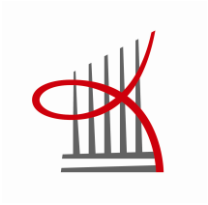


MITTAUSOHJELMISTON TESTAUSMENETELMIEN KEHITYS

Kalle Lahtinen

Opinnäytetyö
Helmikuu 2014
Tietotekniikan koulutusohjelma
Sulautetut järjestelmät ja elektroniikka

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Sulautetut järjestelmät ja elektroniikka

LAHTINEN, KALLE:
Mittausohjelmiston testausmenetelmien kehitys

Opinnäytetyö 37 sivua
Helmikuu 2014

Tässä opinnäytetyöraportissa käsitellään Novatron Oy:llä kesällä 2013 tehtyä ohjelmistotestauksen kehitysprojektia. Projektin tavoitteena oli ohjelmistotestausmenetelmien tehostaminen jatkuvan integraation ja testivetoisen ohjelmistokehityksen työskentelymenetelmien avulla. Testausmenetelmien kehityksen pitkän aikavälin tavoite on nopeuttaa uusien ohjelmistoversioiden julkaisua ja parantaa niiden laatua. Työssä selvitetään ohjelmistotestauksen yleiseen teoriaan liittyviä käsitteitä, mutta raportin pääpaino on Novatron Oy:llä havaittujen ohjelmistotestaukseen liittyvien ongelmien sekä kehitetyn testausjärjestelmän toiminnan ja rakenteen esittelyssä.

Työ aloitettiin tekemällä testattavalle ohjelmistoprojektille käännösautomaatio, joka versionhallintaan tehdyn muutoksen jälkeen automaattisesti kääntää projektin tietokoneella suoritettavaksi sovellukseksi, luo asennustiedostot ja siirtää ne Novatron Oy:n sisäiseen lähiverkkoon. Kun työssä käytettävään automatisointi- ja monitorointityökaluun (Jenkins CI) oli tutustuttu ja käännösautomaatio saatu toimimaan, siirryttiin projektissa ohjelmistotestausmenetelmien opiskeluun.

Kesän aikana testausautomaatiojärjestelmän testaustyökaluiksi valikoituivat Python-ohjelmointikieli, PyTest-testausmoduuli sekä Google test -yksikkötestikirjasto. Kehitystyö eteni siihen vaiheeseen, että testausjärjestelmän toiminta saatiin testatuksi. Automaattista testausta ei kuitenkaan vielä saatu osaksi jokapäiväistä ohjelmistojen kehitystyötä. Testausjärjestelmän perusrakenteet saatiin toimimaan, mutta testitapausten suunnittelu ja käyttöönotto vaativat lisätyötä.

Merkittävin saavutettu hyöty projektin ensimmäisen vaiheen jälkeen oli ohjelmistoprojektin käännösautomaation käyttöönotto, sillä sen avulla saatiin karsituksi ohjelmistokehittäjien päivittäisestä työstä aikaa vieviä ja itseään toistavia tehtäviä. Projektille asetetut ensimmäiset tavoitteet saavutettiin kesän aikana. Testausjärjestelmän jatkokehitys tulee olemaan osana Novatron Oy:n tuotekehitystyötä tulevaisuudessa.

Asiasanat: ohjelmistotestaus, mittausohjelmisto, ohjelmistotestauksen automatisointi, jatkuva integraatio, testivetoinen ohjelmistokehitys

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Embedded Systems and Electronics

LAHTINEN, KALLE:
The Development of Software Testing Methods for a Measurement Application

Bachelor's thesis 37 pages
February 2014

This thesis examines a development project for software testing methods done for Novatron Oy during the summer of the year 2013. The goal of the project was to develop and optimize the software testing methods used at Novatron Oy by modifying the software development towards continuous integration and test-driven development. The project's long-term goal is to create an automated software testing environment which makes the software development process faster and further improves the quality of released software versions. The basics of software testing theory are covered, but the key points of the report are explaining some of the problems of software testing at Novatron Oy and introducing possible solutions for these problems, including the different functions of the automated software testing environment.

The project began with creating an automated build environment for software projects. Whenever a developer commits a change to the subversion-system the automated build environment builds the software project and creates new installation files which are then shared to the local area network. After the build environment was ready and the used automation and monitoring tool (Jenkins CI) was familiar to the project's participants the development team began studying different methods of software testing.

During the development project Python, PyTest (a Python testing module) and Google test (a unit test library) were chosen to be used as the main tools of software testing in the testing environment. The development advanced to the point where the first automated tests were implemented to the testing environment. The automated tests did not yet become a part of the everyday software development. Implementing wide scale test cases to the testing environment require more work in the future.

The most significant advance achieved after the first phase of the testing environment's development was the automation of the software build process. Building a large software project is a complicated task which includes various different steps, taking a lot of time away from the actual software development. The set goals for the project were achieved and the development of the automated testing environment will continue in the future.

Key words: software testing, measurement application, automated software testing, continuous integration, test-driven development

SISÄLLYS

ERITYISSANASTO

CAN	Controller Area Network. Koneohjauksessa käytettävä kenttäväylä.
CIFS	Common Internet File System. Standardi tiedostojen siirtoon lähiverkossa ja internetissä.
DLL	Dynamic Link Library. Tiedostomuoto, jossa on suoritettavaa ohjelmakoodia. Tavallisen suoritustiedoston sijaan muut ohjelmat ja kirjastotiedostot voivat kutsua DLL-tiedostossa olevia toimintoja.
FTP	File Transfer Protocol. Tiedonsiirtoprotokolla tiedostojen siirtoon tietokoneiden välillä.
GNSS	Global Navigation Satellite System. Yleisnimitys satelliittipaikannustekniikalle.
GPS	Global Positioning System. Yhdysvaltain puolustusministeriön kehittämä ja rahoittama satelliittipaikannusjärjestelmä.
IP	Internet Protocol. Internet-protokolla, jonka avulla IP-tietoliikennepaketit liikkuvat halutuille tietokoneille.
Ketterät menetelmät	Joukko ohjelmistotuotannon työskentelymenetelmiä, joille on yhteistä valmiiden ohjelmistojen nopea tuotanto.
RTK	Real Time Kinematic. Satelliittipaikannuksessa käytetty tarkkuusmittausmenetelmä.
SFTP	SSH File Transfer Protocol. Tietoliikenneprotokolla salattuun tiedonsiirtoon.
TCP	Transmission Control Protocol. Tietoliikenneprotokolla tietokoneiden välisten yhteyksien luomiseen internet-verkossa.
Testikattavuus	Tunnusluku, joka kertoo, kuinka suuri osa ohjelmistosta on testattu, eli kuinka suuri osa ohjelmiston kaikista eri tiloista on käyty läpi.
Testitapaus	Yksittäinen ohjelmistotestauskokonaisuuden testi.
XML	Extensible Markup Language. Merkintäkieli, jossa tiedon merkitys on kuvattavissa tiedon sekaan.

1 JOHDANTO

Tämän opinnäytetyön tarkoituksena oli kehittää Novatron Oy:n tuoteperheen ohjelmistoille helposti muokattava ja laajennettava testausautomaatiojärjestelmä. Testausjärjestelmän haluttiin toimivan jatkuvan integraation ja testivetoisen ohjelmistokehityksen työskentelymenetelmien periaatteiden mukaisesti. Tavoitteena oli kehittää yrityksen ohjelmistotestausta tehokkaampaan suuntaan. Työn suunnittelu aloitettiin toukokuussa 2013 ja järjestelmän perusrakenteet saatiin valmiiksi kesän 2013 aikana.

Ohjelmistotestauksen merkitys ohjelmistotuotannossa kasvaa sitä suuremmaksi, mitä laajempi kehitettävä ohjelmistokokonaisuus on. Ohjelmistojen toiminnallisuudet pystytään testaamaan hyvin pitkälle täysin manuaalisesti. Ohjelmiston laajentuessa tuotekehityksessä usein huomataan, että joitain testaukseen liittyviä toimintoja olisi järkevää automatisoida työnteon tehostamiseksi.

Työn alussa perehdytään lyhyesti ohjelmistojen testaukseen liittyviin käsitteisiin ja toimintatapoihin, testattavan järjestelmän ominaisuuksiin sekä Novatron Oy:llä aikaisemmin käytössä olleisiin ohjelmistotuotannon ja testauksen menetelmiin. Lisäksi raportissa käsitellään testausmenetelmien kehitysprojektille ennakkoon asetettuja tavoitteita ja ohjelmistotestauksen automatisoinnin hyötyjä ja haittoja jokapäiväiselle ohjelmistokehitykselle. Tämän opinnäytetyöraportin pääpaino on kuitenkin kehitetyn testausjärjestelmän, luodun testisovelluksen sekä saavutettujen tulosten esittelyssä.

2 OHJELMISTOTESTAUS

2.1 Ohjelmistotestauksen tavoitteet [3]

Ohjelmiston testausta on kaikki toiminta, jolla pyritään varmistamaan, että kehitettävä sovellus toimii sille ennakkoon määritellyn spesifikaation mukaisesti. Yhtenä testauksen pääsääntönä voidaan ohjelmistokehityksessä pitää ajatusta, että ohjelman toiminnassa on aina virheitä, eli ristiriitoja toteutetun ja suunnitellun toiminnan välillä. Mitä enemmän ohjelmistosta löydetään virheitä tuotekehityksen aikana, sitä onnistuneempaa testausmenetelmien suunnittelu ja soveltaminen on ollut. Onnistunut testaus nopeuttaa loppukäyttäjälle päätyvän tuotteen julkaisua ja parantaa sen laatua. Hyvä testaus on järjestelmällistä virheiden etsintää ja vaatii suunnittelua tuotekehityksen alusta asti.

Laajoissa ohjelmistoprojekteissa täysin virheetöntä ohjelmistoa on käytännössä mahdoton tehdä, joten testausta suunniteltaessa tulee muistaa, että täydellisen testikattavuuden tavoitteleminen ei ole projektin etenemisen kannalta järkevää. Julkaisua edeltävällä testauksella tulee pyrkiä siihen, että valmiissa tuotteessa olisi mahdollisimman vähän käyttäjälle ilmeneviä virheitä. Ohjelmiston testauksen voidaan katsoa olevan valmis silloin, kun tuote toimii suunnitellusti. Mikäli tuotteen kehitys jatkuu vielä julkaisun jälkeen, tulee testausmenetelmienkin kehittyä. Testaaja voi tilanteesta riippuen olla osa tuotekehitystä, ulkopuolinen testaushenkilö tai jopa tuotetta käyttävä asiakas.

2.2 Manuaalinen testaus [3], [4]

Manuaalinen testaus ja raportointi ovat usein ensimmäisiä tapoja, joilla kehitettävän ohjelmiston toimintaa arvioidaan. Tämän tason testauksen suurin ongelma on se, että se vaatii aina työntekijän testaamaan ja dokumentoimaan tulokset. Käytännön työelämässä manuaaliseen testaukseen liittyy myös usein paljon itseään toistavia tehtäviä, jotka johtavat tehottomaan ajankäyttöön (esimerkiksi testattavan ohjelmistoversion asennus). Manuaalisesta testauksesta ei kuitenkaan voida täysin luopua, sillä tietokoneavusteisesti on mahdoton analysoida kaikkia ohjelmistoihin liittyviä ominaisuuksia (esimerkiksi ohjelmiston käytettävyys). Lisäksi manuaalisissa testeissä voidaan havaita virheitä, joita automatisoidussa testausympäristössä ei koskaan tulisi esille. Ihmisen toimintaan liittyy

aina epäjohdonmukaisuutta, joka voi ajaa testattavan ohjelmiston tilaan, jota ei testauksen suunnitteluvaiheessa osattu ajatella.

2.3 Yksikkö- ja integraatiotestaus [3], [4]

Yksikkötestaus on ohjelmiston pienimpien toimintakokonaisuuksien eli sovelluksen yksittäisten funktioiden tai luokkametodien testausta. Yksikkötestien suunnittelu ja kirjoitus on yleensä sen ohjelmistokehittäjän vastuulla, joka testattavan ominaisuuden on tehnyt. Yksikkötestit tehdään kehitettävän ohjelmistoprojektin yhteyteen omina testiohjelminaan, jotka kutsuvat testattavia funktioita ennakkoon määritellyillä parametreilla ja raportoivat testien tulokset perustuen paluuarvoihin.

Integraatiotestauksessa testataan ohjelmistokokonaisuuden moduulien välisiä rajapintoja. Tämän tason testien toimintaperiaate on samankaltainen kuin yksikkötestauksessa. Usein integraatiotestauksen käsittämät kokonaisuudet ovat kuitenkin niin laajoja, että testausta varten kehitetään erillinen testisovelluksensa. Integraatiotestauksen suunnitteluun voi osallistua useita ohjelmistokehittäjiä sekä testaushenkilöitä.

2.4 Järjestelmä- ja regressiotestaus [3], [4]

Järjestelmätestausta tehdään, kun kehitettävä ohjelmistoversio on lähes valmis. Järjestelmätesteihin sisällytetään usein ensimmäisenä niin kutsutut sauhutestit (smoke test). Sauhutesteillä pyritään varmistamaan ohjelmiston perustason toiminta ja tuotteen käyttökelpoisuus. Järjestelmätason testeihin voidaan sisällyttää kaikkien järjestelmäkokonaisuuden toimintaan ja dokumentaatioon liittyvien ominaisuuksien testitapaukset. Esimerkkeinä testitapauksista ovat yhteensopivuustestit ulkopuolisten järjestelmien kanssa sekä turvallisuus- ja kuormitustestit. Kokonaisvaltaiseen järjestelmätestaamiseen kuuluu myös sekä yksikkö- että integrointitestit ja järjestelmän kaikkien toimintojen kattava testaus manuaalisesti.

Ohjelmistoregressiolla tarkoitetaan uusien ominaisuuksien käyttöönoton aiheuttamia virheitä ohjelmiston vanhoissa toiminnallisuuksissa. Regressiotestauksella pyritään siis varmistamaan se, että myös aiemmin toimiviksi todetut toiminnot kehitettävässä ohjel-

mistossa eivät lakkaa toimimasta uusissa ohjelmistoversioissa. Ohjelmistotestauksen automatisointi helpottaa regressiotestausta järjestelmätestauksen yhteydessä, sillä vanhojen ominaisuuksien testitapaukset pysyvät muuttumattomina, vaikka ohjelmistoon esiteltäisiinkin uusia toimintoja.

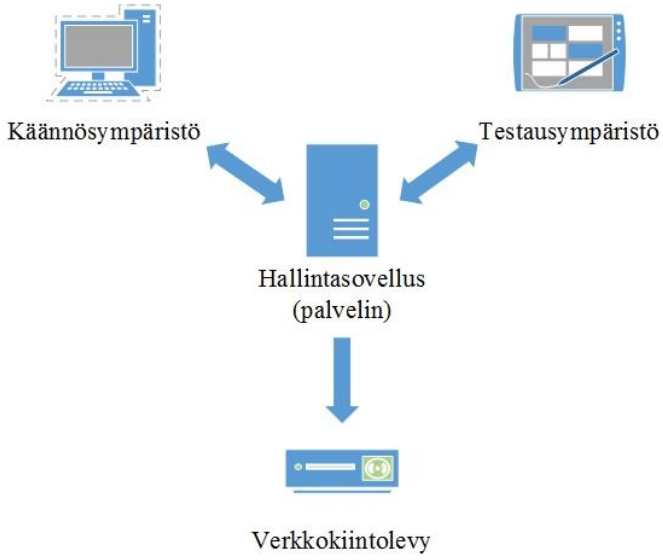
2.5 Ohjelmistotestauksen automatisointi

Ohjelmistotestaus pystytään automatisoimaan monella eri tavalla käyttäen erilaisia työkaluja riippuen testattavasta järjestelmästä ja testauksen laajuudesta. Lähes poikkeuksetta kuitenkin kaikki automaattiset ohjelmistotestausjärjestelmät koostuvat toimintaperiaatteeltaan samankaltaisista osakokonaisuuksista.

Automaattisessa ohjelmistotestauksessa testauksen sekä testitulosten analysoinnin ja raportoinnin tekee ihmisen sijasta jokin siihen tehtävään suunniteltu järjestelmä tai sovellus. Kaikki automatisoidut testausmenetelmät perustuvat siihen, että testattavalle järjestelmälle pystytään ajon aikaisesti antamaan syötteitä ja samalla lukemaan järjestelmän vasteita näille syötteille. Ohjelmallisessa testauksessa eri ominaisuuksien testit perustuvat väittämiin (assert), jotka ovat testin kirjoittajan olettamuksia esimerkiksi funktion tai metodin paluuarvosta. Testattaessa siis kutsutaan funktiota ja oletetaan tälle funktiolle jokin paluuarvo. Mikäli paluuarvo ei vastaa väittämää ja testi on kirjoitettu oikein, voidaan päätellä, että funktion suorituksen yhteydessä tapahtuu jotain virheellistä.

Automaattisten ohjelmistotestausjärjestelmien toiminta perustuu hallintasovellukseen, joka mahdollistaa erilaisten tehtävien ja testien automatisoinnin, seurannan sekä tulosten raportoinnin. Riippuen järjestelmän automaatioasteesta voidaan puhua joko täysin automaattisesta tai puoliautomaattisesta testausjärjestelmästä. Täysin automaattiset järjestelmät tarkkailevat esimerkiksi käytössä olevaa versionhallintajärjestelmää muutosten varalta ja muutoksen tapahtuessa suorittavat niille ennakkoon määritellyt tehtävät. Puoliautomaattisessa järjestelmässä voidaan tehtäväsekvenssin aloitus tai muu osa jättää ihmisen tehtäväksi. Lisäksi automaattisissa ohjelmistotestausjärjestelmissä on käytössä käännöspalvelin ja joko erillinen tai käännöspalvelimen yhteydessä toimiva testausympäristö. Novatron Oy:n testausjärjestelmässä on lisäksi käytössä kaikille näkyvä verkkokiintolevy, johon tallennetaan testattavan ohjelmiston asennustiedostot (kuva 1).

Käännöspalvelimella testattavan ohjelmiston uusin versio käännetään tietokoneella suoritettavaksi sovellukseksi, ja testausympäristössä suoritetaan sovellukselle suunnitellut testitapaukset. Suositeltavaa on, että testaus tapahtuu siinä ympäristössä, missä ohjelmistoa tullaan todellisuudessa käyttämään.



KUVA 1. Ohjelmistotestausjärjestelmän komponentit

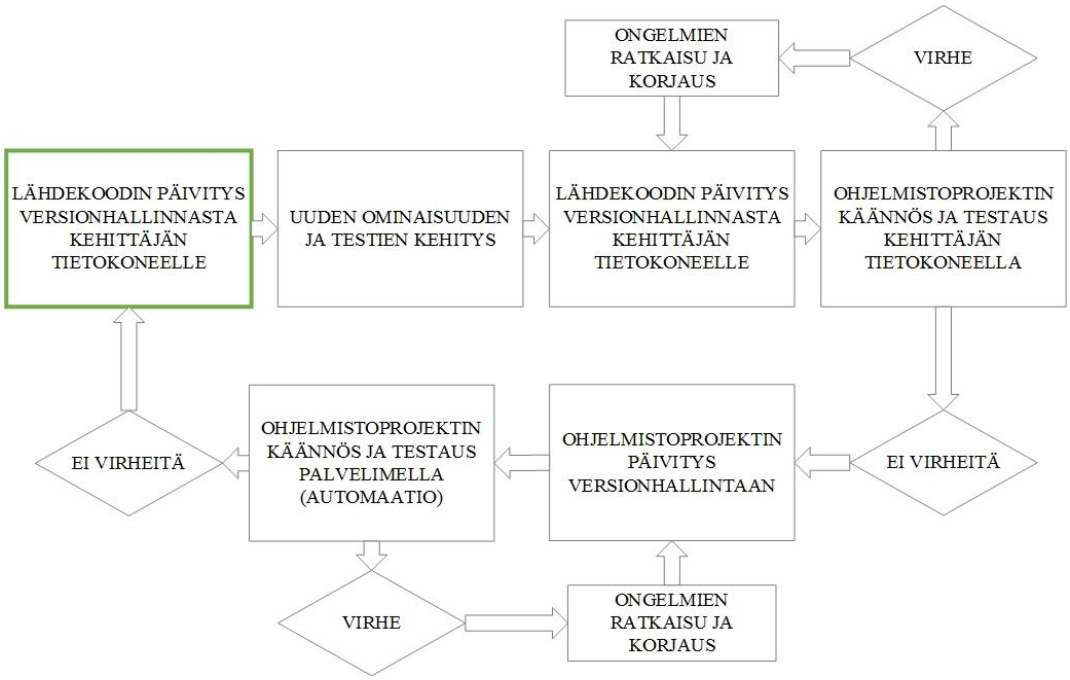
Hallintasovellus, käännöspalvelin sekä testausympäristö vaativat toimiakseen vielä lisäksi työkaluja, joilla osakokonaisuudet saadaan toimimaan yhdessä halutulla tavalla. Näiden työkalujen valinta tehdään tapauskohtaisesti. Käännöspalvelimella tulee olla testien hallintasovelluksen kanssa yhteensopiva käyttöjärjestelmä sekä ohjelmistoprojektien kääntämiseen vaadittavat työkalut. Testausympäristöä varten tulee joko kehittää oma tai valita valmis testisovellus, jolla testattavaa järjestelmää testataan. Lisäksi kaikkien järjestelmän komponenttien tulee kyetä kommunikoimaan hallintasovelluksen kanssa, jotta automatisoitujen tehtävien hallinta ja tulosten raportointi olisi mahdollista.

2.6 Jatkuva integraatio [1]

Kun automaattista ohjelmistotestausta suunniteltiin, oli tavoitteena myös samalla muokata Novatron Oy:n ohjelmistokehityksen työskentelytapoja jatkuvan integraation (continuous integration) ja testivetoisen ohjelmistokehityksen (test driven development) mukaisiksi. Kehitettävien ohjelmistojen laatua ja uusien ohjelmistoversioiden julkaisutahtia haluttiin parantaa ohjelmistokokonaisuuksien ja tuoteperheen laajentuessa. Tuo-

tekehityksen työskentelymenetelmiä pyritään muuttamaan tehokkaammiksi jo ennen kuin uusien tuotteiden ohjelmistoja aletaan suunnitella.

Jatkuva integraatio on yksi ketteristä ohjelmistokehityksen toimintamalleista, jossa jokainen ohjelmistokehittäjä integroi ohjelmakoodiin tekemänsä muutokset versionhallintajärjestelmään päivittäin tai useammin. Jatkuvan integraation tavoitteena on toimintavarmojen ohjelmistosovellusten kehittäminen nopeasti. Tämän toimintamallin pääperiaatteita ovat välitön reagointi ohjelmistokehityksen ongelmatilanteisiin (ohjelmiston käännöksessä ja testeissä ilmenevät virheet) ja käännös- sekä testausautomaation kattava soveltaminen kehitystyössä. Jatkuvan integraation toimintamallin mukaan versionhallinnassa sijaitsevan ohjelmistoprojektin kehityshaaran toiminta pitäisi aina olla taattuna. Toimimatonta versiota ohjelmasta ei saa jättää korjaamatta. Ongelmien ja vikatilanteiden selvittäminen ja korjaaminen ovat kehitystyössä tärkeysjärjestyksessä ensimmäisiä. Uusia ominaisuuksia ei pidä sisällyttää kehitettävän ohjelmiston lähdekoodiin, ennen kuin vanhat ominaisuudet toimivat. Kuvassa 2 on jatkuvan integraation työskentelymalli lohkokaaviona. Mallin mukaan versionhallintaan ei päivitetä uutta sisältöä, ennen kuin ohjelmistokehittäjä on saanut muutokset toimimaan omalla työpisteellään, ja on varmistunut siitä, että tehdyt muutokset toimivat yhdessä muiden ohjelmistokehittäjien lisäämien toimintojen kanssa.



KUVA 2. Jatkuvan integraation mukainen työskentelymalli

2.7 Testivetoinen ohjelmistokehitys [1]

Testivetoisessa ohjelmistokehityksessä testitapausten suunnittelua ja kirjoitusta käytetään kehitystyön ohjenuorana. Menetelmä perustuu kolmeen yksinkertaiseen ohjelmistokehityksen vaiheeseen:

- suunnittele ja kirjoita testitapaus (yleensä yksikkötesti) seuraavaksi kehitettävälle toiminnolle
- suunnittele ja kirjoita toiminnolle toteutus niin, että se läpäisee kirjoitetun testin
- muokkaa ohjelmakoodin rakennetta, jotta kokonaisuus pysyy selkeänä ja modulaarisena (lähdekoodin refaktorointi).

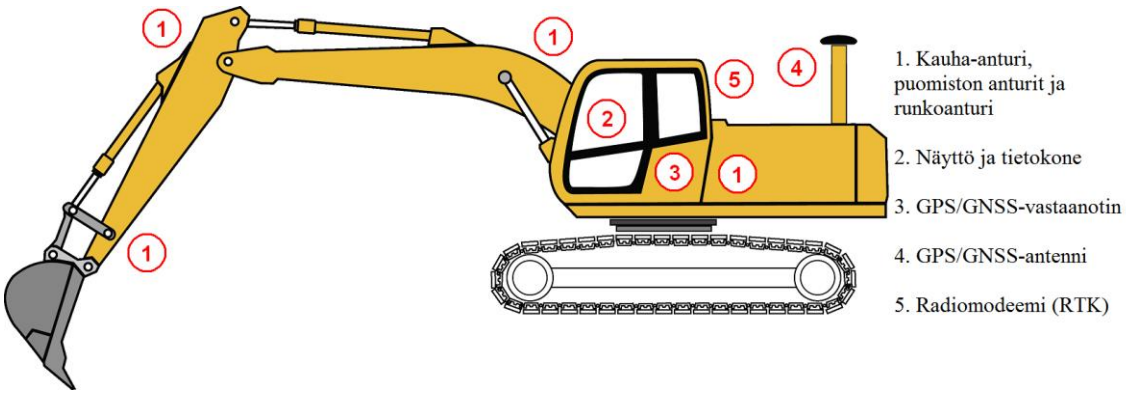
Tavoitteena on kirjoittaa ohjelmakoodia, jolle on olemassa testitapaukset ainakin yksikkötestitasolla kehityksen alusta asti, eikä toimimatonta funktiota tai luokkametodia tule lisätyksi ohjelmistoprojektiin. Uuden toiminnallisuuden kehitystyö loppuu vasta silloin, kun kirjoitettu testi on läpäisty. Lisäksi tällä tavalla toimiessaan kehittäjä joutuu ensin suunnittelemaan uuden toiminnon rajapinnan (miten kehitettävää funktiota tai luokkametodia kutsutaan testissä) ja vasta sen jälkeen kirjoittaa toiminnon ohjelmakoodin, jonka myötä rajapinnoista tulee yksinkertaisia käyttää.

3 LÄHTÖTILANNE

3.1 Testattava järjestelmä

Novatron Oy suunnittelee, valmistaa ja toimittaa mittausjärjestelmiä maanrakennustyökoneisiin. Suurin osa myytävistä järjestelmistä asennetaan kaivinkoneisiin, mutta saatavilla on myös mittausjärjestelmiä muihin liikkuviin työkoneisiin. [5]

Kaivinkoneiden kaivuusyvyysmittarit voidaan jakaa karkeasti 2D- ja 3D-järjestelmiin (kuva 3). 2D-järjestelmien mittauslaskenta perustuu ainoastaan työkoneeseen asennettaviin kiihtyvyysantureihin, joiden antaman datan perusteella voidaan laskea työkoneen kauhan etäisyys ja korkeus käyttäjän määrittelemän pisteen suhteen. 3D-järjestelmissä on kiihtyvyysanturien lisäksi käytössä paikannustekniikka (satelliittipaikannus tai robottitakymetri), joka mahdollistaa koneen reaaliaikaisen paikkatiedon hyödyntämisen laskennassa. 3D-järjestelmä mittaa nimensä mukaisesti kauhan paikkaa ja asentoa kolmessa ulottuvuudessa. Kun 2D-järjestelmällä saadaan suhteellinen tieto kauhan etäisyydestä ja korkeudesta käyttäjän määrittelemään pisteeseen, voidaan 3D-järjestelmällä mitata kauhan absoluuttinen sijainti kolmiulotteisessa koordinaatistossa. 3D-järjestelmää pystyy käyttämään myös 2D-tilassa, mikäli paikannustietoa ei ole saatavilla.



KUVA 3. Novatron Oy:n mittausjärjestelmän komponentit kaivinkoneessa

KUVA 4. Novatron Oy:n Xsite Link -mittausjärjestelmä kaivinkoneessa [5]

3.2 Testaus ennen uusien testausmenetelmien kehitystä

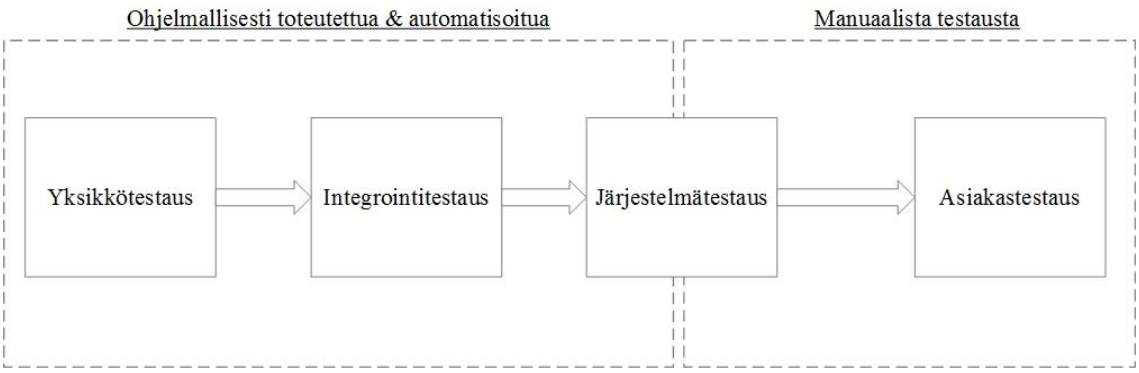
Ennen uusien testausmenetelmien suunnittelua Novatron Oy:n ohjelmistotestaus oli pääsääntöisesti manuaalista. Ohjelmistojen toiminnassa havaitut virheet kirjattiin tuotekehityksen ja asiakastuen ylläpitämään palautejärjestelmään ja ongelmat pyrittiin korjaamaan mahdollisimman nopeasti. Testaus oli tuotekehitys- ja tukihenkilöstön vastuulla ja testaustehtäviä jaettiin käytössä olevien resurssien mukaan. Tämä toimintatapa tulee säilymään myös automaattisen testausjärjestelmän käyttöönoton jälkeen, mutta entistä tehokkaampana. Uuden ohjelmistoversion siirtyessä manuaaliseen järjestelmätestausvaiheeseen, siinä saattoi olla paljon ohjelmointivirheitä, jotka tekivät testauksen loppuun viemisestä mahdotonta. Virheet olivat usein sen laatuisia, että koko testaustyö oli aloitettava korjausten jälkeen uudelleen. Lisäksi testaukseen liittyvä dokumentointi oli melko kirjavaa, eikä yhtä selkeätä raportointimallia ollut. Yksikkö- ja integrointitestejä oli tehty jossain määrin riippuen kehittäjästä ja projektista, mutta yhtenäisiä ohjelmistotestauksen työkaluja ja menetelmiä ei ollut.

3.3 Testausmenetelmien kehitysprojektin tavoitteet

Testausmenetelmien kehitysprojektin tarkoituksena oli suunnitella ja saada toimimaan Novatron Oy:n tuotekehitysosastolle helposti muokattava automaattisen ohjelmistotestauksen toimintaympäristö. Samalla testausmenetelmiä haluttiin kehittää kokonaisuute-

na tehokkaampaan ja selkeämpään suuntaan (kuva 5). Testausautomaation jatkokehittämisestä ja ylläpitämisestä tulee osa Novatron Oy:n jatkuvaa tuotekehitystä.

Projektille määriteltiin selkeät tavoitteet. Koska aikaisempaa kokemusta ohjelmistotestauksen automatisoinnista ei ollut, oli ensimmäinen päätavoite ohjelmistotestauksen automatisoinnin eri menetelmiin tutustuminen. Pidemmän aikavälin tavoitteena oli kehittää järjestelmä, joka mahdollistaa ohjelmistoprojektien kääntämisen ja ohjelmistotestauksen automatisoinnin. Lisäksi Novatron Oy:n ohjelmistotestausmenetelmiä haluttiin kehittää myös manuaalisen testauksen osalta.



KUVA 5. Novatron Oy:lle suunniteltu ohjelmistotestauksen kulku vaiheittain

Testausautomaatiojärjestelmää alettiin suunnitella ensimmäiseksi 3D-järjestelmälle. Testausjärjestelmän kehityksen alkaessa määriteltiin testattavalle järjestelmälle kriittisimmät ominaisuudet, joiden toiminta tulisi testauksen automatisoinnin myötä olla aina varmistettuna. 3D-järjestelmän testauksessa haluttiin ensisijaisesti testata mittausjärjestelmän käyttöliittymän toiminta, tietoliikenneyhteydet järjestelmän komponenttien välillä (CAN-, GPS/GNSS-, RTK- ja internet-liikenne) sekä mittausjärjestelmän laskentatoiminnot ja mittaustarkkuus.

Testaamalla järjestelmän avainominaisuudet automaattisesti voidaan myös manuaalisesti testattaessa olla varmoja siitä, että mikäli näissä toiminnallisuuksissa havaitaan virhe, niin virheen alkuperä on luultavasti käyttäjä- tai asennuslähtöinen. Näin toimimalla siis helpotetaan virheen etsintää rajaamalla mahdollisten aiheuttajien kokonaismäärää pienemmäksi.

Automaattisen ohjelmistotestausjärjestelmän ei ole missään vaiheessa tarkoitus korvata manuaalisesti tehtävää järjestelmätestausta vaan tehostaa ohjelmistokehitystä ja testaus-

ta kokonaisuudessaan. Ohjelmistotuotannon eri vaiheiden automatisointi helpottaa ohjelmistokehitystä suunnittelusta testaukseen, kun rutiininomaisia ja aikaa vieviä tehtäviä ei tarvitse tehdä manuaalisesti. Järjestelmätestaukseen siirtyvä ohjelmistoversio on toimintavakaa, eikä siinä esiinny virheitä, joiden vuoksi testaus täytyy aloittaa alusta uudelleen useita kertoja. Lisäksi jatkuvan integraation mukainen työskentelymalli, jonka toiminnan edellytyksenä ovat automatisoidut käännös- ja testaustoiminnot, mahdollistaa ohjelmistovirheiden löytymisen heti, kun ne esitellään versionhallintaan. Tämä nopeuttaa virheiden korjaamista huomattavasti, sillä ohjelmistovirheen paikallistamiseen ja korjaamiseen kulutetut resurssit kasvavat sitä suuremmaksi, mitä pidempi aika virheen tekemisen ja havaitsemisen välillä on.

4 KEHITETTY TESTAUSJÄRJESTELMÄ

4.1 Jenkins CI [8]

Novatron Oy:n ohjelmistotestausjärjestelmän hallintasovellukseksi valittiin Jenkins CI -niminen (myöhemmin Jenkins) Java-kielinen avoimen lähdekoodin palvelinohjelmisto (kuva 6). Ohjelmistolla voidaan automatisoida, ketjuttaa sekä monitoroida ohjelmistoprojektin kääntämiseen ja testaamiseen liittyviä tehtäviä. Jenkinsin avulla on mahdollista automatisoida mitä tahansa tietokoneella suoritettavia tehtäviä, joiden kontrollointi onnistuu komentoriviliittymän kautta. Valitun hallintasovelluksen käyttömahdollisuuksia ei siis ole rajattu pelkästään ohjelmistoprojektien kääntämiseen ja testaamiseen. Jenkinsissä on oletuksena tuki Subversion- ja CVS-versionhallintajärjestelmien seurantaan. Muiden yleisimpien versionhallintajärjestelmien (esimerkiksi Git) seurantaan on olemassa valmiit erikseen asennettavat ohjelmistoliitännäiset.

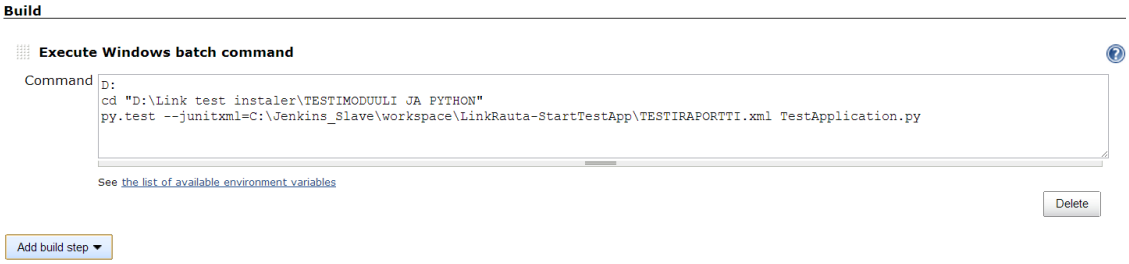
Jenkins on yksi suosituimmista jatkuvan integraation palvelinohjelmistoista. Sitä käyttävät niin avoimen lähdekoodin ohjelmistoprojektit kuin suuret kansainväliset yritykset-kin. Sen etuihin lukeutuvat aktiivinen kehittäjäyhteisö, monipuoliset ja jatkuvasti päivittyvät ohjelmistoliitännäiset sekä yhteensopivuus eri käyttöjärjestelmäalustojen kanssa.

KUVA 6. Jenkinsin internet-selaimella luettavan käyttöliittymän avausnäkymä

Jenkinsille voidaan määritellä erilaisia tehtäviä, joita voidaan jakaa usean eri tietokoneen suoritettavaksi. Testauksen kannalta tehtävien jakaminen on erittäin hyödyllinen

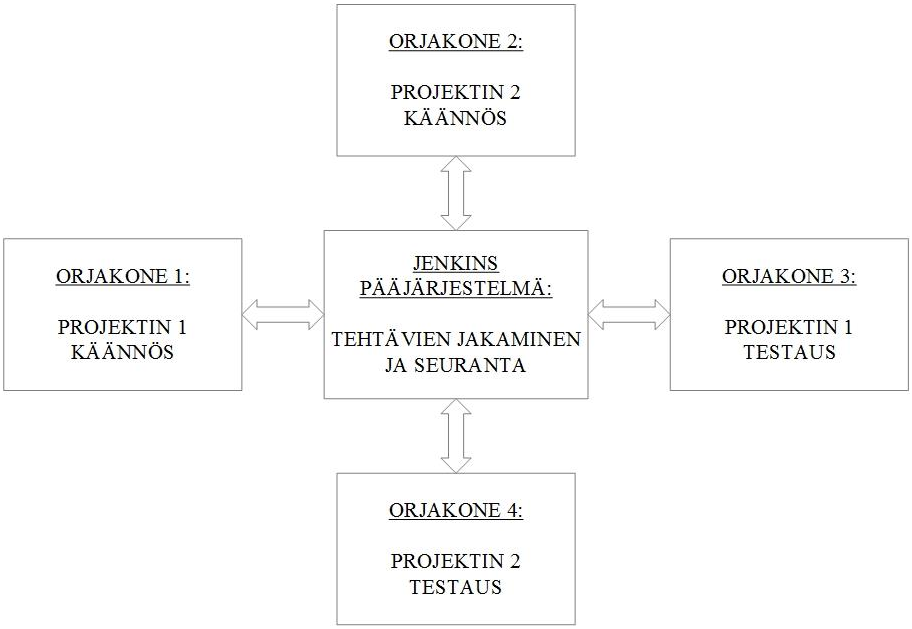
ominaisuus, sillä siten voidaan määritellä, missä ympäristöissä ohjelmistoprojekti käännetään ja missä ohjelmistoa testataan. Jakamalla tehtäviä useiden eri tietokoneiden kesken voidaan myös vähentää yksittäisten tietokoneiden jatkuvaa kuormitusta. Kuormituksen jako eri yksiköille tehostaa ja nopeuttaa tehtävien suoritusta. Tehtävämäärittely koostuu yleisellä tasolla seuraavista vaiheista:

- tehtävän nimeäminen ja kuvaus
- tehtävän suoritusympäristön määrittely
- tehtävän suoritushetken määrittely (esimerkiksi märitellyin väliajoin, versionhallintajärjestelmässä tapahtuneen muutoksen myötä tai jonkun toisen tehtävän suorituksen jälkeen)
- suoritettavien tehtävien määrittely (esimerkiksi komentorivikäskyn kirjoitus, kuva 7)
- tehtävän suorituksen jälkeisten toimintojen määrittely (esimerkiksi sähköpostiilmoitus tuotekehityshenkilöstölle).



KUVA 7. Osa Jenkinsin tehtävämäärittelyä, jossa järjestelmälle syötetään suoritettava komento (Windows-komentorivikäsky)

Jos Jenkins-hallintasovellusta ajetaan eri tietokoneella kuin käännös- ja testausympäristöjä, pitää tätä varten valita kommunikointimenetelmä Jenkinsin ja muiden automaatiojärjestelmän komponenttien välille. Jenkinsin mahdollistama työtehtävien jako eri tietokoneiden kesken tehdään siten, että Jenkinsille määritellään erillisiä orjakoneita (slave-executor), joita Jenkinsin pääjärjestelmä (master-executor) kontrolloi (kuva 8). Pääjärjestelmä vastaa tehtävien käynnistyksestä ja monitoroinnista, mutta itse tehtävien suoritus tapahtuu orjakoneissa. Orjakoneiden ja pääjärjestelmän välinen kommunikointi perustuu erilliseen slave agent -sovellukseen. Slave agent on Java-sovellus, joka vastaanottaa ja välittää tietoa orjakoneen ja pääjärjestelmän välillä. Orjakoneissa ei tehdä muuta kuin niille määrättyjen tehtävien suoritus. Pääjärjestelmä ja orjakoneiden laitteistot voivat poiketa toisistaan, eikä tehtävien jakaminen vaadi eri yksiköille esimerkiksi samaa käyttöjärjestelmää.



KUVA 8. Esimerkki Jenkinsin tehtävänjaon hyödyntämisestä

4.2 Käännöstyökalut

Ohjelmistoprojektin automaattinen kääntäminen vaatii käännöstyökalun, jota voi käyttää komentoriviliittymän kautta. Koska Novatron Oy:n käytössä ollut ohjelmistokehitysympäristö (Code Blocks IDE) ei mahdollistanut projektien kääntämistä komentoriviliittymän kautta, piti tähän tarkoitukseen etsiä vaihtoehtoinen työkalu. Kehitysympäristön vaihtamista kokonaan uuteen ei koettu järkeväksi. Kääntämiseen valittiin GNU Make -niminen (myöhemmin Make) ohjelmisto. Maken toiminta perustuu niin kutsuttuihin make-tiedostoihin, jotka sisältävät ohjeita erilaisten tehtävien suorittamiseksi. Make ei itse tee lähdekooditiedostojen kääntämistä konekieliseksi, vaan tekee sille make-tiedostoissa annettujen ohjeiden mukaisia tehtäviä niillä työkaluilla, mitä sen toimintaympäristössä on mahdollista käyttää. Ohjelmistoprojektin kääntämiseen konekieliseksi tarvitaan edelleen kääntäjä (esimerkiksi gcc-kääntäjä).

Make-tiedostojen luomiseen valittiin avoimen lähdekoodin ohjelmistoprojekti nimeltä CBP2Make, joka generoi Code Blocks -projektitiedostoista suoraan make-tiedostot. Suuremmissa projekteissa make-tiedostot ovat laajoja kokonaisuuksia, jotka sisältävät monia tiedostojen välisien riippuvuuksien määrittelyitä ja suoritettavia komentoja. Yksinkertainen foo-niminen objektitiedosto voitaisiin luoda Make-ohjelmistolla alla olevan

esimerkin mukaisesti gcc-kääntäjän avulla. Foo.o-tiedosto riippuu foo.c- ja foo.h-tiedostoista.

```
foo.o: foo.c foo.h
    gcc –c foo.c
```

4.3 Python [6]

Testausjärjestelmän testisovellus kirjoitettiin Pythonilla, joka on tulkattava ja oliopohjainen ohjelmointikieli. Testisovellusta suunniteltaessa vaatimuksina halutulle ohjelmointikielelle oli yksinkertaisuus, monipuolisuus ja yhteensopivuus C / C++-kielien kanssa (testattava järjestelmä on pääosin C++-kielinen). Python sopi tähän kuvaukseen täydellisesti. Lisäksi Pythonin standardikirjasto pitää sisällään testaukseen liittyviä ohjelmointikirjastoja. Pythonille on tarjolla useita kielen laajennuspaketteja, joiden avulla testausmenetelmien määrää voidaan kasvattaa entisestään. Python-kielisiä sovelluksia suoritetaan Python-tulkin avulla, joten Python-sovelluksia käyttävässä ympäristössä tulee olla tulkki asennettuna. Python-sovelluksia voidaan ajaa komentoriviliittymän kautta kutsumalla Python-tulkkia ja määrittelemällä suoritettavan Python-sovelluksen lähdekooditiedosto. Windows-ympäristössä Python-sovellus voitaisiin suorittaa alla olevan esimerkin mukaisesti.

```
C:\PythonTulkki\Python TestApplication.py
```

4.3.1 Python XML-RPC

RPC (Remote Procedure Call) on tietoliikenneyhteysprotokolla, jonka avulla asiakassovellus (RPC-client) voi kutsua funktioita palvelinohjelmasta (RPC-server), vaikka palvelinohjelmisto sijaitsisi fyysisesti eri tietokoneella. XML-RPC on RPC-toteutus, jossa tiedonsiirto tapahtuu http-yhteyden kautta ja lähetettävä data on XML-formaatissa. Pythonin standardikirjasto sisältää valmiit ohjelmointirajapinnat XML-RPC-yhteyden muodostamiselle kahden sovelluksen välille, josta kuvassa 9 on esimerkki. Kuvan ylemmässä laatikossa luodaan palvelin ja rekisteröidään asiakasohjelmasta kutsuttavat

funktiot. Alemmassa laatikossa muodostetaan yhteys palvelimeen ja kutsutaan rekisteröityä esimerkkifunktiota.

```
#XML-RPC-server-moduulin import
from xmlrpc.server import SimpleXMLRPCServer

#yksinkertainen esimerkkifunktio, joka tullaan rekisteröimään asiakasohjelmiston käytettäväksi
def registerThisFunction(param1, param2):
        return param1+param2

#serverin alustus lokaalisti (osoite ”http://localhost:8000")
server = SimpleXMLRPCServer(”localhost”, 8000)
#haluttujen funktioiden rekisteröinti
server.register_function(registerThisFunction)
#server-ohjelmiston käynnistys
server.serve_forever();
```

```
#XML-RPC-client-moduulin import
import xmlrpc.client

#XML-RPC-palvelimeen yhdistäminen
proxy = xmlrpc.client.ServerProxy(”http://localhost:8000")
#palvelimella rekisteröidyn funktion kutsuminen
proxy.registerThisFunction(2, 2)
```

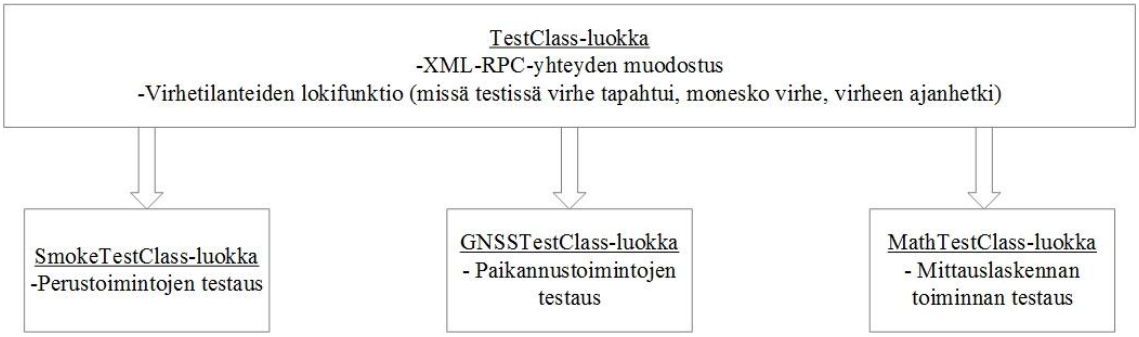
KUVA 9. Esimerkki yksinkertaisesta Python-kielisestä XML-RPC-sovelluksesta

Testausjärjestelmässä XML-RPC-protokollaa käytetään testisovelluksen ja testattavan sovelluksen väliseen kommunikointiin niin, että XML-RPC-palvelinohjelmistoon on rekisteröity testattavan sovelluksen funktioita ja asiakasohjelmistosta (testisovellus) kutsutaan näitä funktioita. Paluuarvojen perusteella analysoidaan testattavan ohjelmiston toimintaa. Käyttämällä XML-RPC-protokollaa testisovellus ja testattava sovellus voidaan erottaa kahdeksi erilliseksi kokonaisuudekseen. Testitapauksia voidaan ajaa missä vain testattavan sovelluksen suoritusvaiheessa, kunhan XML-RPC-sovelluksen palvelinyhteys on auki. Testaus voi tapahtua paikallisesti yhden laitteen sisällä tai vaihtoehtoisesti niin, että testisovellusta ajetaan jossain muussa ympäristössä kuin testattavassa laitteessa.

4.3.2 Testisovellus

Python-kielisestä testisovelluksesta luotiin ensimmäinen versio, jolla testausjärjestelmää päästiin kokeilemaan käytännössä. Testisovellus vaatii vielä jatkokehitystä eikä laajoja testejä ole vielä kirjoitettu. Sovelluksen rakenne on kuitenkin suunniteltuna ja toiminta testattu. Testisovelluksen esimerkkirakenne on kuvassa 10. Ohjelmisto koostuu luokista,

joista jokaisessa on omat testitapauksensa. Jokainen testiluokista perii TestClass-kantaluokan, jossa on XML-RPC-yhteyden muodostamiseen sekä virhetapausten raportointiin tarvittavat luokkametodit. TestClass- ja SmokeTestClass-luokkien esimerkkitoteutukset ovat kuvassa 12.



KUVA 10. Esimerkki testisovelluksen rakenteesta

4.3.3 PyTest [7, 2]

PyTest on Pythonin standardiasennuspaketin ulkopuolinen testaustyökalu, joka mahdollistaa ohjelmallisen ohjelmistotestauksen yksikkötestauksesta laajempiin testauskokonaisuuksiin. Kirjaston avulla kirjoitettuja testitapauksia voidaan parametrisoida, eli määritellä käyttäjän syötteiden mukaan suoritettaviin testimoduuleihin. Testituloksista voidaan generoida JUnitXML-formaatin mukaiset testiraportit, joiden perusteella Jenkins osaa muodostaa graafisia testitulosraportteja. Python-ohjelman lähdekoodiin, joka sisältää PyTest-testejä, sisällytetään avainsanoja, joiden perusteella PyTest suorittaa testeihin liittyvät operaatiot sekä analysoi ja raportoi tulokset. PyTestille voi antaa sitä komentoriviliittymän kautta kutsuttaessa erilaisia parametreja, joilla voidaan vaikuttaa siihen, mitä testitapauksia luettavasta Python-sovelluksesta suoritetaan ja millä tavoin ne raportoidaan. Esimerkiksi jos halutaan kutsua Python-sovellusta ”TestApplication.py” PyTestin avulla ja sen sisältämien testien tuloksista halutaan generoida JUnitXML-raportti, tulee komentoriville kirjoitettavan käskyn olla Windows-ympäristössä alla olevan esimerkin mukainen.

C:\PyTest\py.test –junitxml=C:\JUnitXMLreports\ TestAppliaction.py

Esimerkissä kutsutaan PyTest-työkalua, asetetaan JUnitXML-raportointi päälle ja määritellään tiedostopolku, mihin raportti tulee tallentaa. Kuvassa 11 on esitelty ensimmäi-

siä Python-kielisiä testitapauksia, jotka suoritettiin PyTestin avulla. Testeissä käytetään TestClass-kantaluokasta periytyvää SmokeTestClass-luokkaa, jonka metodeja kutsumalla voidaan kirjautua järjestelmään sisälle, nollata mitta-arvot ja kirjautua järjestelmästä ulos. TestClass-kantaluokassa on metodit XML-RPC-yhteyden muodostamiselle ja testaamiselle sekä virhetilanteissa kutsuttava lokifunktio. SmokeTestClass-luokan metodeilla testisovelluksen toimintaa voidaan testata oikeassa testausympäristössä (kuva 12). SmokeTestClass-luokan metodit tullaan muuttamaan erilaisiksi testisovelluksen jatkokehityksen aikana. Kuvan esimerkissä näkyvät metodit tehtiin niin, että testisovelluksen toimintaa päästiin testaamaan nopeasti. Esimerkiksi käyttöliittymän testauksessa funktioiden toiminnan sitominen kiinteisiin näyttökoordinaatteihin toimii vain niin kauan, kun käyttöliittymässä olevat elementit eivät muutu. Siksi käyttöliittymän toimintoja kutsuvat metodit tullaan myöhemmin toteuttamaan niin, että ne toimivat halutulla tavalla, vaikka käyttöliittymään tehtäisiin muutoksia.

```
import TestClasses

testiLuokka = TestClasses.SmokeTestClass()
#XML-RPC-yhteyden testaus
def test_XmlRpcConnection():
   assert testiLuokka.VerifyXmlRpcConnection() == True
#testaa järjestelmän sisäänkirjautuminen
def test_login():
   assert testiLuokka.testLogIn() == 0
#testaa mitta-arvojen nollaus
def test_resetvalues():
   assert testiLuokka.testResetValues() == True
#testaa uloskirjautuminen
def test_logout():
   assert testiLuokka.testLogOut() == 0
```

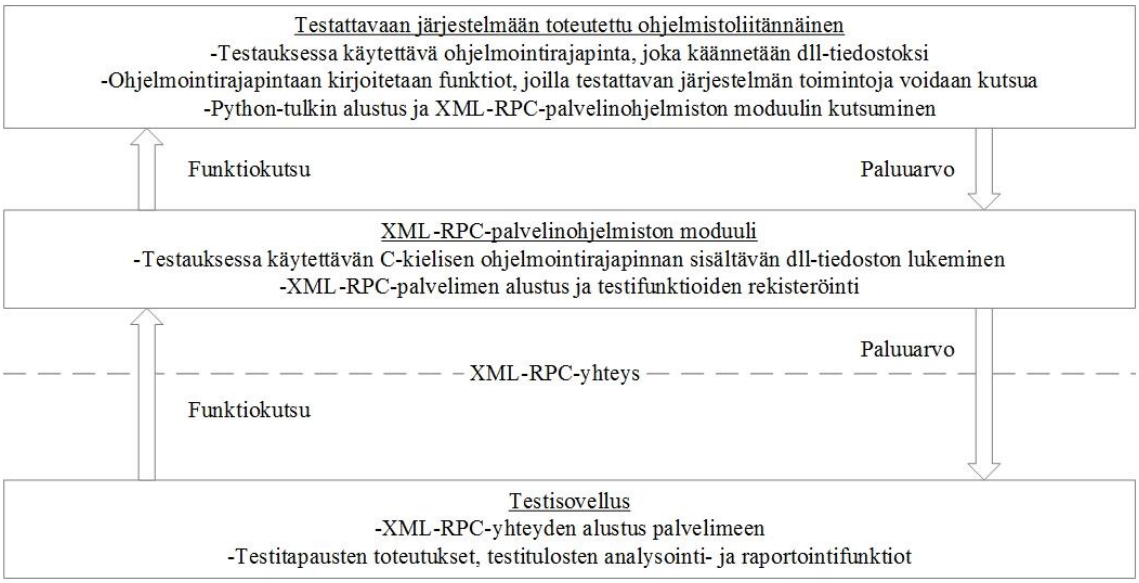
KUVA 11. Testisovelluksen PyTest-esimerkki

```
import xmlrpc.client
import time
from datetime import datetime
#testisovelluksen kantaluokka, tämän luokan perivät luokat, joilla suoritetaan joitain tiettyjä testejä
#esim. SmokeTestClass
class TestClass:
    #XML-RPC-yhteyden avaus
    Proxy = xmlrpc.client.ServerProxy("http://localhost:5050")
    #testeissä havaittujen virheiden kokonaislukumäärä
    ErrorCount = 0

    #kasvattaa virhelukumäärää kutsuttaessa
    def IncreaseErrorCount(self):
        self.ErrorCount = self.ErrorCount + 1

    #XML-RPC-yhteyden testaus, jos funktio palauttaa Falsen, niin yhteys ei toimi
    def VerifyXmlRpcConnection(self):
        test_connection = False

        for x in range(0,20):
            if test_connection == True:
                break
            else:
                time.sleep(2)
                try:
                    test_connection = self.Proxy.testConnection()
                except:
                    print("no xml-rpc connection")

        if test_connection == False:
            self.LogTestError('VerifyXmlRpcConnection', 'TestConnection', str(0))
        return test_connection

    #virheloki-funktio
    def LogTestError(self, TestName, FunctionName, ErrorCount):
        ErrorString = str(datetime.now())+' TestError at '+TestName+' in function: '+FunctionName+' ErrorCount: '+ErrorCount
        print(ErrorString)

#testisovelluksessa käytettävä luokka, joka perii TestClass-luokan
#SmokeTestClass-luokkaan on kirjoitettu funktioita, joita kutsumalla voidaan kokeilla testisovelluksen toimintaa
class SmokeTestClass(TestClass):
    def testLogIn(self):
        TestClass.Proxy.moveMouse(430, 590);
        time.sleep(2);
        TestClass.Proxy.clickMouseLeft();
        time.sleep(2);
        return 0;

    def testLogOut(self):
        TestClass.Proxy.leftAction(True)
        time.sleep(1);
        TestClass.Proxy.moveMouse(50, 100);
        time.sleep(1);
        TestClass.Proxy.clickMouseLeft();
        time.sleep(1);
        TestClass.Proxy.moveMouse(330, 350);
        time.sleep(2);
        TestClass.Proxy.clickMouseLeft();
        time.sleep(5);
        TestClass.Proxy.clickMouseLeft();
        return 0;

    def testResetValues(self):
        TestVariable = True;
        #testataan virheloki-funktion toimintaa
        if TestClass.Proxy.rightAction(False) == 0:
            TestVariable = False;
            TestClass.IncreaseErrorCount(self);
            TestClass.LogTestError(self,'testResetValues','rightAction(False)',str(TestClass.ErrorCount));
        time.sleep(5);
        return TestVariable;
```

KUVA 12. Testisovelluksessa käytetyt TestClass- ja SmokeTestClass-luokat

4.4 Testausrajapinta

Jotta testattavan järjestelmän toimintoja pystyttäisiin ohjelmallisesti kutsumaan, piti testattavaan sovellukseen kirjoittaa erillinen ohjelmistoliitännäinen. Liitännäinen mahdollistaa testausrajapinnassa olevien funktioiden kutsumisen Python-kielisestä testisovelluksesta. Testausrajapinta koostuu testattavan järjestelmän lähdekoodissa olevasta C-kielisestä ohjelmointirajapinnasta ja XML-RPC-palvelinohjelmistosta, joka rekisteröi C-kielisen rajapinnan funktiot testisovelluksen käytettäväksi (kuva 13). C-kielinen ohjelmointirajapinta on osa testattavan järjestelmän ohjelmistoprojektia ja se käännetään dll-tiedostoksi. XML-RPC-palvelinohjelmisto on Python-kielinen testattavan järjestelmän ohjelmistoprojektin ulkopuolinen moduuli, jota kutsutaan, kun testattava järjestelmä käynnistyy. Kun XML-RPC-palvelin on käynnistynyt, kutsuu se puolestaan C-kielistä ohjelmointirajapintaa ja rekisteröi sen funktiot testisovelluksen käytettäväksi. Tällä menettelyllä testisovelluksesta pystytään kutsumaan XML-RPC-yhteyden kautta C-kielisessä rajapinnassa olevia funktioita.



KUVA 13. Testisovelluksessa käytetyn XML-RPC-rajapinnan toiminta

Pythonin standardiasennuspaketin mukana tulee kirjastotiedostot, joiden avulla C++-kielisistä ohjelmistoista on mahdollista kutsua Python-kielisiä funktiota ja ohjelmistomoduuleita. Python-sovelluksessa voidaan lukea dll-tiedostoja ctypes-nimisen kirjaston avulla. Kuvassa 14 on yksinkertainen yhden funktion ohjelmointirajapinta, joka tullaan kääntämään dll-tiedostoksi. Ohjelmointirajapinnan funktio kutsuu WinAPI-luokan SetCursorPos-nimistä metodia. Ohjelmistotestauksessa tämänkaltainen funktio voisi olla

osa käyttöliittymän testauksessa käytettävää ohjelmointirajapintaa. Käännetty dll-tiedosto on nimeltään DLL_Export.dll. Kuvassa 15 on esimerkki Python-kielisestä sovelluksesta, joka lukee käännetyn dll-tiedoston ja rekisteröi sen sisältämän funktion Python-sovelluksessa käynnistettävälle XML-RPC-palvelinohjelmistolle. Kuvassa 16 on C++-kielinen ohjelmisto, jossa alustetaan Python-tulkki ja kutsutaan kuvan 15 Python-sovellusta. Novatron Oy:n testausjärjestelmän käyttämän testausrajapinnan toiminta perustuu kuvissa esitettyjen esimerkkien kaltaiseen ohjelmakoodiin. Kuvien esimerkkikoodi eroaa Novatron Oy:n testausrajapinnasta siten, että kun testattavasta sovelluksesta on kutsuttu Python-tulkkia ja XML-RPC-palvelin käynnistetään, niin testattava sovellus jatkaa normaalia toimintaansa. Kuvan 16 C++-kielisen ohjelmiston suoritus jää paikalleen XML-RPC-palvelimen käynnistymisen jälkeen.

```
#ifndef __API_H__
#define __API_H__
#include <windows.h>

#ifdef BUILD_DLL
   #define DLL_EXPORT __declspec(dllexport)
#else
   #define DLL_EXPORT __declspec(dllimport)
#endif

//DLL-tiedostoksi käännettävien funktioiden esittely
#ifdef __cplusplus
extern "C"
{
#endif

DLL_EXPORT int testMoveMouse(int x, int y);

#ifdef __cplusplus
}
#endif
#endif // __API_H__
```

```
#include "api.h"
//Esimerkkifunktio
int DLL_EXPORT testMoveMouse(int x_coord, int y_coord)
{
   //WinAPIn funktio hiiren liikuttamiseen
   SetCursorPos(x_coord, y_coord);

   if (GetLastError() > 0)
   {
      return 1;
   }

   return 0;
}
```

KUVA 14. Esimerkki ohjelmointirajapinnasta, joka käännetään dll-tiedostoksi. Ylemmässä laatikossa on käännettävän funktion esittely ja alemmassa laatikossa funktion toteutus.

```
from ctypes import *
from xmlrpc.server import SimpleXMLRPCServer

#luetaan dll-tiedosto
mydll = cdll.LoadLibrary('D:\\DLL_Export_Example\\DLL_Export\\bin\\Debug\\DLL_Export.dll')

def moveMouse(x_coord, y_coord):
		#kutsutaan funktiota dll-tiedostosta
		if mydll.testMoveMouse(x_coord, y_coord) == 0:
			return 0
		else:
			return 1

def runXmlRpcServer():
		port = 5050
		address = "localhost"
		server = SimpleXMLRPCServer((address, port))
		print("server address %s" %address)
		print("listening to port %s..." %port)

		#rekisteröidään dll-tiedostoa käyttävä funktio XML-RPC-palvelimelle
		server.register_function(moveMouse)
		server.serve_forever()
		return 0

#käynnistetään XML-RPC-palvelin
runXmlRpcServer()
```

KUVA 15. Python-kielinen XML-RPC-palvelinsovellus (XML_RPC_module), jossa rekisteröidään funktio dll-tiedostosta XML-RPC-asiakassovelluksen käytettäväksi.

```
#include <Python.h>

int main()
{

   PyObject *pName, *pModule, *pDict, *pFunc;
   //Python-tulkin alustus
   Py_Initialize();

   //määritellään kutsuttavan Python-moduuli ja moduulista kutsuttava funktio
   pName = PyUnicode_FromString("XML_RPC_module");
   pModule = PyImport_Import(pName);
   pDict = PyModule_GetDict(pModule);
   pFunc = PyDict_GetItemString(pDict, "runXmlRpcServer");

   //tarkastetaan voiko funktiota kutsua
   if (PyCallable_Check(pFunc))
   {
      //kutsutaan haluttau funktiota
      PyObject_CallObject(pFunc, NULL);
   }
   else
   {
      //tulostetaan virheilmoitus
      PyErr_Print();
   }
   //vapautetaan pModulen ja pNamen varaama muisti
   //pDict ja pFunc ovat lainattuja viittauksia, joten niiden varaamaa muistia ei saa vapauttaa tässä
   Py_DECREF(pModule);
   Py_DECREF(pName);
   Py_Finalize();

   return 0;
}
```

KUVA 16. Python-kielisen moduulin kutsuminen C++-projektista

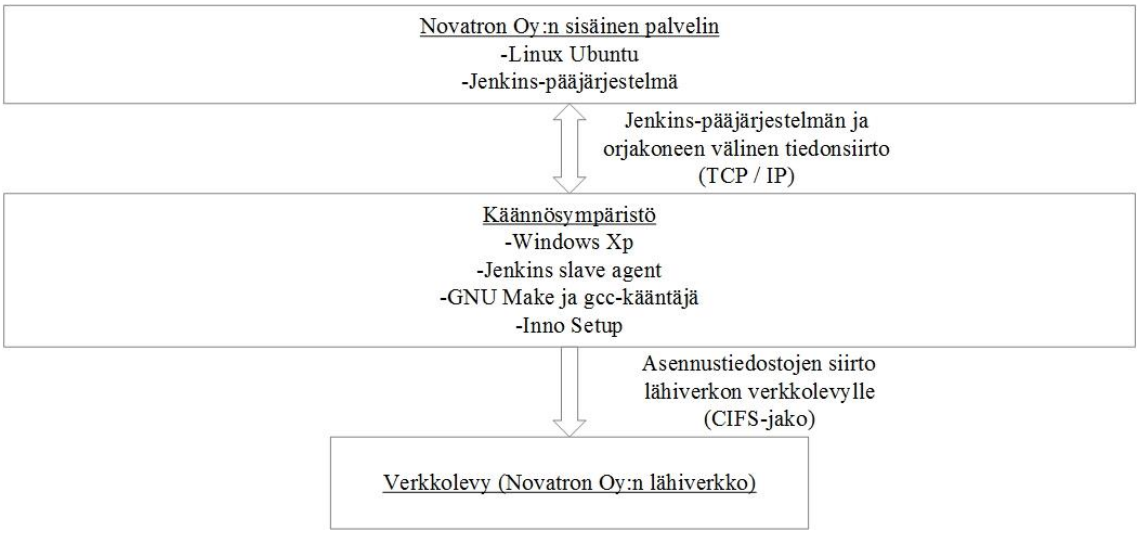
4.5 Ohjelmistoprojektin käännösautomaatio

Automaattisen käännösympäristön suunnittelu oli ensimmäinen vaihe automatisointiprojektissa. Ohjelmistoprojektin käännökseen liittyy paljon tehtäviä, joiden automatisointi tehostaa nopeasti ohjelmistokehittäjien ja testaajien työskentelyä. Koska ensimmäiseksi testattavan järjestelmän käyttöjärjestelmä on Windows XP Embedded, haluttiin käännösympäristön toimivan Windows XP -käyttöjärjestelmän päällä siltä varalta, että myös käännösympäristössä suoritettaisiin testitapauksia. Testattavan ohjelmistoprojektin käännösautomaatiossa hyödynnetään Subversion-versionhallintajärjestelmää. Testattavan järjestelmän ohjelmistoprojektista generoidaan make-tiedostot CBP2make-työkalulla ja käännetään Makella käyttäen gcc-kääntäjää.

Asennustiedostot luodaan Inno Setup -nimisellä ohjelmistolla. Asennustiedoston luomiseksi Inno Setupia varten tulee olla erilliseen iss-määrittelytiedostoon kirjoitettuna mitä tiedostoja luotavaan asennustiedostoon halutaan sisällyttää, sekä mitä asennustiedoston suorituksen yhteydessä halutaan tehdä. Ohjelmiston asennuksen (eli asennustiedoston suorituksen) aikana voidaan esimerkiksi purkaa asennettavan ohjelmiston toiminnan kannalta välttämättömät tiedostot kohdelaitteen massamuistiin, asentaa laitteistoajureita sekä kirjoittaa asennettavan ohjelmiston versionumero tai muita tietoja Windowsin rekisteriin. Inno Setupia voidaan käyttää Windowsin komentorivikäyttöliittymän kautta. Ohjelmisto oli Novatron Oy:llä käytössä jo ennen kuin automaattisen ohjelmistotestauksen kehitys alkoi, joten asennustiedostojen luomista varten tarvittavat määrittelytiedostot olivat jo olemassa. Myös automaattista suoritusta varten tarvittavat komentorivikäskyt oli valmiiksi kirjoitettuna versionhallinnassa oleviin bat-tiedostoihin. Bat-tiedostot sisältävät Windowsin komentorivikäyttöliittymässä suoritettavia käskyjä. Asennustiedostojen luomisen automatisointi tehtiin siis siten, että Jenkinsille määriteltiin tehtävä, jossa suoritetaan jo olemassa oleviin bat-tiedostoihin kirjoitetut komennot.

Käännösautomaatio toimii siten, että Jenkins-pääjärjestelmä tarkastaa kehitettävän ohjelmistoprojektin versionhallintajärjestelmän viiden minuutin välein muutosten varalta. Mikäli Jenkins havaitsee muutoksen, se päivittää ohjelmistoprojektin kokonaisuudessaan versionhallintapalvelimelta orjakoneen (käännösympäristön) kiintolevylle ja generoi projektista make-tiedostot. Kun projekti on onnistuneesti käännetty tietokoneella suoritettavaksi sovellukseksi, luodaan asennustiedostot ja jaetaan nämä asennustiedostot Novatron Oy:n lähiverkkoon. Jenkins mahdollistaa tiedostonsiirron eri palvelimien vä-

lillä ohjelmistoliitännäisten kautta monilla eri tiedonsiirtoprotokollilla (esimerkiksi FTP ja SFTP). Novatron Oy:llä asennustiedostot siirretään lähiverkkoon CIFS-protokollaa käyttäen (kuva 16).



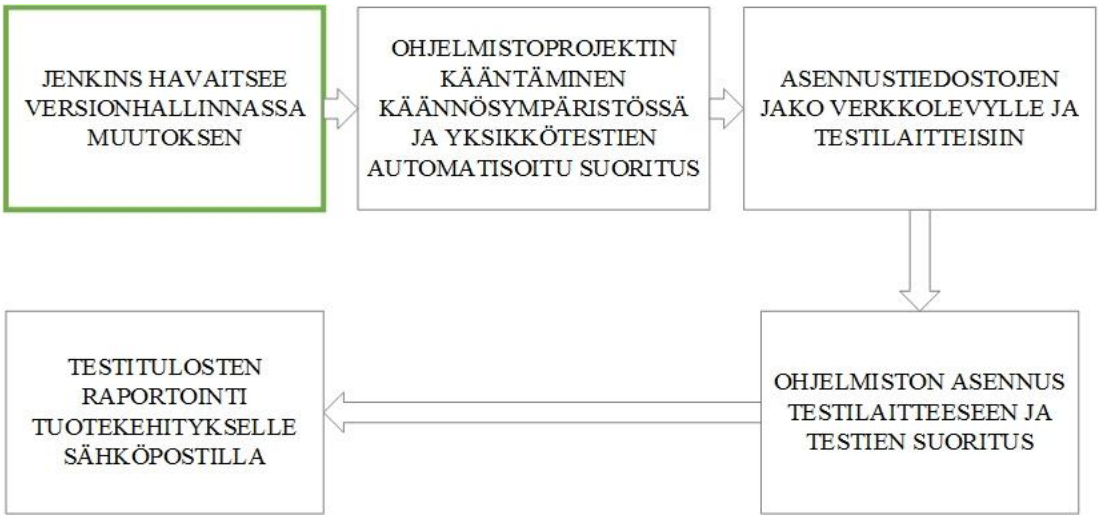
KUVA 16. Käännösautomaatioympäristön rakenne

4.6 Automaattinen testausjärjestelmä

Testausympäristönä Novatron Oy:llä käytetään sitä laitteistoa, jota asiakaskin tulee käyttämään. Tällä menettelyllä pyritään siihen, että testien onnistuessa voidaan olla varmoja siitä, että testatut ominaisuudet toimivat myös asiakkaalla. Erillisen testi-PC:n käyttäminen testausympäristönä vähentää testitulosten luotettavuutta. Testien tulokset voivat poiketa toisistaan, jos ohjelmistoa testataan laitteistokokoonpanoltaan erilaisissa järjestelmissä.

Jotta testattavassa järjestelmässä voidaan Jenkinsin avulla suorittaa automaattisia testejä, tulee siihen asentaa slave agent -sovellus, Python-tulkki sekä Java-suoritusympäristö (slave agent on Java-sovellus). Näiden komponenttien asennus tehdään kertaluontoisesti. Automatisoituja tehtäviä ei voida suorittaa ilman, että nämä sovellukset ovat asennettuna testilaitteessa. Novatron Oy:n testausympäristön testilaitteet eivät tule vaihtumaan, vaan automatisoidut ohjelmistotestit ajetaan aina samoissa laitteissa. Kertaluontoisesti asennettavia työkaluja ei siis tarvitse asentaa uudelleen testausympäristön käyttöönoton jälkeen. Ennen testauksen aloittamista tulee testilaitteeseen siirtää järjestelmän uusimman ohjelmistoversion asennustiedosto ja asentaa testattava ohjelmistoversio. Testatta-

van ohjelmiston asennus on osa automatisoitua testausta. Testattavalle ohjelmistolle tehtiin testausta varten erillinen asennustiedosto, joka pitää sisällään luodun testisovelluksen ja ohjelmistoliitännäisen. Kuvassa 17 on testaus- ja käännösautomaation toiminta lohkokaaviona. Mikäli jokin vaiheista epäonnistuu, ilmoittaa Jenkins siitä aina sähköpostilla tuotekehityshenkilöstölle.



KUVA 17. Testaus- ja käännösautomaation toiminta lohkokaaviona

4.7 Testitulosten raportointi

Kaikessa ohjelmistotestauksessa luotettavan ja selkeän testausraportoinnin tuottaminen on yksi tärkeimmistä kokonaisuuden osista (kuva 18). Jenkinsin suorittamista tehtävistä raportointidataa saadaan kaikesta, mitä testaus- ja käännösautomaatiojärjestelmä tekee. PyTestin generoimien JUnitXML-testiraporttien pohjalta Jenkins osaa muodostaa myös graafisia kuvaajia siitä, kuinka paljon testejä on eri laitteistoissa suoritettu, kuinka suuri osa testeistä on epäonnistunut tai onnistunut sekä kuinka paljon testeihin on kulunut aikaa (kuva 19). Jenkins kerää kaiken järjestelmän konsoli-ikkunaan tulostetun tekstimuotoisen datan yhtenäiseen lokiin. Esimerkiksi ohjelmistoprojektin käännöstehtävästä tallennetaan koko käännösloki Jenkins-pääjärjestelmään. Käyttöliittymää testattaessa lisätään testisovellukseen funktio, joka tallentaa kuvan testattavan laitteen näytöstä testitapauksen suorituksen aikana. Näin voidaan varmistaa, että käyttöliittymä näyttää käyttäjälle siltä niin kuin pitää. Mahdollisimman monipuolinen raportointi testausjärjestelmästä tuotekehityshenkilöstölle lisää testitulosten luotettavuutta. Aina kun testejä automatisoidaan, voidaan myös epäillä itse testien toimivuutta. Mikäli testausjärjestelmän

ulosanti on liian suppeaa, ei voida täysin luottaa saatuihin testituloksiin ja tämä tekee testausjärjestelmän käytöstä lähes hyödytöntä.

Failed

TestApplication.test_resetvalues (from)

Took 6 sec.

Error Message

test failure

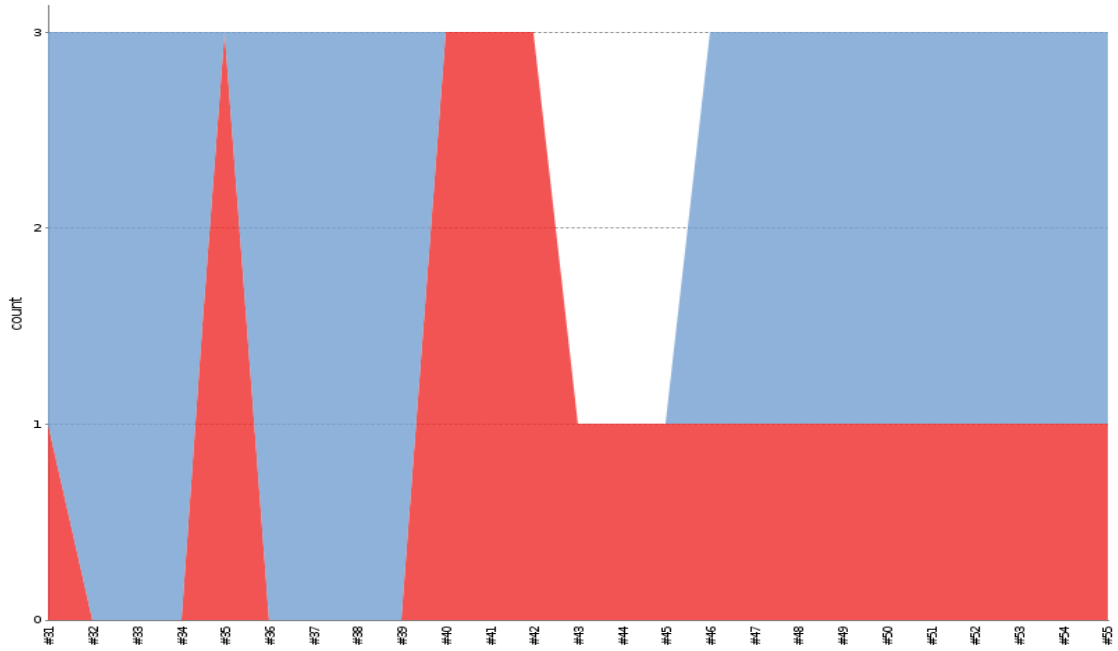
Stacktrace

```
def test_resetvalues():

>  assert testiLuokka.testResetValues() == 1;

E assert 0 == 1

E +  where 0 = <bound method SmokeTestClass.testResetValues of <Testaus.SmokeTestClass object at 0x01086430>>()

E +    where <bound method SmokeTestClass.testResetValues of <Testaus.SmokeTestClass object at 0x01086430>> = <Testaus.SmokeTestClass
object at 0x01086430>.testResetValues

TestApplication.py:29: AssertionError
```

Standard Output

```
!!!!!!!!!!!!!!RESET VALUES TEST!!!!!!!!!!!!!!
```

KUVA 18. Jenkinsin tallentama PyTest-testiraportti epäonnistuneesta testitapauksesta. Raportissa näkyy, mikä olettamus on epäonnistunut sekä mitä testin aikana on kirjoitettu konsoliin.



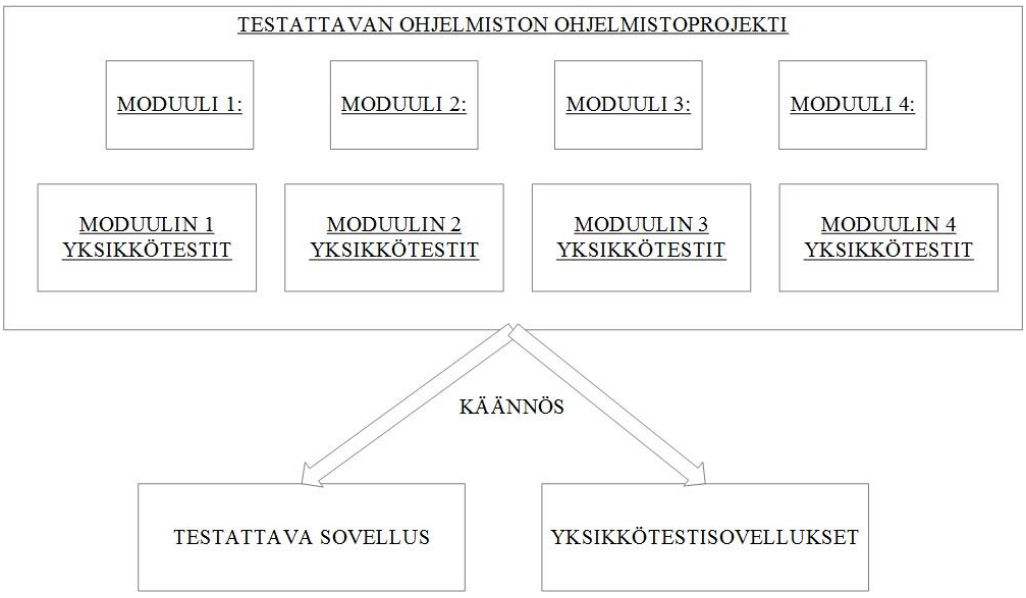
KUVA 19. Jenkinsin generoima kuvaaja testituloksista testitapausten lukumäärän suhteen. Pystyakselilla testien kokonaismäärä ja vaaka-akselilla testin järjestysnumero, punainen kuvaa epäonnistunutta testitapausta ja sininen onnistunutta.

4.8 Yksikkötestaus osana automaatiota

Yksikkötestaukseen eri ohjelmointikielillä on olemassa useita testauskirjastoja, joiden käyttäminen vastaa hyvin pitkälti PyTest-työkalun käyttämistä. Valtaosa yksikkötesta-

uksen testauskirjastoista perustuu samankaltaiseen xUnit-arkkitehtuuriin, joka pohjautuu 1990-luvun lopun Java-kieliseen JUnit-nimiseen yksikkötestauskirjastoon. [1]

Novatron Oy:n yksikkötestauksen testaustyökaluksi valittiin Googlen Googletest (myöhemmin gtest). Yksikkötestien kirjoittamiseen C++-kielellä on olemassa useita muitakin vaihtoehtoja eikä niiden käyttäminen poikkea toisistaan merkittävästi. Tämän projektin yhteydessä haluttiin kuitenkin valita yksi työkalu, jota kaikki kehittäjät tulevat käyttämään. Yksikkötestien suunnittelu, kirjoitus ja suoritus ovat Novatron Oy:llä sen ohjelmistokehittäjän vastuulla, joka kehittää uutta toimintoa. Yksikkötestit tullaan kuitenkin ajamaan myös toisen kerran ohjelmistoprojektin automaattisen käännöksen yhteydessä. Yksikkötestit suoritetaan muusta ohjelmallisesta testauksesta poiketen käännösympäristössä johtuen testien erilaisesta luonteesta. Python-kieliset testit suoritetaan testattavan järjestelmän ajon rinnalla. Python-kielisen testisovelluksen kehittäminen on oma erillinen projektinsa ja sillä testataan suuria kokonaisuuksia kerralla. Yksikkötestit ovat C++-kielisiä ja ne ovat osa kehitettävän ohjelmiston ohjelmistoprojektia. Yksikkötestiprojekteista käännetään omat sovelluksensa samalla kun testattavakin ohjelmisto käännetään (kuva 20).



KUVA 20. Yksikkötestit testattavassa sovelluksessa

Yksikkötestisovellukset ovat konsolisovelluksia, joissa suoritetaan yksikkötestitapausten funktiot. Testausautomaatiossa yksikkötestisovellukset ajetaan aina ohjelmistoprojektin käännöksen jälkeen, jotta voidaan varmistua siitä, että yksikkötestit varmasti tulevat aina ajetuksi. Yksikkötestauskirjastoissa on toteutukset erilaisille väittämätyypeil-

le, testien parametrisoinnille ja testitapausten tulosten raportoinnille. Gtestillä luotu yksikkötestisovellus tulostaa oletuksena konsoli-ikkunaan suoritetut testitapaukset ja niiden tulokset. Tämän lisäksi myös gtestillä pystytään testituloksista generoimaan JUnitXML-muotoisia raportteja. Kuvassa 21 on kaksi yksinkertaista yksikkötestiä, joissa verrataan luokkametodien paluuarvoja olettamiin. Toinen yksikkötesteistä onnistuu ja toinen epäonnistuu (kuva 22).

```
TEST(TestCase1, SummaTest)
{
	//TestCase1 testaa summa-nimistä funktiota, jonka paluuarvon tulisi olla 4
	Laskin testiLaskin;
	testiLaskin.asetaLuvut(2,2);
	EXPECT_EQ(4, testiLaskin.summa());

}

TEST(TestCase2, ErotusTest)
{
	//TestCase2 testaa erotus-nimistä funktiota, jonka paluuarvon tulisi olla 0
	//väittämään on laitettu väärä luku, jotta testiraporttiin saadaan epäonnistunut testitulos
	Laskin testiLaskin;
	testiLaskin.asetaLuvut(2,2);
	EXPECT_EQ(4, testiLaskin.erotus());

}
```

KUVA 21. Yksinkertaisia yksikkötestiesimerkkejä

```
Running main() from gtest_main.cc
[==========] Running 2 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 1 test from TestCase1
[ RUN      ] TestCase1.SummaTest
[       OK ] TestCase1.SummaTest (0 ms)
[----------] 1 test from TestCase1 (0 ms total)

[----------] 1 test from TestCase2
[ RUN      ] TestCase2.ErotusTest
d:\testi\laaskintestit\laskintestit\testit.cpp(19): error: Value of: testiLaskin.erotus()
  Actual: 0
Expected: 4
[  FAILED  ] TestCase2.ErotusTest (1 ms)
[----------] 1 test from TestCase2 (1 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 2 test cases ran. (2 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] TestCase2.ErotusTest

 1 FAILED TEST
```

KUVA 22. Gtestin yksikkötestiraportti

5 TULOKSET

5.1 Vaikutukset työskentelyyn

Kesällä 2013 automatisoitiin testattavan ohjelmistoprojektin kääntäminen sekä asennuspakettien luominen ja jakaminen Novatron Oy:n lähiverkkoon. Testausjärjestelmän kehitys eteni siihen vaiheeseen, että automaattisten testien suunnittelu voitiin aloittaa. Ohjelmistotestauksen automatisointiin vaadittavat toiminnot saatiin valmiiksi. Selvästi suurin käytännön hyöty kehitystyön ensimmäisen vaiheen jälkeen oli käännösautomaation käyttöönotto. Aikaisemmin ohjelmistoprojektin kääntäminen oli täysin kehittäjien omalla vastuulla, eikä ollut täyttä varmuutta siitä, toimiiko kehittäjän tietokoneella oleva ohjelmistoversio versionhallinnassa olevan ohjelmakoodin kanssa. Lisäksi aikaa vievä asennuspakettien generoinnin automatisointi vähensi ohjelmistokehittäjien päivittäisessä työssä esiintyviä rutiininomaisia tehtäviä. Tämä jätti enemmän aikaa uusien ominaisuuksien toiminnan kehittämiseen, joka on ohjelmistokehittäjän ensisijainen tehtävä. Asennustiedostojen jakamisen myötä koko yrityksen henkilöstöllä on mahdollisuus hakea samasta paikasta aina uusimman ohjelmistoversion asennustiedosto. Aikaisemmin asennustiedostojen jako oli vähemmän keskitettyä ja epäyhtenäistä.

Lisäksi testausautomaation kehityksen yhteydessä tehtiin uusi raportointipohja manuaaliselle testaukselle, jonka tavoitteena oli tehdä testitulosten dokumentoinnista yhdenmuotoista yrityksen sisällä. Raportointipohja on selkeä ja yksiselitteinen järjestelmätestauksen ohje, jota voidaan myös käyttää tuotekehityksen ohjekirjana myös automaattisten testien suunnittelussa.

5.2 Ensimmäiset automatisoidut testit

Vaikka testijärjestelmän soveltaminen Novatron Oy:n tuotekehityksessä ei vielä tätä raporttia kirjoitettaessa ole jokapäiväistä, saatiin järjestelmälle kuitenkin ensimmäiset testitapaukset, joilla voitiin varmistaa testausjärjestelmän toimivuus. Ensimmäiset testit pitivät sisällään testattavan järjestelmän käynnistämisen, sisäänkirjautumisen, järjestelmän mittaaman korkeusarvon nollaamisen 2D-tilassa sekä järjestelmän sammuttamisen.

5.3 Järjestelmän jatkokehitys

Seuraavat askeleet järjestelmän jatkokehityksen kannalta ovat automatisoitujen testitapausten määrittely ja toteutus. Tavoitteena on suunnitella, mitä asioita automaattisella testauksella halutaan ensin varmentaa ja miten järjestelmätason ohjelmallinen testaus tullaan tekemään. Lisäksi testausjärjestelmän laajennussuunnitelmiin kuuluu esimerkiksi työkonesimulaattorin hankinta ulkopuoliselta yritykseltä. Työkonesimulaattori mahdollistaa reaalimaailmaa vastaavan simulaation hyödyntämisen testauksessa. Testattaessa järjestelmää simulaattorilta saadaan anturidataa, joka vastaa oikeaan kaivinkoneeseen asennetun järjestelmän datavirtaa. Ohjelmistokehityksen työkaluihin ja automatisoituun testausjärjestelmään voidaan myös lisätä muistinhallinnan, järjestelmän kuormituksen ja lähdekoodin analysointiin tarkoitettuja työkaluja.

Järjestelmän soveltaminen muissa tuotekehitysprojekteissa tulee ensisijaisesti liittymään ohjelmistoprojektien käännöksen automatisointiin. Ohjelmistoprojektin automaattinen kääntäminen on Jenkinsin avulla helppoa ja se hyödyntää nopeasti ohjelmistoprojektin parissa työskentelevän henkilöstön työntekoa.

6 YHTEENVETO

Työn tavoitteena oli kehittää Novatron Oy:n tuotekehitykselle järjestelmä, joka mahdollistaa ohjelmistoprojektien kääntämisen ja ohjelmallisen testauksen automatisoinnin. Lisäksi haluttiin yhtenäistää ja tehostaa Novatron Oy:n ohjelmistotestausta tuotteiden koko kehityskaaren ajaksi. Testien ja käännösten automatisointijärjestelmän perustoiminnot toteutettiin kokonaisuudessaan. Nykyisellään käytössä on yhden ohjelmistoprojektin käännösautomaatio ja rakenteet automaattisen ohjelmistotestauksen suunnittelua ja toteutusta varten. Automatisointijärjestelmän kehitys jatkuu vielä vuoden 2013 jälkeenkin, niin pitkään kuin ohjelmistotestaukselle on tarvetta. Lisäksi järjestelmän soveltamista muihin projekteihin harkitaan.

Työ oli projektina vaikea, koska aiheeseen liittyvää kokemusta ei työhön osallistuvilla henkilöillä juuri ollut. Järjestelmän kehitys eteni kuitenkin jatkuvasti, eikä tietotaidon puute muodostunut ylitsepääsemättömäksi esteeksi. Ohjelmistotestauksen teoriaan ja käytäntöihin tutustuminen on hyvin yleishyödyllistä tietoa, josta on hyötyä niin matalan kuin korkeankin tason ohjelmointityössä. Lisäksi työn aikana pääsi tutustumaan sulautettuihin mittausjärjestelmiin. Erityisesti perehtyminen anturi- ja paikannustekniikan sovelluksiin opetti paljon uutta.

LÄHTEET

1 Fowler, M. Jatkuva integraatio, testivetoinen kehitys ja xUnit. http://www.martinfowler.com/articles/continuousIntegration.html, http://www.martinfowler.com/bliki/TestDrivenDevelopment.html ja http://www.martinfowler.com/bliki/Xunit.html

2 Holger K. PyTest opas. http://pytest.org/latest/

3 Hower R. Ohjelmistotuotannon ja testauksen opas. http://www.softwareqatest.com/index.html

4 Joensuun yliopisto, tietojenkäsittelyn laitos, ohjelmistotuotannon tietokeskus. Integrointi- ja järjestelmätestaus. http://cs.joensuu.fi/tSoft/testaus.htm

5 Novatron Oy:n kotisivu. http://www.novatron.fi

6 Python Software Foundation. Python-ohjelmointikielen virallinen sivusto. http://www.python.org

7 Reitz, K. The Hitchhiker's Guide to Python - Python opas. http://docs.python-guide.org/en/latest/

8 Smart, J. F. Jenkins the definitive guide - Jenkins CI opaskirja. Ladattu 21.2.2013. http://www.wakaleo.com/books/jenkins-the-definitive-guide.