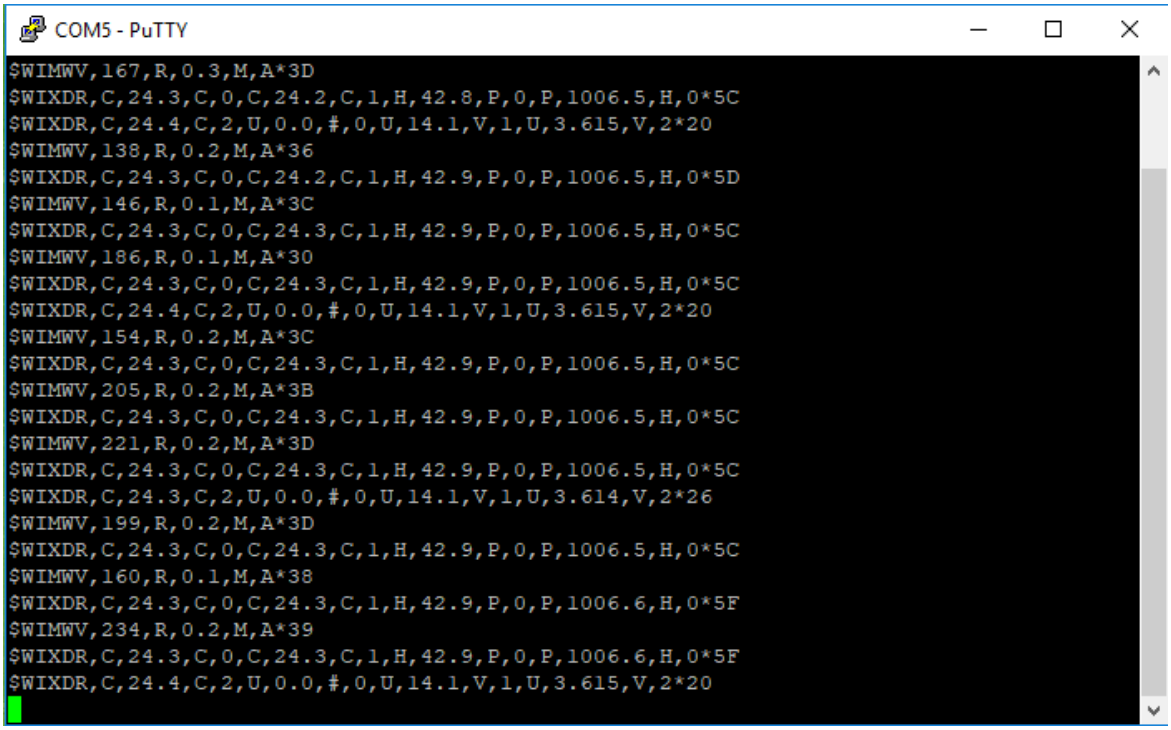
WXT536 on osa suomalaisen, mittauslaitteita valmistavan Vaisalan tuotevalikoimaa. WXT536 on monipuolisin malli Vaisalan WXT530-sarjan sääasemista. Sen avulla pystytään mittaamaan sadetta, tuulen suuntaa ja nopeutta, ilmanpainetta, lämpötilaa sekä kosteutta (Vaisala 2017, 2).

Kuva 10. Vaisalan WXT536-sääasema

WXT536-laitteen (kuva 10) datan lukeminen tässä käytännön toteutuksessa tapahtui käyttämällä sarjaliikenne-USB-muunninta, jolla RS-485-sarjaliikennestandardin mukainen liikenne saadaan helposti yhteensopivaksi Raspberry Pi:n kanssa. Raspberry Pi:n käyttöjärjestelmä tarjosi suoran tuen RS-485-standardin mukaiselle liikenteelle. Integroinnin kehitysvaiheessa käytettyyn

Windows 10:ä käyttävään kannettavaan sen sijaan jouduttiin asentamaan erilliset ajurit.

Ajurien asennuksen jälkeen vuorossa oli laitteen konfigurointi haluttujen tietojen lähettämiseksi. Tämä tapahtui laitteen mukana tulleen konfigurointityökalun avulla. Työkalulla pystytään muun muassa säätämään lähetettävän datan tiedot ja sen, kuinka nopeasti laite lähettää. Näistä oleellisin kohta oli asettaa laite lähettämään halutut mittausarvot sarjaliikenneporttiin. Konfiguraation jälkeen datan vastaanottamisen pystyi helposti testaamaan esimerkiksi PuTTy-sovelluksella, jonka avulla voidaan lukea WXT536:n käyttämää sarjaporttia.



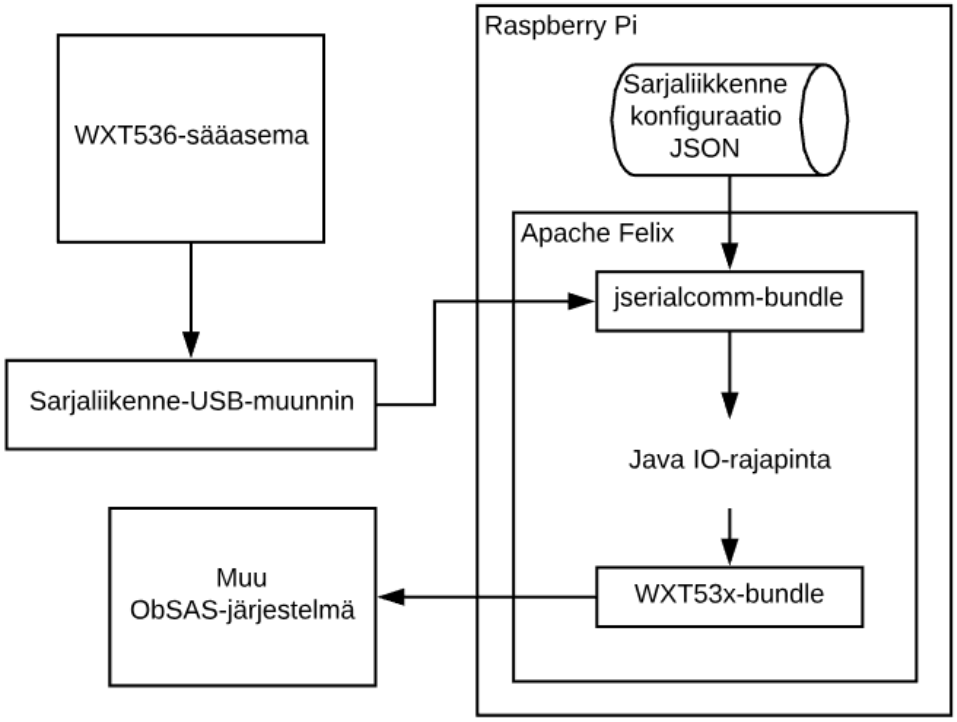
Kuva 11. WXT536 RS-485-sarjaliikennedata

Kuvasta 11 voidaan nähdä, että WXT536:n konfigurointi on onnistunut ja sarjaportista COM5 voidaan lukea mittatuloksia.

4.2 Simulaatio-arkkitehtuurin kuvaus ObSAS-ohjelmistossa

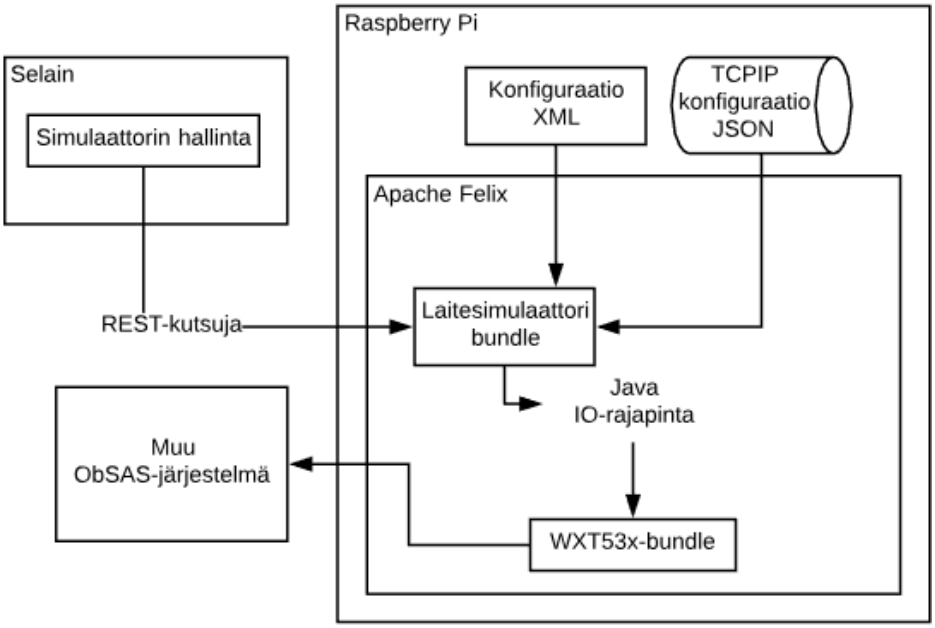
Ennen käytännön toteutuksen aloittamista oli hyödyllistä muodostaa yleiskuva siitä, kuinka tuleva toteutus täytyisi tehdä. Kuvassa 12 on kuvattu, kuinka prosessi etenee, kun käytetään oikeata WXT536-sääasemaa. Sääaseman lähettämät tiedot muutetaan sarjaliikenne-USB-muuntimen avulla Raspberry Pi:lle luettavaan muotoon. Raspberry Pi:llä on ajossa Apache Felix -

viitekehys, jonka sisällä kaikki JAR-paketit eli bundlet suoritetaan. Sarjaliikenne ObSAS-järjestelmässä vaatii liitteessä 2 olevan sarjaliikenne-JSON-tiedoston.



Kuva 12. Prosessikaavio oikeaa laitetta käytettäessä

Jserialcomm on Java-kirjasto, jonka tarkoitus on tarjota alustasta riippumattomat keinot sarjaliikenteen lukemiseksi (Fazecast, 2018). Java IO-rajapinta on puolestaan Javan sisäinen komponentti, jonka avulla voidaan kirjoitta ja lukea dataa (Jenkov 2014). Tämä rajapinta välittää jserialcommin lähettämän tiedon eteenpäin WXT53x-bundlelle, jossa tapahtuu itse viestin käsittely. Viestinkäsittelyn jälkeen saadut mittausarvot lähetetään muun ObSAS-ohjelmiston käsiteltäväksi.



Kuva 13. Prosessikaavio simuloitua laitetta käytettäessä.

Kuvassa 13 on esitelty sama prosessikulku, kun käytetään simuloitua laitetta. Jserialcommin tilalla on laitesimulaattori-bundle, joka sisältää kaikki ObSAS-ohjelmistossa simuloidut laitteet, mukaan lukien WXT536:n simulaation. Simulaattori tarvitsee konfiguraatio-XML-tiedoston ja liitteen 3 mukaisen TCPIP-protokollan mukaisen JSON-tiedoston. Lisäksi kuvan 12 oikean laitteen tilalla on simulaattorin hallinta, jonka kautta simuloituja arvoja pystytään muuttamaan REST-kutsujen kautta.

4.3 WXT536-integraatio

Monesti kehitysvaiheessa laiteintegraatiot joudutaan aloittamaan käänteisellä suunnittelemisella, eli luomalla ensin laitetta matkiva simulaatio ja tekemällä sen pohjalta varsinainen integrointi järjestelmään. Tämä johtuen siitä, että oikeita laitteita ei läheskään aina ole saatavilla kehitysvaiheessa. Observis Oy:llä oli kuitenkin saatavilla oikea WXT536-laite, joten ensin pystyttiin keskittymään laiteintegrointiin ja sen jälkeen simulaation tekemiseen.

Alkukonfiguraatioiden jälkeen (kuva 11) oli vuorossa varsinainen integrointiosio, joka alkoi syventymällä WXT530-sarjan ohjekirjaan. Ohjekirjan tärkeimpänä sisältönä oli laitteen lähettämän datan kyselyyn ja vastaanottamiseen tarvittavat tiedot, sekä datan sisällön selitykset. Nämä selitykset olivat välttämättömiä datan jäsentelijän eli parsijan tekemisessä.

Ennen varsinaisen parsijan tekemistä täytyi datan vastaanottamiselle ja käsittelylle tehdä tarvittavat alkuvalmistelut. Helpointa oli lähteä liikkeelle vastaanotettavan datan kirjoittamisesta muuttujiin.

```
public class WXT53xDeviceData {

    //WXT536 device data and possible units
    private Double minWindSpeed; // m/s, km/h, mph, knots
    private Double maxWindSpeed; // m/s, km/h, mph, knots
    private Double avgWindSpeed; // m/s, km/h, mph, knots
    private Double minWindDirection; // deg
    private Double maxWindDirection; // deg
    private Double avgWindDirection; // deg
    private Double airPressure; // hPa, Pa, bar, mmHg, inHg
    private Double airTemperature; // C, F
    private Double internalTemperature; // C, F
    private Double relativeHumidity; /// %RH
    private Double rainAccumulation; // mm, in
    private Double rainDuration; // s
    private Double rainIntesity; // mm/h, in/h
    private Double rainPeakIntensity; // mm/h, in/h
    private Double hailAccumulation; // hits/cm2, hits/in2, hits
    private Double hailDuration; // s
    private Double hailIntensity; // hits/cm2h, hits/in2h, hits/h
    private Double hailPeakIntensity; // hits/cm2h, hits/in2h, hits/h
    private Double heatingTemperature; // C, F
    private Double heatingVoltage; // V
    private Double supplyVoltage; // V
    private Double refVoltage; // V
    private String informationField; // aplhanumeric

    public WXT53xDeviceData() {
        super();
    }

    public Double getMinWindSpeed() {
        return minWindSpeed;
    }

    public void setMinWindSpeed(Double minWindSpeed) {
        this.minWindSpeed = minWindSpeed;
    }
```

Kuva 14. WXT536-muuttujat ja mahdolliset yksiköt

Kuvassa 14 WXT:n datalle luodaan oma Java-luokka nimeltä WXT53xDeviceData, jossa kaikille laitteen lähettämille arvoille luodaan oma muuttuja. Huomiona luokan nimen osa, joka on WXT53x eikä WXT536. Tämä johtuen siitä, että kaikki Vaisalan WXT530-sarjan sääasemat lähettävät samanlaista dataa, joten tämä kyseinen toteutus toimii kaikilla sarjan laitteilla. Jokaiselle muuttujalle luodaan omat asettava ja palauttava metodi eli yleisesti puhutaan getereistä ja settereistä. Kuvassa 14 on esimerkkinä luotu getteri ja setteri muuttujalle *minWindDirection*. Nämä metodit nimensä mukaisesti asettavat muuttujalle arvon ja hakevat muuttujalle asetetun arvon.

Parsijalle tehdään oma luokka nimeltä *WXT53xParser*, ja sen sisälle metodi nimeltä *parse(String msg)* (liite 1). Liitteessä 1 *parse(String msg)* metodille lähetetään parametrina WXT-laitteelta tullut viesti. Laitteen tila asetetaan nollaksi, joka järjestelmässä tarkoittaa, että laite on undefined-tilassa, kun parseri-luokka luodaan. Seuraavana suoritetaan viestin olemassaolon tarkistus. Jos metodille lähetetty String msg on tyhjä tai sitä ei ole olemassa, laite asetetaan virhetilaan. Muussa tapauksessa ajetaan metodi *initialize()*, joka luo uuden aikaisemmin esitellyn WXT53xDeviceData-olion ja asettaa sen parserTemplate-olioon. ParserTemplate-olio itsessään sisältää parsintaan tarvittavat yleiset metodit, kuten virheenhallinnat ja parsittujen viestien eteenpäin lähetyksen.

Tämän jälkeen luodaan tyhjä String[] array, johon tarkistusmetodit *checkWindMessage(String msg)* ja *checkPTUMessage(String msg)* palauttavat arvot, jos tarkistus on mennyt läpi (liite 1). Se, kumpaan metodiin viesti lähetetään, riippuu saapuvan viestin ensimmäisistä merkeistä, jotka ovat joko $WIXDR tai $WIMWV riippuen siitä, onko kyseessä PTU-viesti vai tuuleen liittyvät viesti. Esimerkkinä käytän PTU-viestiä, joka on seuraavanlainen:

$WIXDR,C,27.2,C,0,C,27.7,C,1,H,47.8,P,0,P,1006.5,H,0*5D.

```
private String[] checkPTUMessage(String msg) {

    if (msg == null || msg.length() < 3) {
        return new String[0];
    }

    String[] fields = msg.substring(0, msg.length() - 3).split(",");
    String[] res = new String[fields.length - 1];
    for (int i = 1; i < fields.length; i++) {
        res[i - 1] = fields[i];
    }
    return res;

}
```

Kuva 15. PTU-viestin pilkkominen arrayhin

Kuvassa 15 on esimerkki PTU-viestin (Pressure, Temperature ja Humidity) pilkkomisesta. Metodissa tehdään vielä uusi viestin tarkistus sisällön olemassaolosta ja viestin pituudesta. Mikäli viesti on tyhjä tai liian lyhyt, liitteen 1 fields-muuttujalle palautetaan tyhjä array. Muussa tapauksessa saatu viesti pilkotaan jokaisen pilkun kohdalta arrayn arvoksi lukuun ottamatta viimeistä

kolmea merkkiä. Tämän jälkeen luodaan uusi res array, jonka pituus määritetään fields arrayssä olevien muuttujien lukumäärän perusteella. Tällä mahdollistetaan se, että saapuvan viestin pituutta ei tarvitse tietää etukäteen. Tämän jälkeen luodaan for-silmukka, jossa poistetaan fields arrayn ensimmäinen arvo, koska sitä ei enää tämän jälkeen tarvita. Silmukan jälkeen palautetaan pilkottu res[] array takaisin liitteen 1 *parse(String msg)*-metodiin.

Tämän jälkeen on vuorossa itse viestin parsinta.

```
private int parsePTUMessage(String[] fields) {

    int state = EDeviceState.NORMAL;
    String tempUnit = "C";
    if (fields != null) {
        for (int i = 0; i < fields.length; i += 4) {
            String[] values = Arrays.copyOfRange(fields, i, i + 4);

            // Message for pressure, in hPa
            if (values[0].equals("P") && values[3].equals("0")) {
                Double airPressure = Double.parseDouble(values[1]);
                parserTemplate.getDeviceData().setAirPressure(airPressure);
                logger.log(level, "Pressure: " + airPressure.toString() + "hPa");
            }
        }
        logger.fine("commit");
        parserTemplate.commit();
    } else {
        logger.log(level, "Error parsing message");
        state = EDeviceState.FAILURE;
        parserTemplate.raiseError(new IllegalStateException("Error parsing message"));
    }
    return state;
}
```

Kuva 16. PTU-viestin parsinta

Kuvassa 16 laitteen tila asetetaan oletuksena normaaliksi ja asetetaan oletuslämpötilayksikkö celsiuksiksi. Jälleen yhden null-tarkistuksen jälkeen for-silmukalla käydään läpi fields array, neljän luvun osioissa ottamalla fields arraystä kopio values arrayhin. Tämä johtuu siitä, että WXT:n lähettämä mittausdata koostuu aina neljästä arvosta. Esimerkiksi yllämainitussa esimerkkiviestissä arvot ”P,1006.5,H,0” muodostavat ilmanpaineen mittausviestin. Tässä viestissä P on muuntimen lähettämän viestin tyyppi, joka tässä tapauksessa on paine. 1006.5 on muuntimen antama mittausarvo, H kertoo mitä yksikköä mittaus on ja 0 on viestin tunnus kyseiselle mittaukselle (Vaisala 2017, 110). Tätä mittausarvoa käytetään myös if-lauseen demonstroimisessa, jossa values arrayn arvot käydään tarkistamassa. Jos arrayn paikassa nolla löytyy merkki P ja arrayn paikasta kolme löytyy merkki 0, niin silloin tiedetään, että kyseessä on ilmanpaineesta kertova mittaustulos. Tällöin luodaan muuttuja Double airP-

ressure, jonka arvoksi asetetaan arrayn paikan yksi arvo. Tämän jälkeen parseTemplate:lta käydään hakemassa tämän kyseenomaisen laitteen datatyypit (kuva 14) ja asetetaan ilman paineen arvo muuttujaan. Lisäksi lokitietoihin tulostetaan viesti, joka kertaa mitä mitattiin, kuinka paljon ja millä yksiköllä. Samanlaiset if-lauseet löytyvät jokaiselle kuvassa 14 määritellyille muuttujille.

Kun for-silmukka on saatu valmiiksi, kirjoitetaan lokiviesteihin ilmoitus ja kutsutaan parseTemplaten *commit()*-metodia, joka lähettää viestit eteenpäin. Mikäli parsimisessa tapahtui virheitä, esimerkiksi saatu viesti oli virheellinen eikä vastannut mitään annettuja ehtoja, asetetaan laite virhetilaan seuraavaan parsintaan asti.

WXT-laitteen lähettämissä tuulen mittaustiedoissa on PTU-viesteistä poiketen myös oma merkkinsä tiedon oikeellisuuden tarkistukselle. Tuulen mittaustietojen viesti on seuraavanlainen: ”$WIMWV,229,R,0.2,M,A*35”. Viesti on validi eli pätevä, jos kyseisen viestin viimeinen kirjain on A. Viimeisen kirjaimen ollessa V on viestin mitta-arvoissa vikaa (Vaisala 2017, 104). Tässä tapauksessa laite asetetaan virhetilaan.

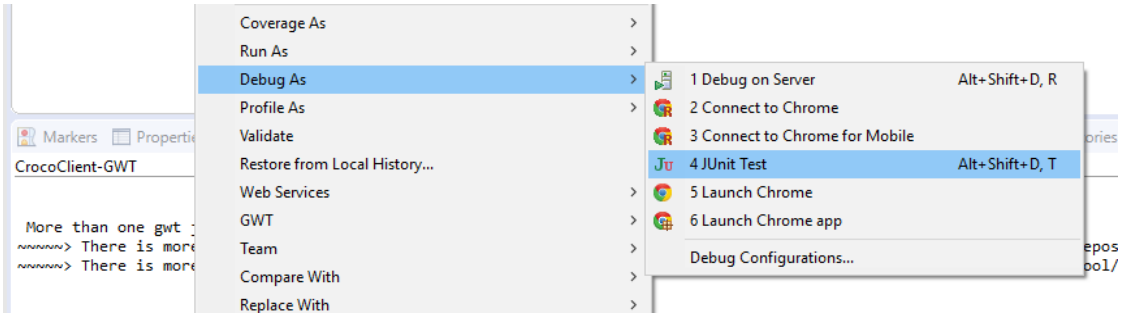
Tämän jälkeen parsimismetodit palauttavat laitteen tilan takaisin liitteen 1 *parse(String msg)*-metodille. Ennen tilan eteenpäin lähetystä tarkistetaan, onko tuulitietojen parsinta aiheuttanut laitteeseen virhetilan. Tämän jälkeen laitteen tila lähetetään eteenpäin muun ohjelmiston käsiteltäväksi.

Kehitysvaiheessa JUnit testit helpottivat tekemistä huomattavasti. Testit mahdollistivat sen, että esimerkiksi parseria tehdessä ei ollut tarpeen jokaisen muutoksen jälkeen paketoida koodia uudeksi bundleksi, asentaa sitä Apache Felixiin ja lukea Felixin lokitietoja. Sen sijaan yksinkertaisella JUnit testillä pystyttiin tekemään nopeasti muutoksia koodiin ilman koodin paketointia.

```
public class ParserTests {

    private static final Logger logger = Logger.getLogger(ParserTests.class.getName());
    public static final String WINDMESSAGE = "$WIMWV,229,R,0.2,M,A*35";
    public static final String PTUMESSAGE = "$WIXDR,C,27.2,C,0,C,27.7,C,1,H,47.8,P,0,P,1006.5,H,0*5D";

    @Test
    public void testNullParse() {
        WXT53xParser parser = new WXT53xParser();
        parser.parse("");
    }

    @Test
    public void parserTest1() {
        WXT53xParser parser = new WXT53xParser();
        ParserTemplate<WXT53xDeviceData> template = parser.getTemplate();
        template.onParseReady(() -> {
            WXT53xDeviceData data = template.getDeviceData();
            logger.info(new Gson().toJson(data));
        }).onError(e -> {
            logger.log(Level.SEVERE, e.getMessage(), e);
            assert (false);
        });
        parser.parse(PTUMESSAGE);

    }

}
```

Kuva 17. JUnit-testi

Kuvassa 17 olevalla JUnit-testillä pystyttiin helposti testaamaan aikaisemmin kuvatun parserin toimintaa. Testiin määriteltiin muutama erilainen viesti. Tässä esimerkissä on kaksi erilaista testiä, joiden toimintaperiaate on sama. Molemmissa luodaan uusi WXT53xParser luokan ilmentymä. Näin ollen voidaan kutsua WXT53xParser-luokan metodia *parse(String msg)* ja lähettää siihen luokan alussa määriteltyjä viestejä. *testNullParse()* testi lähettää parserille pelkän tyhjän viestin, joka aiheuttaa liitteen 1 mukaisesti virhetilan. Seuraavassa testissä parserille lähetetään kaikki WXT530-sarjan tukemat mittaustulokset kerralla.



Kuva 18. JUnit testin suorittaminen

Nämä testit voidaan suorittaa Eclipsessä debug- eli virheenkorjaustilassa, jolloin koodin suorittamista voidaan tauottaa halutuista kohdista pysäytyspisteiden eli breakpointtien avulla (kuva 18).

Virheelliset muutokset esimerkiksi parserissa estävät projektin paketoinnin. Tämä on hyödyllistä varsinkin silloin, jos tiedetään, että laitteen lähettämä data on aina tietynlaista.

Bundlen testaamista varten Felixin hakemistoihin oli lisättävä laitekonfiguraatio-JSON, jotta ObSAS-ohjelmiston laitekonfiguraattori osasi etsiä sääaseman.


```
{
    "serialDeviceConfiguration": {
        "address": "COM5",
        "connectionType": "SERIAL",
        "description": "Vaisala_WXT536",
        "deviceKey": "WXT536",
        "deviceMeasType": "WEATHER",
        "deviceModel": 1001,
        "name": "WXT536",
        "speed": 19200,
        "bits": 8,
        "parity": "N",
        "stopBits": 1,
        "serialPortType": "RS485",
        "measPoints": [
            {
                "measPointKey": "AIR_TEMPERATURE",
                "order": 0,
                "phenomenonType": "AIR_TEMPERATURE"
            },
            ...
        ]
    }
}
```


Kuva 19. Osa esimerkki WXT-536:en sarjaliikenne JSON-tiedostosta

Kuvassa 19 WXT-sääasemalle asetetaan tietyt sarjaliikenteen lukemista varten tarvittavat konfiguraatiot. Tärkeimpinä ovat sarjaportin osoite, sarjaliikenteen nopeus, bittien määrä, pariteetti, pysäytysbittien määrä sekä sarjaliikenteen protokolla. Lisäksi JSON-tiedostossa määritellään mittauspisteet, joita muu ObSAS-ohjelmisto tarvitsee. Tämän JSON-tiedoston sisältö riippuu siitä, onko käytössä sarjaliikennettä vai TCPIP-yhteyttä käyttävä laite. Koko JSON-tiedosto on katsottavissa liitteessä 2.

4.4 WXT536-simulaattori

Laiteintegroinnin jälkeen vuorossa oli simulaattorin lisääminen ObSAS-ohjelmiston kehittämistä varten tehtyyn laitesimulaatio-ohjelmistoon. Simulaation tekeminen oli välttämätöntä, koska WXT-sääasemia ei ollut käytössä kuin yksi kappale. Laitesimulaation vaatimuksena oli, että se pystyy lähettämään oikean laitteen kaltaisia viestejä satunnaisilla arvoilla.

Simulaattorin teko alkoi luomalla uusi luokka DeviceSimulator_WXT536, joka periytetään luokasta GenericDeviceSimulator. GenericDeviceSimulator-luokka pitää sisällään metodit, jotka ovat yhteisiä kaikille ObSAS-ohjelmistossa simuloitaville laitteille.

```
public class DeviceSimulator_WXT536 extends GenericDeviceSimulator{

    private static final Logger logger = Logger.getLogger(DeviceSimulator_WXT536.class.getSimpleName());
    private boolean stopped = false;
    private WXT536SimulatorConfiguration config = new WXT536SimulatorConfiguration();
    private InputStream in;
    private OutputStream out;
    private static final List<String> MESSAGES = Arrays.asList("$WIXDR","$WIMWV");
    private int counter = 0;

    public DeviceSimulator_WXT536(BundleContext context) {
        super(context);
    }

    private void startListening() {
        Thread sender = new Thread(this::mainThread);
        sender.start();
        try {
            sender.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        new Thread(new MessageReceiverSenderThread()).start();
        logger.info("Started simulation for device WXT365");
    }
```

Kuva 20. Osa DeviceSimulator_WXT536-luokkaa

Kuvassa 20 määritetään luokan alussa, onko simulaattori pysäytetty. Simulaattoria varten on myös luotava oma konfiguraatioluokka, jossa määritellään, kuinka usein laitesimulaatio lähettää viestejä. Tätä konfiguraatioluokkaa tarvitaan myös simulaattorin hallinnan toteuttamisessa. Tämän jälkeen luodaan syöttö- ja ulostulovirta in ja out sekä lista *MESSAGES*, johon lisätään WXT:n datan mukaiset etuliitteet. Näiden lisäksi tarvitaan myös laskuri *counter*, jota käytetään *MESSAGES*-listan arvojen vaihtamiseen.

```
private void mainThread() {
    while (!isStopped()) {
        String messageType = getMessage();
        String message = generateMessage(messageType);
        if (message != null) {
            try {
                out.write(message.getBytes(StandardCharsets.US_ASCII));
                out.flush();
                logger.warning(message);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        try {
            Thread.sleep(config.getSimulationInterval());
        } catch (InterruptedException e) {
            logger.warning("Got interrupted, stopping simulation.");
            setStopped(true);
        }
        if (Thread.currentThread().isInterrupted()) {
            setStopped(true);
        }
    }
}

public synchronized boolean isStopped() {
    return stopped;
}

private String getMessage() {
    String msg = MESSAGES.get(counter);
    counter = (counter + 1) % MESSAGES.size();
    return msg;
}
```

Kuva 21 mainThread-, isStopped- ja getMessage-metodit

Simulaattorille luodaan metodi *startListening(),* jossa luodaan simulaattorille uusi säie eli threadi kutsumalla luokan sisäistä *mainThread()*-metodia (kuva 21). Jos while-lauseessa oleva synkronisoitu metodi *isStopped*() palauttaa arvon false, *mainThread()*-metodi aloittaa simuloitujen viestien generoimisen. Ensimmäisenä kutsutaan *getMessage()*-metodia (kuva 21), jossa haetaan *MESSAGE*-listan arvo *counter*-muutujan avulla. Tälle muuttujalle asetetaan uusi arvo jakojäännöksen avulla, jonka myötä arvo on aina yhden suurempi kuin edellinen, mutta ei koskaan isompi kuin *Messages*-listan pituus. Tämän jälkeen metodi palauttaa listasta haetun etuliitteen. Tämä etuliite lähetetään eteenpäin *generateMessage(messageType)*-metodille.

```
public String generateMessage(String messageType) {

    SimulatorAlertConfiguration fullConf = getAlertConfiguration();
    if (fullConf != null) {
        config = fullConf.getWxtConfig();
    } else {
        logger.warning("No configuration, using default config.");
        config = new WXT536SimulatorConfiguration();
    }

    String message = null;

    Double minTemp = -20.0;
    Double maxTemp = 20.0;

    Double minHum = 0.0;
    Double maxHum = 50.0;

    Double minPres = 1000.0;
    Double maxPres = 1050.0;

    Double minWindDir = 1.0;
    Double maxWindDir = 200.0;

    Double minWindSpeed = 1.0;
    Double maxWindSpeed = 20.0;

    if(messageType.startsWith("$WIXDR")) {
        String randomTemp = getRandomDouble(minTemp, maxTemp);
        String randomAvgTemp = getRandomDouble(minTemp, maxTemp);
        String randomHumidity = getRandomDouble(minHum, maxHum);
        String randomPressure = getRandomDouble(minPres, maxPres);

        message =   messageType +
                ",C," + randomTemp + ",C,0" +
                ",C," + randomAvgTemp + ",C,1" +
                ",H," + randomHumidity + ",P,0," +
                "P," + randomPressure + ",H,0*5D\n";
    }

    else if (messageType.startsWith("$WIMWV")) {

        String validity = "A";
        String randomWindDir = getRandomDouble(minWindDir, maxWindDir);
        randomWindDir = randomWindDir.substring(0, randomWindDir.length() - 2);
        String randomWindSpeed = getRandomDouble(minWindSpeed, maxWindSpeed);

        message = messageType + "," + randomWindDir + ",R," + randomWindSpeed + ",M," + validity + "*35\n";
    }

    return message;
}
```

Kuva 22 generateMessage()-metodi

Kuvassa 22 luodaan ensin tyhjä String-muuttuja *messages* ja asetetaan minimi- ja maksimi arvot generoinneille. Tässä toteutuksessa simulaattorin vaatimuksena oli tuottaa vain kahdenlaisia viestejä, joten yksinkertaisella if-lauseilla tarkistetaan, minkä muotoinen etuliite metodille on välitetty. Jos kyseinen etuliite on muotoa ”$WIXDR”, luodaan PTU-viesti, muussa tapauksessa luodaan tuulen suunnan ja nopeuden kertova viesti. Sattumanvaraisia numeroita tuotetaan *getRandomDouble()*-metodin avulla.

```
private String getRandomDouble(Double minValue, Double maxValue) {

    DecimalFormat df = new DecimalFormat("###0.0");
    df.setDecimalFormatSymbols(new DecimalFormatSymbols(Locale.ENGLISH));
    Random random = new Random();

    String value = df.format(minValue + (random.nextDouble() * (maxValue - minValue)) );
    return value;
}
```

Kuva 23. Sattumanvaraisten tuplien luominen

Kuvassa 23 sattumanvaraisten numeroiden luomiseen käytetään Javan sisäistä numerogeneraattoria *Random*. Sen lisäksi tarvitaan Javan sisäinen desimaaliformatoija eli uudelleenjärjestäjä *DecimalFormat*, jolle annetaan lokalisointi-parametrit. Generoinnissa minimi- ja maksimiarvot sijoitetaan kaavaan, joka palauttaa sattumanvaraisen arvon annettujen lukujen väliltä, ja palauttaa arvon *String*-muodossa takaisin *generateMessage()*-metodille.

Generoinnin jälkeen arvot lisätään *messages*-muuttujaan oikeille paikoille. Tämän jälkeen valmis viesti lähetetään takaisin kuvan 21 metodille, josta se lähetetään eteenpäin muun ObSAS-ohjelman luettavaksi *out.write()*-metodilla. Lähetyksen jälkeen säie asetetaan odottamaan konfiguraatioluokassa määrittelyn ajan verran, joka on tässä toteutuksessa kuusi sekuntia. Simulaatio keskeytetään, jos säikeessä tapahtuu jotain odottamatonta, se lopetetaan tai mikäli muuttuja *stopped* asetetaan jossain vaiheessa true-tilaan.

Tämänkin simulaation ja generoinnin tekemisessä JUnit-testit olivat suuri apu.

```
public class RandomNumberTest {

    public static final String WINDMESSAGE = "$WIMWV";
    public static final String PTUMESSAGE = "$WIXDR";

    @Test
    public void testNumber() {

        DeviceSimulator_WXT536 simulator = new DeviceSimulator_WXT536(null);

        simulator.generateMessage(PTUMESSAGE);

    }

}
```

Kuva 24. Numeroiden generoinnin JUnit-testi

Kuvan 24 testillä pystytään helposti testaamaan generoinnin toimivuutta luomalla ensin uusi DeviceSimulator_WXT536-luokan ilmentymä, jonka *generateMessage(String messageType)*-metodille lähetään tietty etuliite (kuva 25).

Simulaattorin jälkeen kuvan 24 JSON-tiedosto tarvitsi muutoksia, koska kyseessä ei enää ollut oikea laite. Koska simulaatio on ohjelmallinen ”laite”, täytyy se määritellä käyttämään TCPIP-protokollaa.

```
{
    "networkDeviceConfiguration": {
        "portNumber": 9090,
        "deviceKey": "WXT536TCPIP",
        "name": "WXT365_SIMULATOR",
        "deviceMeasType": "WEATHER",
        "description": "Simulated WXT536",
        "connectionType": "TCPIP",
        "deviceModel": 1001,
        "address": "127.0.0.1",
        "measPoint": [
            {
                "measPointKey": "AIR_TEMPERATURE",
                "order": 0,
                "phenomenonType": "AIR_TEMPERATURE"
            },
            ...
        ]
    }
}
```

Kuva 25. Osa esimerkki WXT-536:n TCPIP-protokollan mukaisesta JSON-tiedostosta

Konfiguraatiossa simulaatiolle tulee määritellä oma portti ja osoite, jota kuunnellaan. Osoite määritetään osoittamaan paikallista isäntää, koska kyseessä on paikallinen simulaatio (kuva 25). Koko JSON-tiedosto on katsottavissa liitteessä 3. Tämän JSON:n lisäksi ObSAS-ohjelmisto tarvitsi vielä oman XML-tiedoston, jossa määritellään ObSAS-simulaattorin tarvitsemat parametrit.

```
<?xml version="1.0" encoding="UTF-8"?>
<DeviceConfiguration>
    <Device>
        <MeasType>WEATHER</MeasType>
        <Port>9090</Port>
        <Name>WXT536TCPIP</Name>
        <Model>1001</Model>
    </Device>
</DeviceConfiguration>
```

Kuva 26. Esimerkki WXT536:n simulaation tarvitsemasta XML-tiedostosta

Kuvan 26 XML-tiedostossa määritellään mittaustiedon tyyppi, porttinumero, laitteen nimi sekä ObSAS-ohjelmiston käyttämä laiteen id-numero. Konfiguraatioiden ollessa kohdallaan Apache Felixin lokitiedoissa näkyy onnistunut viestin generointi ja WTX53x-parserin pilkkoma viesti (kuva 27).

```
INFO: Got data: "$WIMWV,317,R,15.6,M,A*35"
elokuuta 12, 2018 3:17:40 IP. fi.observis.sas.deviceinterface.wxt53x.impl.WXT53xDeviceController lambda$bindParser$0
INFO: Parse ready.
elokuuta 12, 2018 3:17:41 IP. fi.observis.sas.deviceinfoprovider.runtime.DeviceManager logObservation
INFO: MEAS: WXT365_SIMULATOR - 1001:WIND_DIRECTION = 317.0 D
elokuuta 12, 2018 3:17:41 IP. fi.observis.sas.deviceinfoprovider.runtime.DeviceManager logObservation
INFO: MEAS: WXT365_SIMULATOR - 1001:WIND_SPEED = 15.6 m/s
```

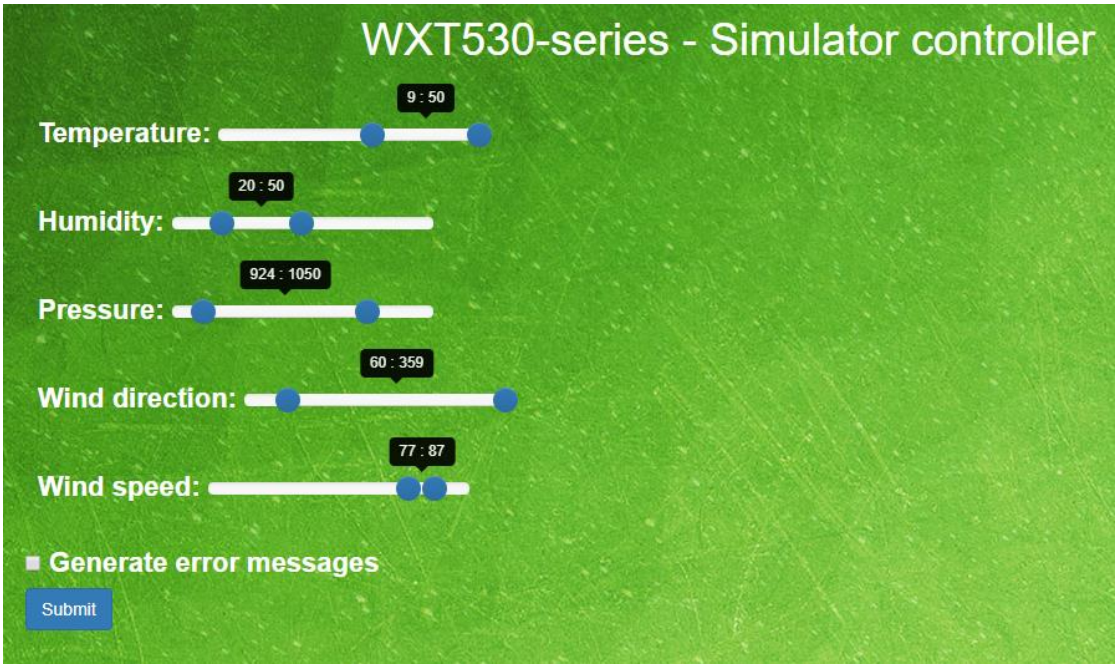
Kuva 27. Apache Felixin lokitiedot

Tätä simulaattoria voidaan nyt käyttää Raspberry Pi –mikrotietokoneilla, ObSAS-ohjelmiston vaatimissa käyttötarkoituksissa simuloimaan oikean laitteen toimintaa.

4.5 Simulaattorin hallinta

Simulaation ja laiteintegroinnin jälkeen ObSAS-ohjelmistoon tarvittiin myös simulaation hallintanäkymä, jossa graafisen käyttöliittymän kautta pystytään muuttamaan simuloitavien arvojen minimi- ja maksimiarvoja. Lisäksi käyttöliittymässä tulee olla mahdollisuus asettaa laite luomaan virheellisiä mittaustietoja.
Tätä toteutusta käytetään myös tulevaisuudessa pohjana muiden laitesimulaatioiden hallintanäkymiä tehtäessä.

Hallintanäkymässä käyttöliittymästä lähetetään REST-rajapintaan pyyntöjä, jotka käsitellään Javan servelt-arkkitehtuurin avulla.



Kuva 28. WXT530-sarjan simulaation hallintanäkymä

Pyyntöjen sisältönä on html-sliderien eli liukusäätimien sen hetkinen arvo (kuva 28). Näiden säätimien arvot pyydetään sivun latauksen yhteydessä palvelimelta GET-pyynnön avulla. Tämän pyynnön saadessaan servlet-koodin metodi *doGet()* hakee ennalta määrätyn tiedoston Felixin etc-hakemistosta. Jos tätä tiedostoa ei ole olemassa, metodissa luodaan uusi *SimulatorAlertConfiguration*-luokan instanssi eli ilmentymä.

```
//GET action for sending json data to client
@Override
public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    resp.setStatus(200);

    SimulatorAlertConfiguration simconfig = ConfigUtils.getConfiguration(
        context, configFileName, SimulatorAlertConfiguration.class
    );

    if(simconfig == null) {
        simconfig = new SimulatorAlertConfiguration();
    }

    String method = req.getParameter("method");

    if(method != null && !method.equals("") && method.equals("values")) {

        WXT536SimulatorConfiguration wxtConf = simconfig.getWxtConfig();
        resp.getWriter().write(mapper.toJson(wxtConf));
    }

}
```

Kuva 29. Servlet GET-pyynnön käsittely

Tämän jälkeen haetaan käyttöliittymästä lähetetystä pyynnöstä parametri nimeltä *method* (kuva 29). Tämän parametrin ollessa *values*, haetaan WXT-laitteen konfiguraatiot, joista luodaan JSON-tiedosto Gson-kirjastoa käyttäen (Google, 2016). Tämä luotu JSON-tiedosto lähetetään takaisin käyttöliittymään, jossa saadusta tiedosta luodaan uudet liukusäätimet ja niiden aloitusarvoiksi asetetaan datan mukaiset arvot (kuva 30).

```
$( document ).ready(function() {
      $.ajax({
          type: "GET",
          url: 'wxt/wxtServlet',
          data : {"method" : "values"},
          success: function(data){
              var result = JSON.parse(data);
              console.log(result);

              var tempSlider = new Slider('#temp', {
                  precision: 2,
                  tooltip: 'always',
                  value : [result.minTemp, result.maxTemp]
              });
              ...

          },
          error: function(data) {
              console.log(data);
          },
      });
  });
```

Kuva 30. JQuery:llä toteutettu REST-pyynnön lähetys käyttöliittymästä ja uuden liukusäätimen luominen saaduilla arvoilla

Uusien arvojen tallentaminen tapahtuu lähettämällä sliderit sisältävän formin eli lomakkeen tiedot POST-pyynnön avulla servletille. Kuvassa 31 metodi *doPost()* käsittelee tämän pyynnön. *doGet()*-metodin tapaan aluksi haetaan tai luodaan uusi konfiguraatio luokan instanssi. Tämän jälkeen *handleSetRequest*-metodi hakee WXT:n konfiguraatiot eli määrittelyt sekä saadun POST-pyynnön parametrin *temp*. Samanlaiset muuttujat löytyvät kaikille tämän opinnäytetyön sisällössä esillä olleille arvoille.

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    resp.setStatus(200);

    SimulatorAlertConfiguration simconfig = ConfigUtils.getConfiguration(
        context, configFileName, SimulatorAlertConfiguration.class
    );

    if(simconfig == null) {
        simconfig = new SimulatorAlertConfiguration();
    }

    handleSetRequest(req, simconfig);

    ConfigUtils.writeToFile(new File("etc/" + configFileName), simconfig);
    try {
        resp.sendRedirect("/simulate/wxt.html");
    }catch(Exception e) {
        logger.log(Level.WARNING, "Failed to redirect: ", e);
    }

}

private void handleSetRequest(HttpServletRequest req, SimulatorAlertConfiguration simconfig) {

    String temp = req.getParameter(TEMP);
    WXT536SimulatorConfiguration wxtConfig = simconfig.getWxtConfig();
    String[] tempMinMax = parseMinMaxValue(temp);

    String errorMessages = req.getParameter(ERRORMESSAGES);
    if(errorMessages == null) {
        errorMessages = "false";
    }

    setDouble(wxtConfig::setMinTemp, tempMinMax[0], wxtConfig.getMinTemp());
    setDouble(wxtConfig::setMaxTemp, tempMinMax[1], wxtConfig.getMaxTemp());
    setBoolean(wxtConfig::setErrorMessages, errorMessages);

}
```

Kuva 31. POST-pyynnön käsittely

Saatu arvo lähetetään metodille *parseMinMaxValue(String value),* jossa saatu teksti pilkotaan minimi- ja maksimi arvoiksi, sijoitetaan array muuttujaan ja palautetaan takaisin (kuva 32). Tämä saatu array sijoitetaan String[]-muuttujaan. Seuraava POST-kutsusta haetaan mahdollinen virhetietojen luomisesta kertova boolean-tyyppinen muuttuja. Jos kyseinen muuttuja on null, arvoksi ase-

tetaan ”false”. Tämä siksi, että HTML-lomakkeiden lähetyksessä ei oteta mukaan kuvan 28 valintalaatikon arvoa, mikäli sitä ei ole valittu. Tämän jälkeen asetetaan *tempMinMax*-muuttujan arvot *setDouble()*-metodilla. WXT:n konfiguraatioluokasta käydään hakemassa oletusarvo, jos *value*-muuttujan arvo on tyhjä.

```
private String[] parseMinMaxValue(String value) {

    String[] values = new String[2];

    if (value != null) {
        values = value.split(",");
        values[0] = values[0].trim();

        if(values.length > 1) {
            values[1] = values[1].trim();
        }
    }
    return values;
}

public void setDouble(Consumer<Double> setter, String value, Double configValue) {
    if(value != null && !value.isEmpty()) {
        try {
            setter.accept(Double.parseDouble(value));
        }
        catch (Exception e) {
            logger.warning(e.toString());
        }
    } else {
        try {
            setter.accept(configValue);
        }
        catch (Exception e) {
            logger.warning(e.toString());
        }
    }

}
```

Kuva 32. Minimi- ja maksimi arvon parsinta ja setDouble-metodi

Parsinnan ja uuden arvon asettamisen jälkeen käyttäjä uudelleenohjataan takaisin hallintanäkymän etusivulle, jolloin suoritetaan uudestaan kuvien 29 ja 30 GET-pyyntö.

Hallintaominaisuuden lisääminen aiheuttaa muutoksia kuvan 22 *generateMessage()*-metodiin, koska ObSAS-ohjelmistoon halutaan muokattavissa olevia arvoja käyttävä simulaatio. Kuvassa 22 olevassa toteutuksessa kaikki generoidut eli luodut arvot ovat staattisia eli muuttumattomia, joten generoitujen arvojen muuttaminen vaatisi koodin uudelleenkirjoittamista. Jotta simulaatiosta saadaan dynaaminen eli aktiivinen, täytyy metodi muuttaa käyttämään WXT:n konfiguraatioluokan arvoja (kuva 33).

```
public String generateMessage(String messageType) {

    SimulatorAlertConfiguration fullConf = getAlertConfiguration();
    if (fullConf != null) {
        config = fullConf.getWxtConfig();
    } else {
        logger.warning("No configuration, using default config.");
        config = new WXT536SimulatorConfiguration();
    }

    String message = null;
    WXT536SimulatorConfiguration wxtConf = fullConf.getWxtConfig();

    Double minTemp = wxtConf.getMinTemp();
    Double maxTemp = wxtConf.getMaxTemp();

    Double minHum = wxtConf.getMinHumidity();
    Double maxHum = wxtConf.getMaxHumidity();

    Double minPres = wxtConf.getMinPressure();
    Double maxPres = wxtConf.getMaxPressure();

    Double minWindDir = wxtConf.getMinWindDir();
    Double maxWindDir = wxtConf.getMaxWindDir();

    Double minWindSpeed = wxtConf.getMinWindSpeed();
    Double maxWindSpeed = wxtConf.getMaxWindSpeed();

    if(messageType.startsWith("$WIXDR")) {
        String randomTemp = getRandomDouble(minTemp, maxTemp);
        String randomAvgTemp = getRandomDouble(minTemp, maxTemp);
        String randomHumidity = getRandomDouble(minHum, maxHum);
        String randomPressure = getRandomDouble(minPres, maxPres);

        message =   messageType +
                ",C," + randomTemp + ",C,0" +
                ",C," + randomAvgTemp + ",C,1" +
                ",H," + randomHumidity + ",P,0," +
                "P," + randomPressure + ",H,0*5D\n";
    }

    else if (messageType.startsWith("$WIMWV")) {

        String validity = "A";
        if (wxtConf.getErrorMessages()) {
            validity = "V";
        }

        String randomWindDir = getRandomDouble(minWindDir, maxWindDir);
        randomWindDir = randomWindDir.substring(0, randomWindDir.length() - 2);
        String randomWindSpeed = getRandomDouble(minWindSpeed, maxWindSpeed);

        message = messageType + "," + randomWindDir + ",R," + randomWindSpeed + ",M," + validity + "*35\n";
    }

    return message;
}
```

Kuva 33. Viestin generointi WXT-konfiguraatioluokan avulla

Kuvassa 33 arvot määräytyvät aina hallintanäkymästä lähetettyjen arvojen mukaan. Tämä mahdollistaa muun muassa sellaisten tilanteiden testaamisen, jossa sääaseman arvot tippuvat asetettujen rajojen alapuolelle. Hallintanäkymästä (kuva 29) voidaan myös asettaa simulaattori luomaa viallisia viestejä, joiden avulla pystytään testaamaan ObSAS-ohjelmiston reagointia laitteen virhetiloihin.

5 PÄÄTÄNTÖ

Tämän opinnäytetyön tavoitteena oli tutkia, kuinka oikea laite integroidaan jo olemassa olevaan järjestelmään ja kuinka sen pohjalta luodaan simulaatio ja sen hallintatyökalu. Lisäksi tämän työn tulosten tarkoitus on toimia pohjana tulevalle testaamisen jatkokehitykselle Observis Oy:ssä. Kaikki edellä mainitut tavoitteet saavutettiin tämän opinnäytetyön käytännön toteutuksessa ja edellä esitelty WXT530-sarjan laiteintegraatio ja simulaattori ovat käytössä Observis Oy:llä.

Testaamisen kannalta tämän toteutuksen kaltainen testiympäristö on oivallinen työkalu, kun halutaan varmistua siitä, että yrityksen tekemät ohjelmistot ovat luotettavia ja toimivat oikeiden laitteiden kanssa. Testiympäristö ja koneellinen testaaminen mahdollistavat testaamisen toistettavuuden. Lisäksi testiympäristössä voitaisiin suorittaa oikeilta laitteilta nauhoitettuja tietoja, jolloin esimerkiksi pystytään selvittämään, miksi jokin tietty asia tai tilanne aiheutti järjestelmässä virheen. Raspberry Pi:n ja simulaattorien tarjoamien mahdollisuuksien avulla ObSAS-järjestelmää pystytään myös helposti ja kustannustehokkaasti esittelemään mahdollisille uusille asiakkaille.

Suoriuduin mielestäni tämän opinnäytetyön käytännön toteutuksesta hyvin ja työn aihe oli erittäin mielenkiintoinen. Suurimmat haasteet olivat laiteintegroinnin ja Java-ohjelmointikielen kanssa, mutta näiden kanssa työskentely ja ongelmien ratkominen kehittivät omaa ammattiosaamistani paremmaksi monellakin osa-alueella. Myös valitsemani toimintatapa sopi käytännön toteutuksen tekemiseen. Oli helppoa aloittaa tarkastelemalla järjestelmää yleisestä näkökulmasta ja sen jälkeen paneutua järjestelmässä syvemmälle aina koodin tasolle asti. Lisäksi sääaseman kanssa työskentely oli järkevää aloittaa tarkastelemalla, mitä tietoa laite lähettää, kuinka sen pohjalta tehdään simulaatio ja lopulta toteuttaa simulaation hallinta.

Olen myös tyytyväinen, että toteutukseni on yrityksellä käytössä ja että se vastaa annettuja vaatimuksia. Tavoitteena on olla jatkamassa kehitystyötä testiympäristön parissa, jotta siitä saadaan nykyistä monipuolisempi kehitystyökalu. Kuten aikaisemmin mainittua, tämän ympäristön päivittämiseksi olen

jo kehittänyt päivityksen automatisoivan työkalun, joka vie ympäristöä kohti täysautomaatiota.

LÄHTEET

Acharya, S. 2014. Mastering Unit Testing Using Mockito and JUnit. Birmingham: Packt Publishing.

Banks, J., Gibson, R. 1997. Don't Simulate When. Word-dokumentti. Saatavissa: http://site.iugaza.edu.ps/sagha/files/2010/02/Dont_Simulate_When.doc [viitattu 7.6.2018].

Effective Software Testing. 2006. IBM. PDF-dokumentti. Saatavissa: ftp://public.dhe.ibm.com/software/my/events/quality/pdf/Effective_Software_Testing.pdf [viitattu 5.6.2018].

Fazecast. 2018. jSerialComm. WWW-dokumentti. Saatavissa: http://fazecast.github.io/jSerialComm/ [viitattu 22.8.2018].

Gedeon, W. 2010. OSGI and Apache Felix 3.0 Beginner's Guide. Birmingham: Packt Publishing.

Google. 2016. google-gson. WWW-dokumentti. Saatavissa: https://github.com/google/gson [viitattu 22.8.2018].

Harper, C. 2015. What is Emulation? Benefits, Downsides and More. WWW-dokumentti. Päivitetty 28.12.2015. Saatavissa: https://www.maketecheasier.com/what-is-emulation/ [viitattu 10.6.2018].

Jenkov, J. 2014. Java IO Tutorial. WWW-dokumentti. Päivitetty 10.4.2014. Saatavissa: http://tutorials.jenkov.com/java-io/index.html [viitattu: 22.8.2018].

Jääskeläinen, O. 2016. Mikrobitin suuressa testissä Raspberry Pi ja 11 muuta korttitietokonetta. WWW-dokumentti. Päivitetty 20.10.2016. Saatavissa: https://www.mikrobitti.fi/2016/10/mikrobitin-testi-raspberry-pi-11-korttitietokonetta/ [viitattu: 22.5.2018].

Kasurinen, J. 2013. Ohjelmistotestaamisen käsikirja. Jyväskylä. Docendo.

Martin, J. 2016. Raspberry Pi 3 release date, price and specifications. Päivitetty 29.2.2016. WWW-dokumentti. Saatavissa: https://www.techadvisor.co.uk/new-product/desktop-pc/raspberry-pi-3-release-date-price-specifications-3635881/ [viitattu 21.5.2018].

Nield, D. 2017. The Raspberry Pi is now the third best-selling computer of all time. WWW-dokumentti. Päivitetty 18.3.2017. Saatavissa: https://www.techradar.com/news/the-raspberry-pi-is-now-the-third-best-selling-computer-of-all-time [viitattu 22.5.2018].

Observis. 2018. ObSAS. WWW-dokumentti. Saatavissa: http://obsas.fi/index.php [viitattu 21.8.2018].

OSGi Alliance 2017. Architecture. WWW-dokumentti. Saatavissa: https://www.osgi.org/developer/architecture/ [viitattu 18.8.2018].

OSGi Alliance 2017. Where to start. WWW-dokumentti. Saatavissa: https://www.osgi.org/developer/where-to-start/ [viitattu 18.8.2018].

Pietiläinen, N. 2018. Yksikkötestausohjeisto. Kaakkois-Suomen ammattikorkeakoulu. Tietojenkäsittelyn koulutusohjelma. Opinnäytetyö.

Räsänen, S. 2004. Verkko-opetuksen tietotekniikkaa – Simulaatio opetuksessa. WWW-dokumentti. Saatavissa: http://www.cs.uku.fi/tutkimus/publications/reports/B-2004-3.pdf [viitattu 30.5.2018].

Shah, S. 2014. Maven for Eclipse. Birmingham: Packt Publishing.

Tassey, G. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. PDF-dokumentti. Saatavissa: https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf [viitattu 5.6.2018].

Upton, E., Duntemann, J., Roberts, R., Mamtora, T., Everard, B. 2016. Learning Computer Architecture with Raspberry Pi. Hoboken, New Jersey, Yhdysvallat: John Wiley & Sons, Inc.

Upton, E., Halfacree, G. 2016. Raspberry Pi User Guide. Neljäs painos. Hoboken, New Jersey, Yhdysvallat: John Wiley & Sons, Inc.

User Guide. 2017. Vaisala. PDF-dokumentti. Saatavissa: https://www.vaisala.com/sites/default/files/documents/WXT530-Users-Guide-M211840EN.pdf [viitattu 4.8.2018].

Weather Transmitter WXT530 Series. 2017. Vaisala. PDF-dokumentti. Saatavissa: https://www.vaisala.com/sites/default/files/documents/WXT530-Datasheet-B211500EN-E.pdf [viitattu 4.8.2018].

van der Aalst, W., Voorhoeve, M. s.a. Simulation handbook. PDF-dokumentti. Saatavissa: http://bpmcenter.org/wp-content/uploads/reports/2000/BPM-00-04.pdf [viitattu 30.5.2018].

Wardman, K., Kim, D. 1995. When to Simulate. PDF-dokumentti. Saatavissa: https://thesystemsthinker.com/wp-content/uploads/pdfs/060303E.pdf [viitattu 7.6.2018].

Xiannong, M. 2002. Advantages and Disadvantages. WWW-dokumentti. Päivitetty 18.10.2002. Saatavissa: https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node3.html [viitattu 7.6.2018].

Xiannong, M. 2002. When is Simulation the Appropriate Tool?. WWW-dokumentti. Päivitetty: 18.10.2002. Saatavissa: https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node2.html [viitattu 7.6.2018].

Liite 1

OSA WXT53xParser-LUOKKASTA

```
public class WXT53xParser {

    private ParserTemplate<WXT53xDeviceData> parserTemplate = new ParserTemplate<>();
    private static final Logger logger = Logger.getLogger(WXT53xParser.class.getName());
    private static final Level level = Level.FINE;
    private WXT53xDeviceData data;
    private Boolean failure = false;

    public Boolean getFailure() {
        return failure;
    }

    public void setFailure(Boolean failure) {
        this.failure = failure;
    }

    public ParserTemplate<WXT53xDeviceData> getTemplate() {
        return parserTemplate;
    }

    private void initialize() {
        data = new WXT53xDeviceData();
        parserTemplate.setDeviceData(data);
    }

    String windSpeedDenominator = "M";

    public int parse(String msg) {

        int state = 0;

        try {
            if (msg == null || msg.equals("")) {
                state = EDeviceState.FAILURE;
                return state;
            }
            initialize();
            String[] fields = null;

            if (msg.startsWith("$WIMWV,")) {
                fields = checkWindMessage(msg);
                state = parseWindMessage(fields);
            } else if (msg.startsWith("$WIXDR,")) {
                fields = checkPTUMessage(msg);
                state = parsePTUMessage(fields);
            }

        } catch (Exception e) {
            state = EDeviceState.FAILURE;
            parserTemplate.raiseError(e);
        }

        if(getFailure() == true) {
            state = 3;
        }
        return state;
    }
```

Liite 2

WXT536-SÄÄASEMAN SARJALIIKENTEEN MUKAINEN-JSON


```
{
    "serialDeviceConfiguration": {
        "address": "COM5",
        "connectionType": "SERIAL",
        "description": "Vaisala_WXT536",
        "deviceKey": "WXT536",
        "deviceMeasType": "WEATHER",
        "deviceModel": 1001,
        "name": "WXT536",
        "speed": 19200,
        "bits": 8,
        "parity": "N",
        "stopBits": 1,
        "serialPortType": "RS485",
        "measPoints": [
            {
                "measPointKey": "AIR_TEMPERATURE",
                "order": 0,
                "phenomenonType": "AIR_TEMPERATURE"
            },
            {
                "measPointKey": "AIR_HUMIDITY",
                "order": 1,
                "phenomenonType": "AIR_HUMIDITY"
            },
            {
                "measPointKey": "AIR_BAROMETRIC_PRESSURE",
                "order": 2,
                "phenomenonType": "AIR_BAROMETRIC_PRESSURE"
            },
            {
                "measPointKey": "WIND_DIRECTION",
                "order": 3,
                "phenomenonType": "WIND_DIRECTION"
            },
            {
                "measPointKey": "WIND_SPEED",
                "order": 4,
                "phenomenonType": "WIND_SPEED"
            }
        ]
    }
}
```

Liite 3

WXT536-SÄÄASEMAN TCPIP-PROTOKOLLAN MUKAINEN JSON


```
{
    "networkDeviceConfiguration": {
        "portNumber": 9090,
        "deviceKey": "WXT536TCPIP",
        "name": "WXT365_SIMULATOR",
        "deviceMeasType": "WEATHER",
        "description": "Simulated WXT536",
        "connectionType": "TCPIP",
        "deviceModel": 1001,
        "address": "127.0.0.1",
        "measPoint": [
            {
                "measPointKey": "AIR_TEMPERATURE",
                "order": 0,
                "phenomenonType": "AIR_TEMPERATURE"
            },
            {
                "measPointKey": "AIR_HUMIDITY",
                "order": 1,
                "phenomenonType": "AIR_HUMIDITY"
            },
            {
                "measPointKey": "AIR_BAROMETRIC_PRESSURE",
                "order": 2,
                "phenomenonType": "AIR_BAROMETRIC_PRESSURE"
            },
            {
                "measPointKey": "WIND_DIRECTION",
                "order": 3,
                "phenomenonType": "WIND_DIRECTION"
            },
            {
                "measPointKey": "WIND_SPEED",
                "order": 4,
                "phenomenonType": "WIND_SPEED"
            }
        ]
    }
}
```