Esa Hannila

OHJELMISTOJEN AUTOMAATTINEN TESTAUSJÄRJESTELMÄ

OHJELMISTOJEN AUTOMAATTINEN TESTAUSJÄRJESTELMÄ

Esa Hannila
Opinnäytetyö
Kevät 2018
Tieto- ja viestintätekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehitys

Tekijä: Esa Hannila
Opinnäytetyön nimi: Ohjelmistojen automaattinen testausjärjestelmä
Työn ohjaaja: Timo Vainio
Työn valmistumislukukausi ja -vuosi: Kevät 2018
Sivumäärä: 58 + 13 liitettä

Opinnäytetyön aiheena oli suunnitella ja toteuttaa yrityksen ohjelmistokehityksen tarpeisiin soveltuva automaattinen testausjärjestelmä. Suunnittelun tavoitteena oli määritellä ja havainnollistaa automaattinen testausjärjestelmä, jota voidaan jatkokehittää ja johon voidaan liittää tulevaisuudessa tuotannon testilaitteistot ja sovellukset.

Testausjärjestelmän suunnittelun ja kehityksen lähtökohtana oli käyttää mahdollisimman paljon valmiita sovelluksia, jotka ovat soveltuvia testausjärjestelmän käyttöön. Järjestelmän tuli voida toimia ilman ulkoisia palveluntarjoajia tai linkityksiä sisäverkon ulkopuolisiin sovelluksiin.

Testausjärjestelmän suunnittelu ja toteutus tehtiin projektiluontoisesti. Projektin suunnitteluvaiheessa testausjärjestelmästä tuotettiin projektisuunnitteludokumentti. Projekti eteni suunnittelusta määrittelyvaiheeseen, jossa testausjärjestelmälle asetettiin yrityksen tarpeita vastaavat vaatimusmäärittelyt ja käytettävät työkalut. Määrittelyvaiheen tuloksena tuotettiin tekninen sekä toiminnallinen määrittelydokumentti. Määrittelyvaiheen jälkeen suoritettiin käytännön toteutustyö.

Projektin toteutus onnistui hyvin ja valmistui projektisuunnitelman mukaisesti aikataulussa. Käytännön toteutuksessa tuotettiin projektille asetetut vaatimukset ja järjestelmä toimi odotusten mukaisesti.

Testausjärjestelmän kehitystä jatketaan projektin päättymisen jälkeen liittämällä opinnäytetyön ulkopuolelle jääneet käytännön toteutuksen osuudet. Järjestelmää tullaan jatkojalostamaan yrityksen tuotannon käyttöön myöhemmässä vaiheessa.

Asiasanat: ohjelmistokehitys, testaus, automatisointi, versionhallinta

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, software engineering.

Author(s): Esa Hannila
Title of thesis: Automated testing system for software
Supervisor(s): Timo Vainio
Term and year when the thesis was submitted: Spring 2018
Pages: 58 + 13 appendices

The subject of this thesis was to design and implement automatic testing system for the company's software development needs. The goal of design was to define and visualize automated testing system that is available for further development to use on the production testing.

The start points of designing testing system was to use as much as possible of complete applications and tools that fits in system needs. Testing system has to ran without external network applications or authentication methods.

Thesis design and implementation part was completed in project style. At planning state of the project, a project planning document was produced. Technical- and functional specification documents was produced at the project definition phase. After definition state implementing was started.

Project was completed successfully without major problems. Automated testing system filled all requirements that were set in project designing phase. System worked as expected.

Testing system development will be continued after the thesis project by implementing system parts that were excluded from practical part of the thesis. The testing system will be further developed for the production testing uses.

Keywords: software development, testing, automation, version-control

ALKULAUSE

Haluan kiittää opinnäytetyön toimeksiannosta työnantajaani Nukute Oy:tä sekä teknistä projektipäällikköä Juha-Pekka Syrjälää työn ohjaamisesta. Kiitän myös Oulun ammattikorkeakoulun opettajia, Timo Vainiota opinnäytetyön ohjaamisesta sekä Tuula Hopeavuorta opinnäytetyön kieliasun ja tekstin ohjaamisesta.

Erityinen kiitos vaimolleni kaikesta tuesta ja perheen hyvinvoinnista huolehtimisesta sekä opinnäytetyön teon mahdollistamisesta.

Oulussa 14.2.2018

Esa Hannila

SISÄLLYS

TIIVISTELMÄ 3

ABSTRACT 4

ALKULAUSE 5

SISÄLLYS 6

SANASTO 9

1 JOHDANTO 11

2 OHJELMISTOTESTAUS 12

2.1 Testauksen suunnittelu 12

2.2 Testitapausten suunnittelu 13

2.3 Testauksen vaiheet 13

2.3.1 Moduulitestaus 14

2.3.2 Integrointitestaus 15

2.3.3 Järjestelmätestaus 15

2.4 Testauksen laatu 15

2.5 Testauksen seuranta 16

3 TESTAUSANALYYSI 17

3.1 Staattinen analyysi 17

3.2 Dynaaminen analyysi 17

4 TESTAUSMENETELMÄT 19

4.1 Valkolaatikkotestaus 19

4.2 Harmaalaatikkotestaus 20

4.3 Mustalaatikkotestaus 20

5 TIETOTURVA 21

5.1 Tietoturvasuunnitelma 21

5.2 Tietoturvatestaus 21

6 TESTAUKSEN AUTOMATISOINTI 23

6.1 Automatisoinnin suunnittelu 23

6.2 Testitapauksen automatisointi 24

6.3 Testausjärjestelmän alusta 24

6.4 Jatkuva integraatio 24

6.5 Tietoturvatestaus 25

6.5.1 Automatisointi 25
6.5.2 Staattinen analysointityökalu (SAST) 26
6.5.3 Dynaaminen analysointityökalu (DAST) 26
6.6 Yleiset ongelmat testauksen automatisoinnissa 26
7 LÄÄKINNÄLLISTEN LAITTEIDEN TESTAUKSEN ERITYISVAATIMUKSET 28
7.1 Vaatimusten mukaisuuden osoitus 28
7.2 Lääketieteellisen laitteen testaus 29
8 TYÖKALUT 30
8.1 Gogs-versionhallinta 30
8.2 Jenkins-automatisointijärjestelmä 30
8.3 GitLab CE 30
8.4 GitLab-Runner 31
8.5 Docker CE 31
9 TESTAUSJÄRJESTELMÄN SUUNNITTELU JA TOTEUTUS 33
9.1 Järjestelmän suunnittelu 33
9.1.1 Järjestelmäkuvaus 34
9.1.2 Käyttötapausten suunnittelu 34
9.1.3 Toiminnan suunnittelu 36
9.1.4 Järjestelmäarkkitehtuurin suunnittelu 37
9.2 Työkalujen valinta 39
9.3 Järjestelmän toteutus 40
9.3.1 GitLab CE 40
9.3.2 Demo-ohjelma 41
9.3.3 GitLab CI -työkalun käyttöönotto 42
9.3.4 GitLab-Runner 43
9.3.5 Docker CE 44
9.4 Automatisoinnin toteutus 48
9.4.1 Node.js 49
9.4.2 Android-sovelluksen testaus 51
9.4.3 Sulautetun laitteen testaus 52
9.4.4 Toteutuksen tarkastelu 53
10 YHTEENVETO 54
LÄHTEET 55

LIITTEET

SANASTO

Apache-lisenssi	Avoimen lähdekoodin lisenssi, jossa lisenssin alaista ohjelmistoa tai järjestelmää voi käyttää, kopioida, muokata, sulauttaa, julkaista, jakaa ja luovuttaa tai myydä rajoituksetta. Lisenssissä on vapautettu myös mahdolliset patentit.
Bash	Komentotulkki (Unix shell)
Chai	Node.js-ohjelman testaukseen hyödynnettävä vertailutyökalu
CD	Continuous deployment, sovelluksen automatisoitu julkaisu, joka voidaan suorittaa esimerkiksi ohjelmakoodien merkitsemin yhteydessä testausten suorittamisen jälkeen
CI	Continuous integration, sovelluksen moduulien integrointi kokonaisjärjestelmään automatisoidulla testauksella
DAST	Dynamic Analysis Scanning Tools, dynaamisen analysoinnin skannaustyökalut
FreeBSD	Usealle alustalle soveltuva käyttöjärjestelmä
Git	Versionhallintaohjelmisto
Instrumentoitu testaus	Instrumentoidussa testauksessa simuloidaan todellisen henkilön käyttäytymistä ohjelmassa syötteiden antamisella, painikkeiden painamisilla, navigoinneilla ja muilla mahdollisilla keinoilla.
Iteraatio	Ketterissä ohjelmistokehitysmenetelmissä tehtävät jaotellaan tietynkestoisiin työvaiheisiin, eli iteraatioihin

IVD	In vitro -diagnostiikka, potilaasta tai terveestä henkilöistä otetuista lääketieteellisistä näytteistä tehtävä tutkimus
macOS	Graafinen käyttöjärjestelmä
MIT-lisenssi	Avoimen lähdekoodin lisenssi, jossa lisenssin alaista ohjelmistoa tai järjestelmää voi käyttää, kopioida, muokata, sulauttaa, julkaista, jakaa ja luovuttaa tai myydä rajoituksetta.
Mocha	Node.js-ohjelman testauksen kehitysympäristö
Node.js	Googlen avoimen lähdekoodin V8 Javascript -kone
NPM	Node.js Package Manager, Node.js-pakettihallintajärjestelmä
Ohjelmistokehityshaara	Versionhallintajärjestelmissä käytetty looginen ohjelmistokehityksen haara, jossa ohjelmiston kehitystyötä tehdään erillisenä muista haaroista
Penetraatiotestaus	Ohjelmiston testaamista tietoturvariskien varalta. Suoritetaan samoja toimenpiteitä kuin aidoissa hyökkäyksissä aiheuttamatta haittaa ohjelmistolle
Pipeline	GitLab CE -järjestelmän CI-työkalun työjono
Pöytätestaus	Ohjelmistosuunnittelija selittää toiselle ohjelmiston tarkoitusta ja ongelmaa, kuitenkin itse vian havaiten
SAST	Static Analysis Scanning Tools, staattisen analysoinnin työkalut
Ubuntu	Linux-käyttöjärjestelmän distribuutio
UML	Unified Modeling Language, graafinen mallinnuskieli
Unicorn daemon	Moniprosessiarkkitehtuuri, joka parantaa prosessorin ytimien hyödyntämistä.

1 JOHDANTO

Ohjelmistojen automaattinen testausjärjestelmä on tärkeä osa lääketieteellisten laitteistojen ja ohjelmistojen hyväksymis- ja laadunvarmistusprosessia. Testausjärjestelmän tarkoituksena on varmentaa kehitettävän ohjelmiston sekä laitteiston turvallisuus ja soveltuvuus lääketieteellisessä käytössä.

Tämän opinnäytetyön tavoitteena oli suunnitella ja toteuttaa automaattinen testausjärjestelmä, joka täyttää lääketieteellisen järjestelmän testauksesta määritellyt vaatimukset. Testattavat laitteistot ja ohjelmistot ovat Nukute Oy:n kehitteillä oleva järjestelmäkokonaisuus, jonka tarkoitus on avustaa diagnosoimaan maailmanlaajuisesti merkittävää sairautta, uniapneaa.

Opinnäytetyö toteutettiin projektina, jonka suunnitteluvaiheessa tuli toteuttaa projektisuunnitelma sekä tekninen ja toiminnallinen määrittelydokumentti. Suunnittelutyössä tuli hyödyntää UML-mallinnuskielen kaavioita helpottamaan testausjärjestelmän kokonaisuuden hahmottamista.

Yrityksellä ei ollut käytettävissä ohjelmistojen testaukseen soveltuvaa alustaa tai järjestelmää, joten suunnittelutyö aloitettiin alusta. Testausjärjestelmän suunnittelutyössä huomioitiin yrityksessä kehitteillä olevalle ohjelmistolle asetetut vaatimukset, kuten automatisoitavat yksikkö-, moduuli- ja integraatiotestaukset. Testit tuli suorittaa automatisoidusti aina versionhallintajärjestelmään tallennettujen lähdekoodimuutosten jälkeen.

Testausjärjestelmän kehittämisessä hyödynnettiin avoimen lähdekoodin ohjelmistoja. Järjestelmän alusta tuli suunnitella siten, että sitä voidaan myöhemmässä vaiheessa jatkojalostaa tuotannon testauksen ja laadunvarmistuksen käyttöön.

Tavoitteena oli suunnitella ja toteuttaa järjestelmäosien yksinkertaiset testit, jotta myös muut työntekijät voisivat niitä hyödyntää jatkokehityksessä. Opinnäytetyön käytännön osuuden ulkopuolelle rajattiin lähdekoodien katselmointi- ja raportointityökalujen asennukset ja kehitystyö sekä laitteiston käännös- ja testaustoimet ja niiden suunnittelu.

2 OHJELMISTOTESTAUS

Ohjelmisto on aina yksilöllisesti tuotettu kokonaisuus, joka voi koostua useista ohjelmamoduuleista ja osakokonaisuuksista. Ohjelmistokehityksen määrittelyvaiheessa ohjelmistolle asetetaan sen toiminnalliset ja ei-toiminnalliset vaatimukset. Määrittelyvaiheessa unohdetaan usein huomioida ohjelmistotestaus, joka voi olla yhtä monimutkainen kokonaisuus kuin itse ohjelmistokin. (1, s. 10.)

Ohjelmistotestaus on tärkeä osa ohjelmistotuotantoa. Ajoissa suunniteltu ja toteutettu ohjelmistotestaus vähentää merkittävästi kustannuksia sekä parantaa ohjelmiston laatua. Testauksen tavoitteena on suorittaa ohjelmistolle riittävän kattavat testaukset, jotta ohjelmiston vaatimusten mukainen toiminto voidaan varmistaa ja ohjelmakoodissa ilmentyneet virheet voidaan havaita ja korjata mahdollisimman aikaisessa vaiheessa. (2, s. 4.) Ohjelmistotestauksella voidaan mitata myös ohjelmiston ei-toiminnallisia vaatimuksia, esimerkiksi ohjelmiston suorituskykyä (2, s. 14).

Testauksen suunnittelun ja toteutuksen kannalta on tärkeää, että suunnittelija on tietoinen ohjelmiston käyttötarkoituksesta, tuotteen käyttötavasta sekä siitä, kuka ohjelmistoa käyttää. Näiden lisäksi testauksen suunnittelijan tulee tietää ohjelmistolle asetetut vaatimusmäärittelyt. (3, s. 193–194.)

Ohjelmistotestaus on vaiheittain toteutettava prosessi, jossa usein ensimmäinen vaihe on testattavan ohjelmiston kääntäminen ohjelmistokielelle soveltuvan käännösohjelman avulla tietokoneen ymmärtämään muotoon. Ohjelmiston käännöksellä varmennetaan ohjelmoitsijan kirjoittaman syntaksin oikeellisuus sekä ohjelman suoritettavuus. (3, s. 193.)

2.1 Testauksen suunnittelu

Ohjelmistotestauksen ensimmäinen vaihe on suunnittelu. Sillä varmistetaan testausten kattavuus. Testauksen suunnittelussa huomioidaan ohjelmistolle asetetut vaatimukset, olosuhteet ja tilanteet, joissa testattavaa ohjelmistoa käytetään. Huolellinen suunnittelu on avain testattavan ohjelmiston toimintavarmuuden ja vaatimusten mukaisen toiminnan varmistamista. (2, s. 28.)

Testausta suunniteltaessa suunnittelijan tavoitteena on havaita ohjelmiston toimintojen kannalta tärkeitä kohtia, jotka ovat testattavissa ja vaativat sitä. Testaussuunnitelmassa määritetään mitä ja miten testaustoimenpiteitä ohjelmistolle suoritetaan. Suunnitelmassa määritetään myös testauksesta odotettavissa olevat testitulokset. (2, s. 28.)

2.2 Testitapausten suunnittelu

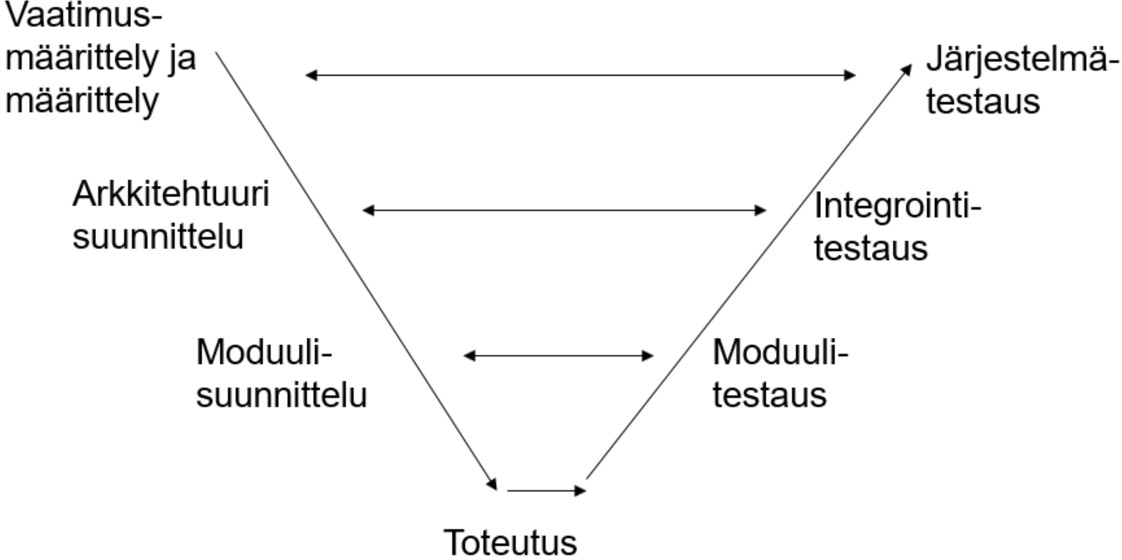
Testitapausten suunnittelu toteutetaan ohjelmistotestauksen suunnittelun yhteydessä. Testitapaukselle määritetään testauksessa käytettävät menetelmät, testattavat tilanteet, testin syötteet ja odotettu tulos. (1, s. 9.)

Testitapausten suorittaminen voidaan toteuttaa manuaalisesti tai automatisoidusti. Testitapauksen luonne, odotettu tulos ja testin toistettavuus määrittävät sen, onko testi automatisoitavissa tai onko testin suorittamista järkevää automatisoida. (1, s. 12.)

Testitapauksen automatisointia mietittäessä tarkastellaan testistä odotettavissa olevaa tulosta ja sitä, voidaanko testin tulosta mitata tai verrata koneellisesti saatuun tulokseen. Hyvä esimerkki automatisoitavasta testistä voisi olla testitapaus, jossa testin tuloksena on tiedosto, jonka tulisi olla täsmälleen odotetun tiedoston kaltainen. Odotetun tuloksen ja testauksen tuloksena saadut tiedostot ja niiden samankaltaisuus voidaan todentaa lukemalla tiedoston sisältö tavu kerrallaan. Tällaisen testauksen suorittaminen manuaalisesti on käytännössä mahdotonta. (4, s. 11.)

2.3 Testauksen vaiheet

Testitapaukset jaotellaan perinteisessä ohjelmistokehityksessä V-mallin mukaisiin vaiheisiin testausprosessin hallittavuuden parantamiseksi (kuva 1). V-mallissa testaus on jaoteltu moduulitestaukseen, integrointitestaukseen ja järjestelmätestaukseen. Testiprosessissa jokainen vaihe suoritetaan kokonaisuudessaan ennen seuraavaan vaiheeseen siirtymistä siten, että vaiheessa suoritetut testit eivät havainneet virheellistä toimintaa tai vikaa ohjelmakoodissa. (1, s. 15.)



KUVA 1. Testauksen V-malli (5, s. 14)

Testivaiheessa aiheutunut virhe tai väärä toiminta pysäyttää testin suorittamisen kyseiseen tasoon ja virheestä tuotetaan testiraportti, jota hyödyntämällä ohjelmistosuunnittelija voi paikantaa virheen ja tehdä tarvittavat muutokset virheen korjaamiseksi.

Virheen korjauksen jälkeen suoritetaan kaikki virhettä edeltäneet testitapaukset uudelleen, jotta tehdyt muutokset eivät aiheuttaneet uusia virheitä alemman tason testeissä (6, s. 13).

2.3.1 Moduulitestaus

Moduulitestaus suoritetaan testauksen ensimmäisenä vaiheena, jossa yksittäisten moduulien toimintoja ja funktioita ohjelmistosuunnittelija testaa. Moduulitestauksen määrittelystä ja suunnittelusta vastaa testaaja, koska suunnittelija tuntee moduulin ohjelmakoodin rakenteen ja sen tavoitellut toiminnot ja tulokset. (6, s. 11.)

Moduulitestausta suorittaessa joudutaan usein toteuttamaan moduulin ympäristöä simuloivaa ohjelmakoodia, joilla korvataan moduulin tarvitsemia muita moduuleita ja funktioita (6, s. 11).

2.3.2 Integrointitestaus

Integraatiotestauksessa suoritetaan testausta moduulien ja niistä muodostuvien ryhmien julkisia rajapintoja ja toimintoja vasten. Testivaihe suoritetaan, kun moduulitestaus on suoritettu ilman havaittuja virheitä. Integraatiotestauksen suunnittelijan tulee tuntea moduulien rajapinnat ja niiden odotettu toiminta ja tulokset. Testauksen suorittaja suunnittelee ja toteuttaa tarvittavat ympäristöä simuloivat moduulit ja osakokonaisuudet. (3, s. 193.)

Testauksessa havaitut virheet raportoidaan ohjelmistosuunnittelijalle, jotta suunnittelija voi paikantaa ja korjata vian aiheuttaneen ohjelman. Muutosten jälkeen ohjelmistotestaus aloitetaan alusta varmistaakseen, että muutokset eivät aiheuta moduulitasolle uusia virheitä. (3, s. 189.)

2.3.3 Järjestelmätestaus

Järjestelmätestauksessa suoritetaan kokonaisten ohjelmistojen toimintojen testausta. Testivaihe suoritetaan, kun moduuli- ja integrointitestausvaiheet on suoritettu ilman havaittuja virheitä. Järjestelmätestauksen suunnittelijan tulee tuntea ohjelmiston vaatimukset voidakseen suunnitella riittävän kattavan testauksen. (3, s. 194.)

Tässä testausvaiheessa suoritetaan myös ei-toiminnallisten vaatimusten mukaisia testauksia. Ei-toiminnallisilla testeillä voidaan esimerkiksi varmistaa ohjelmiston suorituskyky tai kuormitettavuus. Suorituskykyä testaamalla voidaan selvittää esimerkiksi ohjelmiston skaalautuvuus ja ohjelmiston suoriutuminen suuren datamäärän käsittelystä yhtäaikaisesti. (3, s. 194.)

2.4 Testauksen laatu

Ohjelmistotestauksen huolellinen suunnittelu ja testien jatkuva kehittäminen ovat avain laadukkaaseen testaamiseen. Testauksen laatua voidaan mitata siten, kuinka kattavasti ohjelman eri toiminnot on testattu, ja sen testauksen vähimmilläänkin tulisi kattaa kaikki ohjelman toiminnot tietyillä syötteillä oletetuissa käyttöympäristöissä. (7.)

Ohjelmistojen testausta ei voida käytännössä koskaan suorittaa täydellisesti. Testien täydellinen kattavuus tarkoittaisi ohjelmiston kaikkien toimintojen testaamista kaikilla mahdollisilla syötteillä jokaisessa mahdollisessa ympäristössä. (7.)

Testauksen automatisoinnilla voidaan parantaa testauksen laatua merkittävästi, kun suoritettavat testit voidaan ajaa automatisoidusti aina ohjelmakoodin muutosten yhteydessä sekä ajoitetusti esimerkiksi öisin. Testauksen automatisoinnista kerrotaan enemmän luvussa 6.

2.5 Testauksen seuranta

Testauksen seurannan tarkoituksena on tarkkailla ohjelmistotestauksessa toistuvasti suoritettavien testien havaitsemia virheitä ja niiden määrien muutoksia sekä missä vaiheessa virheet ilmenevät. Seurannan avulla voidaan analysoida testausprosessin kehittymistä sekä ohjelmistosuunnittelijoiden tuottamien virheiden määriä. (1, s. 10.)

Kun integraatiotestauksen onnistuneeseen suorittamiseen kuluva aika merkittävästi kasvaa, voidaan useimmissa tilanteissa todeta kehitettävän ohjelmiston tai osakokonaisuuden sisältävän enemmän virheitä ja siitä on tulossa haavoittuvaisempi. Jotta edellä mainitulta tilanteelta voidaan välttyä tai vähentää sen realisoitumista, voisi tiheämpää integraatiotestausta harkita. (1, s. 10.)

3 TESTAUSANALYYSI

Ohjelmiston testausanalyysiä voidaan suorittaa manuaalisesti sekä automatisoidusti. Testausanalyysi jaotellaan kahteen kategoriaan, joita ovat staattinen ja dynaaminen analyysi. (6, s. 13–14.)

Testausanalyysin osat jaottuvat siten, että ohjelmiston staattinen analyysi suoritetaan ohjelmiston koodauksen yhteydessä, kun taas dynaaminen analyysi suoritetaan yksikkö-, integraatio- ja järjestelmätestauksen yhteydessä (3, s. 210).

3.1 Staattinen analyysi

Staattinen analyysi on virheiden analysointia ilman ohjelman suorittamista. Analyysiä voidaan suorittaa manuaalisesti ja automatisoidusti analysointiin tarkoitetuilla ohjelmistoilla ja työkaluilla. Manuaalinen ohjelmakoodin katselmointi on myös osa staattista analyysia. (3, s. 210.)

Analyysin avulla tutkitaan ohjelman kompleksisuutta ja sillä voidaan havaita virheet jotka aiheutuvat muuttujien, osoittimien ja funktioiden väärinkäytöksistä. Staattisella analyysilla voidaan ohjelmistossa ilmeneviä vikoja vähentää merkittävästi ennen dynaamista testaamista. (3, s. 194.)

Ohjelmakäännöksen työkalut kykenevät laajasti havaitsemaan virheellisen syntaksin aiheuttamia ongelmia, joten tästä syystä sitä voidaan pitää myös yhtenä testausvaiheena. Ohjelmistolle voidaan suorittaa myös automatisoitua staattista analyysia siihen soveltuvilla työkaluilla. Staattisen analysoinnin työkalusta on kerrottu lisää luvussa 5.

3.2 Dynaaminen analyysi

Dynaamisessa analyysissa testattava ohjelma tai ohjelmanosa suoritetaan, jotta ohjelman ja funktioiden toimintaa voidaan testata. Analyysin suorittaminen edellyttää toimivaa ohjelmistoa ja toimintaympäristö. (3, s. 210.)

Ohjelmistolle määritetyt toiminnalliset ja ei-toiminnalliset vaatimukset ovat testattavissa dynaamisella analyysillä. Dynaamista analyysia voidaan suorittaa myös automatisoidusta, josta on kerrottu lisää luvussa 6.

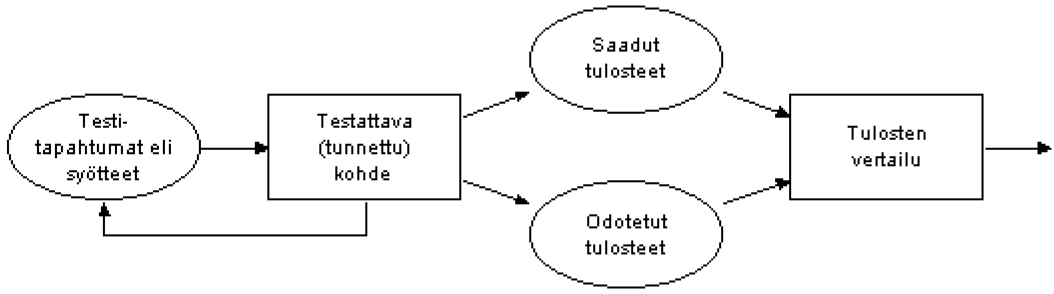
4 TESTAUSMENETELMÄT

Testausmenetelmän valinta on riippuvainen testitapauksen luonteesta ja testausvaiheesta. Perustestausmenetelmiä ovat mustalaatikkotestaus, harmaalaatikkotestaus ja valkolaatikkotestaus. Menetelmän valintaan voivat vaikuttaa myös testaajien taidot. (2, s. 35.)

Testausmenetelmät ovat sidonnaisia V-mallin mukaisiin testausvaiheisiin. Testausten siirtyessä seuraavaan vaiheeseen siirrytään myös valkolaatikkotestauksesta enemmän mustalaatikkotestaukseen. (2, s. 23.)

4.1 Valkolaatikkotestaus

Valkolaatikkotestauksessa (white-box testing) testin suorittaja on yleisesti ohjelmointisuunnittelija, joka on suunnitellut ja toteuttanut testattavan funktion tai moduulin toiminnon ja on tietoinen sen rakenteesta. Valkolaatikkotestauksen toimintaperiaate on esitetty alla (kuva 2). Moduulitestauksessa hyödynnetään valkolaatikkotestausmenetelmää. (3, s. 193.)



KUVA 2. Valkolaatikkotestaus (7)

4.2 Harmaalaatikkotestaus

Harmaalaatikkotestaus (gray-box testing) on mustalaatikko- ja valkolaatikkotestauksen välimuoto. Testin suunnittelijana on yleensä jokin muu, kuin moduulin tai funktion suunnitellut ohjelmistosuunnittelija. (3, s. 193.)

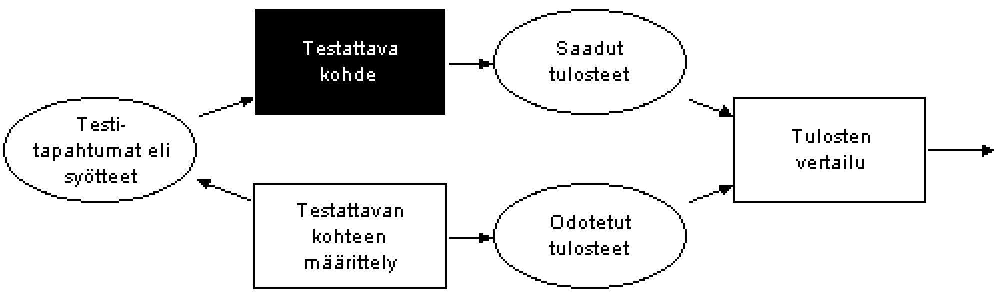
Integraatiotestauksen vaiheessa harmaalaatikkotestaaja on tietoinen ohjelmakoodin rakenteesta ja sisällöistä, mutta testaa moduulien rajapintojen raja-alueita. Testissä tyypillisesti testataan nollalla jakamista, lukujen pyöristämisiä sekä päivämäärien alku- ja loppuarvoja. (3, s. 193–194.)

4.3 Mustalaatikkotestaus

Mustalaatikkotestauksella (black-box testing), tarkoitetaan kokonaisen ohjelmiston järjestelmätestausta, jossa testaaja ei tiedä testattavan osan rakennetta tai koodia ja osan toiminta voi olla hämärän peitossa, lukuun ottamatta syötteitä ja testauksesta saatuja tuloksia. Järjestelmätestausvaiheessa mustalaatikkotestauksen suorittajalla on tiedossa ohjelmiston vaatimukset, joita vasten testejä suoritetaan. (3, s. 194.)

Mustalaatikkotestaus voi pitää sisällään myös stressitestausta, käytettävyystestausta ja hyväksymistestausta. Ohjelmiston loppukäyttäjä voi olla osallisena mustalaatikkotestauksessa. (3, s. 194.)

Mustalaatikkotestauksessa testitulosten oikeellisuus varmistetaan saatujen ja odotettujen tulosten vertailulla, kuten kuvassa 3 on esitettynä.



KUVA 3. Mustalaatikkotestaus (7)

5 TIETOTURVA

Nykypäivän ohjelmistot ja järjestelmät ovat alttiita monille tietoturvahaavoittuvuuksille ja ne voivat olla merkittävä riski yritystoiminnalle. Tietoturvasta huolehtiminen vähentää riskejä ja se on osa riskienhallintaa. (8, s. 8.)

Jotta tietoturvasuunnitelma voidaan laatia, tulee ohjelmiston tietoturvan vaatimusten olla tarkoin selvillä. Ohjelmistolle voi olla myös ulkoisia tietoturvavaatimuksia, esimerkiksi lainsäädännölliset vaatimukset, ja nämä tulisi huomioida jo ohjelmistokehitysprojektin vaatimusmäärittelyvaiheessa. (8, s. 8.)

5.1 Tietoturvasuunnitelma

Jokaisessa ohjelmistokehityksessä tulisi laatia tietoturvasuunnitelma, jossa huomioidaan kaikki tarpeellinen tietoturvaan liittyvä asia, joka koskee ohjelmiston lisäksi laitteistoinfrastruktuuria ja tietoliikennettä (8, s. 9).

Tietoturvasuunnitelma ottaa kantaa siihen, miten uhkiin varaudutaan ja miten järjestelmän tietoturvaa voidaan mitata. Järjestelmän turvallisuuden varmentaminen ei ole pelkästään ohjelmistokehityksen aikainen vaihe, jonka vuoksi ohjelmiston auditointitarpeet ja -kohteet huomioidaan suunnitelmassa. Huolellinen tietoturvasuunnittelu ja järjestelmän lokien kirjaus ja niille määritetty ohjeistus parantaa merkittävästi ohjelmiston ja järjestelmän turvallisuutta. (8, s. 10.)

5.2 Tietoturvatestaus

Tietoturvatestaus on erittäin tärkeä ohjelmistotestauksen osa-alue, jota ei tule jättää huomiotta ohjelmistokehityksessä. Tietoturvatestaus voi olla esimerkiksi ohjelmakoodin katselmointia, penetraatiotestausta sekä kriittisten ongelmatilanteiden pöytätestausta. Testauksen avulla voidaan löytää, varmentaa ja uudelleenarvioida tietoturvasuunnitelmassa huomioidut tietoturvariskit. (8, s. 17–18.)

Ohjelmakoodien katselmoinnilla sekä muilla manuaalisilla testausmenetelmillä voidaan parantaa ohjelmiston laatua, mutta siitä huolimatta tietoturvatestauksessa tulisi hyödyntää automatisoituja tietoturvatestaustyökaluja (9). Tietoturvatestauksen automatisoinnista on kerrottu tarkemmin luvussa 6.

Ohjelmiston tietoturvatestausta voidaan suorittaa staattisella ja dynaamisella testauksella samalla tavoin, kuten ohjelmiston muitakin ominaisuuksia ja toiminnallisuuksia (9).

Staattinen tietoturvatestaus

Staattisella tietoturvatestauksella tarkoitetaan ohjelman lähdekoodin tarkastelua tietoturvan näkökulmasta. Testaus suoritetaan ilman ohjelman suorittamista ja se on tärkeä vaihe haavoittuvuuksia havaitsemiseksi mahdollisimman aikaisessa vaiheessa. Manuaalista tietoturva-analysointia voidaan suorittaa ohjelmakoodin katselmoinnin yhteydessä. (9.)

Staattisella testauksella voidaan löytää muun muassa tietojen salaukseen liittyviä ongelmia, kuten heikkojen salausalgoritmien käyttöä tai salasanojen säilytyksestä koituvia haittoja (9). Ohjelmiston tietoturvan staattista testausta voidaan suorittaa myös automatisoidusti, josta lisää luvussa 6.3.

Dynaaminen tietoturvatestaus

Tietoturvan dynaaminen testaus tarkoittaa ohjelman haavoittuvuuksien ja riskien testaamista ohjelmiston suorittamisen yhteydessä. Testauksen suorittaja ei usein tiedä käytetyistä teknologioista, ohjelmistokehyksistä eikä lähdekoodista. (9).

Dynaaminen tietoturvatestaus ajoittuu usein ohjelmistokehityksen elinkaaren loppupäähän, jonka seurauksena dynaamisella testauksella havaitut haavoittuvuudet ja korjaaminen ovat kalliimpia sekä työläämpiä. Tyypillisesti dynaamisessa testausvaiheessa havaitut viat määritetään korjattavaksi vasta seuraavassa iteraatiossa. Poikkeuksena voidaan ottaa haavoittuvuudet, jotka ovat liiketoiminnalle tai ohjelmistolle kriittisiä, jolloin ne saatetaan korjata ohjelmiston hätäpaikkauksessa.

Dynaamisella testauksella voidaan havaita myös suorituksenaikaiset ja ympäristöriippuvaiset ongelmat. Tätä testaustapaa käytetään tyypillisesti verkkosovelluksille ja -palveluille, eikä se välttämättä ole niin hyödyllinen muille ohjelmistotyypeille. (9.) Ohjelmiston tietoturvan dynaamista testausta voidaan suorittaa myös automatisoidusti, josta lisää kappaleessa 6.3.

6 TESTAUKSEN AUTOMATISOINTI

Ohjelmistokehityksen elinkaaren aikana tulee tarve suorittaa laajoja ohjelmistotestauksia, jotta voidaan varmistaa ohjelmiston määritysten mukainen toiminta ja tietoturva. Testitapausten määrä on suuri ja ne vaativat testausten suorittajilta paljon aikaa. (1, s. 23.)

Testauksen automatisointia tärkeyttä mietittäessä voidaan tarkkailla seuraavan kaltaista tilannetta: Ohjelmiston testaaja havaitsee toiminnassa puutteita tai virheitä ja raportoi niistä ohjelmistosuunnittelijalle. Ohjelmistosuunnittelija paikantaa ja korjaa havaitut virheet ja puutteet, jonka jälkeen testaaja joutuu suorittamaan kaikki virhettä tai puutetta edeltäneet testitapaukset manuaalisesti uudelleen varmistaakseen, etteivät ohjelman lähdekoodiin tehdyt muutokset tuottaneet uusia virheitä aikaisempiin testitapauksiin. Tällä tavalla testaus kuluttaisi valtavasti testaajien aikaa ja vaatisi suuren määrän henkilöstöresursseja. Ohjelmiston testaus olisi täten suurelta osin erittäin vajavainen, jonka seurauksena ohjelmistot olisivat rikkinäisempiä ja haavoittuvaisempia. (1, s. 23–24.)

Ohjelmiston luotettavuutta ja laatua voidaan parantaa merkittävästi automaattisella ohjelmistotestauksella. Testitapaukset voidaan toistaa muuttumattomina aina tarpeen vaatiessa niin useasti kuin tarve vaatii. On olemassa myös testejä, joiden suorittamista manuaalisesti voidaan pitää käytännössä mahdottomina. (1, s. 23.)

6.1 Automatisoinnin suunnittelu

Automatisoinnin suunnittelussa tavoitteena on tarkastella jokainen ohjelmiston testitapaus yksittäin ja selvittää sen automatisoitavuus ja kannattavuus. Kaikkea testausta ei voida automatisoida, eikä automatisointi välttämättä ole kannattavaa. (1, s. 23.)

Suunnittelussa tulisi huomioida manuaalisen testauksen automatisointiin kuluva aika, joka on tyypillisesti 3–10 kertaa manuaalista toteutus suurempi (1, s. 7).

6.2 Testitapauksen automatisointi

Testitapausten automatisointi voidaan aloittaa hyvin varhaisessa vaiheessa ohjelmistokehitystä. Testitapausta tarkastelemalla voidaan tuottaa testauskoodi, joka simuloi todellisen käyttäjän tai moduulin osaa eri syötteillä. Testauskoodi on sarja eri komentoja ja ohjeita, joita testaustyökalulla suoritetaan. Automatisoidun testitapauksen avulla voidaan ohjelmistotestaajalta paljon aikaa vieneen testauksen suorittaa vain muutamissa minuuteissa. (1, s. 24–25.)

6.3 Testausjärjestelmän alusta

Versionhallintajärjestelmää voidaan pitää tyypillisenä testausjärjestelmän alustana, joka kykenee välittämään käännös- ja testaustyökaluille käskyn suorittaa testattavalle ohjelmalle testitapauksille ennalta määritetyt toimenpiteet. Versionhallintajärjestelmiin voidaan konfiguroida eri tapahtumia, joiden perusteella käskyt suoritetaan. Helpoiten ymmärrettävissä olevina esimerkkeinä voidaan pitää lähdekoodin tehdyn muutoksen jälkeinen testien ja käännösten suorittaminen sekä ajoitetut järjestelmätestaukset. (10, s. 14.)

Versionhallintajärjestelmällä yleisesti dokumentoidaan ja ylläpidetään ohjelmistoon tehtyjä muutoksia sekä ohjelman lähdekoodeja (10, s. 14).

6.4 Jatkuva integraatio

Jatkuva integraatio (continuous integration, CI) on kehitetty helpottamaan ohjelmistosuunnittelijoiden työtä sekä parantamaan ohjelmiston laatua. Se tarkoittaa päivittäistä ohjelmistomoduulien ja osakokonaisuuksien yhteensovittamista ja testausta, jossa integraatiotestaus suoritetaan välittömästi lähdekoodiin tehdyn muutoksen jälkeen. (10, s. 6.)

Jatkuvassa integraatiossa ohjelmisto käännetään aina uudelleen ja sille suoritetaan ennalta määritetyt moduuli- ja integraatiotestaukset. Jotta jatkuva integraatio toimii ohjelmistokehityksessä, on testitapaukset käytännössä automatisoitava. (10, s. 13.)

Automaattisella jatkuvan integroinnin testauksella voidaan havaita lähdekoodiin tuotetut virheet ja toiminnalliset puutteet manuaalisia testejä huomattavasti aikaisemmassa vaiheessa. Jatkuvan integroinnin avulla projektin loppuvaiheen viat voidaan minimoida, joka edesauttaa vähentämään projektiin kohdistuvia lisäkustannuksia. Jatkuvan integraation avulla voidaan lyhentää integrointiin kuluvaa aikaa sekä vähentää integraatio-ongelmia. (10, s. 13.)

6.5 Tietoturvatestaus

Testauksen automatisoinnin ja jatkuvan integraation käyttöönoton yhteydessä voidaan hyödyntää tietoturvatestaukseen soveltuvia työkaluja. Tietoturvan automaattiset testaustyökalut kykenevät havaitsemaan suuren osan ohjelmistossa ilmestyvistä tunnetuista vakavista haavoittuvuuksista. Tieturvatestauksen automatisoinnin avulla voidaan vähentää tietoturvaryhmän työmäärää, jolloin he kykenevät tutkimaan ja testaamaan monimutkaisempia tietoturvaongelmia ja haavoittuvuuksia. (9.)

Ohjelmiston tietoturvatestauksen tarpeellisuutta ei voida missään tilanteessa väheksyä, eikä testauksia voida tehdä ohjelmiston kannalta ajateltuna liikaa. Tietoturvatestauksen automatisoinnilla saavutetaan huomattavasti luotettavampi, vähemmän haavoittuvaisempi ja turvallisempi ohjelmisto. Tietoturvatestauksen automatisoinnin avulla tietoturvassa ilmenevät haavoittuvuudet voidaan havaita paremmin ja huomattavasti aikaisemmin, jolloin mahdolliset riskitekijät voidaan paremmin välttää. Havaittuihin tietoturvaongelmiin voidaan reagoida nopeammin ja paikkaukset ohjelman lähdekoodiin voidaan tehdä aikaisemmin sekä edullisemmin.

6.5.1 Automatisointi

Ohjelman automaattista tietoturvatestausta voidaan suorittaa staattisella sekä dynaamisella analysointityökaluilla. Useat CI-työkalut voivat sisällyttää automaattiset skannaustyökalut, jotka suoritetaan automaattisesti ohjelmistoon tehdyn lähdekoodimuutoksen yhteydessä sekä ajastetuissa tehtävissä. (11.)

Tietoturvatestauksen automatisoinnin yhteydessä tulee huomioida automatisoinnin ulkopuolelle jäävät testitapaukset, jotka vaativat manuaalista suorittamista.

Automaattiset tietoturvatyökalut eivät usein havaitse harvinaisia tai jopa hyvin yleisiäkään haavoittuvuuksia. Tietoturvauhkien havaitsemattomuus voi johtua joko työkalun vääränlaisesta konfiguraatiosta tai niiden luontaisista ominaisuuksista. Ohjelmiston autentikointi- ja valtuutusmenettelyt ovat aina erityishuomiota vaativia kohtia, jotka tulee automatisoinnista huolimatta analysoida manuaalisesti. (9.)

6.5.2 Staattinen analysointityökalu (SAST)

SAST-työkalulla voidaan analysoida ohjelman lähdekoodia, vaikka sovellus ei olisi suoritettavissa. Työkalulla voidaan löytää muun muassa salauksen ongelmia, kuten heikkojen algoritmien käyttöä tai salasanojen säilytystavoista koituvia haittoja. (9.)

SAST-työkalu soveltuu hyvin ketterien ohjelmistokehitysmenetelmien käyttöön ja sillä säästetään aikaa sekä nopeutetaan sovelluksen kehitystyötä ja käyttöönottoa. Työkalu voidaan sisällyttää CI/CD-prosesseihin, jolloin testaus voidaan suorittaa automaattisesti aina sovelluksen lähdekoodin muutoksen yhteydessä. (9.)

6.5.3 Dynaaminen analysointityökalu (DAST)

Dynaamisen analysointityökalun avulla voidaan havaita suorituksenaikaiset ja ympäristöriippuvaiset tietoturvaongelma. DAST-työkalut ovat yleisesti parempia löytämään tietyntyyppisiä haavoittuvuuksia kuin SAST-työkalut, esimerkiksi valtuutusongelmia. (9.)

DAST-työkalu voidaan liittää CI/CD-prosessien yhteydessä suoritettavaksi automaattisesti.

6.6 Yleiset ongelmat testauksen automatisoinnissa

Testauksen automatisoinnin ja työkalujen valinnassa saattaa syntyä mielikuva, että testitapausten automatisoinnilla ratkaistaan kaikki testauksen ongelmat. Automaattisen testausjärjestelmän heikkoutena voi olla myös kokematon testaaja sekä testitapausten huono tai puutteellinen suunnittelu ja dokumentointi. (1, s. 39.)

Huonosti suunniteltu automatisointi voi keskeyttää koko automatisointiprosessin. Tämän seurauksena automatisoidut testit voivat olla epätäydellisiä tai testit itsessään voivat sisältää virheitä. (1, s. 39.)

Oman haasteen testauksen automatisointiin ja sen kannattavuuteen tuovat ohjelmistoon ja sen vaatimuksiin ja sisältöihin tehtävät muutokset ohjelmistokehityksen eri vaiheissa. Jos testauksen automatisointi on suunniteltu huolella ja testausjärjestelmän valinnassa ja kehityksessä on onnistuttu, voidaan automatisoinnin kannattavuutta ja testauksen helppoutta parantaa merkittävästi. (1, s. 39–40.)

7 LÄÄKINNÄLLISTEN LAITTEIDEN TESTAUKSEN ERITYISVAATIMUKSET

Terveydenhuollon laitteiden ja ohjelmistojen kehitystyössä on huomioitava terveydenhuollon laitteista ja tarvikkeista asetettu laki (629/2010). EU:n direktiiveissä terveydenhuollon laitteista puhutaan termillä ”lääkinnällinen laite”. (12, s. 4.)

Laki määrittelee terveydenhuollon laitteeksi instrumentin, laitteiston, välineen, ohjelmiston, materiaalin tai muun yksinään tai yhdistelmänä käytettävän laitteen tai tarvikkeen, jonka valmistaja on tarkoittanut käytettäväksi ihmisen

- sairauden diagnosointiin, ehkäisyyn, tarkkailuun, hoitoon tai lievitykseen
- vamman tai vajavuuden diagnosointiin, tarkkailuun, hoitoon, lievitykseen tai kompensointiin
- anatomian tai fysiologisen toiminnon tutkimiseen, korvaamiseen tai muunteluun
- hedelmöittymisen säätelyyn (13, § 1:5).

Lääkinnällisen laitteen markkinoille saattaminen ja käyttöönotto vaativat Euroopan talousalueella laitteen lainmukaisten vaatimusten täyttymisen. Laitteessa tulee olla vähintään suomen-, ruotsin- ja englanninkieliset tiedot laitteistosta, ellei tietoja ole annettu hyvin tunnettujen ohje- ja varoitusmerkkien avulla. (13, § 3:12.)

Lääketieteellisen laitteen ja ohjelmiston tuotekehityksessä on syytä nojautua EN ISO 13485:2016 -standardiin. Standardi ei kata kaikkia laatujärjestelmään kohdistuvia vaatimuksia. (12, s. 12.)

7.1 Vaatimusten mukaisuuden osoitus

Tuotekehityksen tulosten tulee olla vertailtavissa lähtötietoihin ja niiden on vastattava lähtötiedoissa esitettyihin vaatimuksiin. Tuotekehitystyön eri vaiheet tulee olla määritettynä ja jokainen vaiheen tulee olla katselmoitu. Katselmointi täytyy olla suoritettu tarkkojen menettelyohjeiden mukaisesti. IVD-laitteiden kliinisessä tutkimuksessa tehdään myös suorituskyvyn arviointitutkimus. (12, s. 14.)

Tuotekehityksen lopputulos sisältää oston, tuotannon, laadunvarmistuksen, pakkaamon, kuljetuksen, varastoinnin, asennuksen ja huollon ohjeistukset tuotteen elinkaaren vaiheiden hoitamiseksi ja ne tulee olla dokumentoitu (12, s. 16).

7.2 Lääketieteellisen laitteen testaus

Ohjelmistokehityksessä on huolehdittava ohjelmistoon tehtyjen muutosten tarkka dokumentointi. Dokumenteista tulee selvitä muun muassa tehdyn muutoksen peruste, tekijä ja muutos.

Lääketieteellisen laitteen ja ohjelmiston testaus on välttämätön osa ohjelmistokehitystä. Testitapausten tulee olla suunniteltu kattavasti, testien tulee olla toistettavissa muuttumattomana ja testauksista tulee luoda tarkat dokumentaatiot.

Lääkinnällisen laitteen vaatimuksenmukaisuutta koskevien tietojen on oltava saatavilla vähintään viiden vuoden ajan valmistuksen päättymisestä (13, § 3:14).

8 TYÖKALUT

8.1 Gogs-versionhallinta

Gogs on avoimen lähdekoodin versionhallintajärjestelmä, joka on saatavilla MIT-lisenssillä. Gogs-ohjelmiston voi asentaa paikallisesti yrityksen palvelimille ilman veloitusta. Ohjelmisto on git-yhteensopiva, joten siirtyminen esimerkiksi github.com-ympäristöstä onnistuu vaivatta. (14.)

Gogs on ulkoasultaan ja toiminnallisuuksiltaan lähes GitHubin kaltainen. Järjestelmä on kirjoitettu Go-ohjelmointikielellä (14).

8.2 Jenkins-automatisointijärjestelmä

Jenkins on avoimen lähdekoodin automatisointijärjestelmä teknisten tehtävien toistuvaan suorittamiseen. Järjestelmä tukee versionhallintatyökaluja ja se kykenee suorittamaan shell-komentotulkin sekä Windowsin komentotulkin komentoja. (15.)

Jenkins voi suorittaa ohjelmakäännökset usean eri ulkoisen herätteen seurauksena, esimerkiksi versionhallintajärjestelmään tehdyn ohjelmakoodimuutoksen jälkeen. Järjestelmä kykenee suorittamaan myös ajoitettuja toimintoja sekä suorittamaan useita testauksia ja ohjelmajulkaisuja. Jenkins tarjoaa laajan kirjaston lisäosia ohjelman kääntäjistä julkaisutyökaluihin. Järjestelmän alusta on kirjoitettu Java-ohjelmointikielellä ja julkaistu MIT-lisenssillä. (15.)

8.3 GitLab CE

GitLab CE on avoimen lähdekoodin versionhallintajärjestelmä, jonka GitLab-yritys on kehittänyt. Ohjelmasta on olemassa myös maksullinen versio (GitLab EE), joka sisältää tiettyjä lisäominaisuuksia. (16.)

Gitlab CE sisältää muun muassa wiki-toiminnot, ongelmien seurantatyökalun ja CI/CD-työkalut. Ohjelmisto on kirjoitettu Ruby-ohjelmointikielellä ja julkaistu MIT-lisenssillä. Ohjelman asennus voidaan tehdä paikallisille palvelimille. (16.)

8.4 GitLab-Runner

GitLab-Runner on avoimen lähdekoodin projekti, jota käytetään ennalta määritettyjen tehtävien suorittamiseen ja tehtävien tulosten palauttamiseen GitLab-järjestelmään. Ohjelmaa käytetään GitLab CI:n kanssa yhdessä, joka koordinoi suoritettavia ohjelmia. (17.)

Ohjelma on asennettavissa GNU/Linux-, macOS-, FreeBSD- ja Windows-käyttöjärjestelmiin. Ohjelma on ilmainen ja julkaistu MIT-lisenssillä (17).

8.5 Docker CE

Docker CE -ohjelmisto on käyttöjärjestelmien ja ohjelmistojen virtualisointiin tarkoitettu ohjelmistokonttiympäristö, jolla voidaan kääntää, testata ja hallita suurta määrää sovelluksia aina kehityksestä tuotantoon saakka. Docker-alustan voi asentaa paikallisesti palvelimelle ja sitä voidaan käyttää myös pilvipalveluissa. Ohjelmistosta on saatavilla myös kaupallinen version (Docker EE). Docker CE on julkaistu Apache-lisenssillä. (18.)

Docker-levykuva (Docker-image) on levykuva käyttöjärjestelmästä, joka voi sisältää valmiiksi asennettuja ohjelmistoja ja sillä on omat ympäristömuuttujat. Docker-levykuvalle voidaan jakaa laitteiston resursseja, kansioita ja laitteistoja tarpeen mukaan. (18.)

Docker-kontit (Docker-container) ovat Docker-levykuvaa hyödyntäviä instansseja, joita voidaan luoda lähes rajaton määrä aina tarvittaessa. Kontit eivät oletuksena säilytä niille määritettyjä tiedostoja ja asetuksia tehtävien suorittamisen jälkeen. Kontille määritetyt tehtävät suoritetaan itsenäisinä prosesseina, eivätkä ne ole riippuvaisia muista konteista tai Docker-järjestelmää suorittavan käyttöjärjestelmän ympäristömuuttujista tai ohjelmistoista. (18.)

Konteista ei ole pääsyä Docker-alustaa suorittavaan käyttöjärjestelmään tai sen tiedostoihin ilman erillisiä konfiguraatioita. Kun Docker-kontin suorittamat ohjelmakoodien käännökset, lokitiedot tai asetukset halutaan ladata kontista muiden

ohjelmien käyttöön, voidaan Docker-konttiin liittää Docker-järjestelmää suorittavan tiedostojärjestelmän linkkejä, joita voidaan hyödyntää datan tallentamiseen pysyvästi. (18.)

Docker on tehokas työkalu ohjelmistokehityksen käytössä. Sillä voidaan varmistaa käännösten ja testien käyttöjärjestelmäriippumattomuus. Kontin sisällä ajettavat käännökset ja testit suoritetaan joka kerta samalla tavalla, eikä se ole riippuvainen Docker-alustan ympärillä olevasta käyttöjärjestelmästä, ellei sitä olla siten määritetty. Riippumattomuus alustan ympäristöstä mahdollistaa kontin siirtämisen ja kopioimisen usealle koneelle, jolloin käännökset ja testausympäristö ovat erittäin hyvin ja helposti skaalattavissa. (18.)

9 TESTAUSJÄRJESTELMÄN SUUNNITTELU JA TOTEUTUS

Opinnäytetyön käytännön osuutta ryhdyttiin toteuttamaan projektiluontoisesti yrityksen käytänteiden mukaisesti. Aiheen valinnan jälkeen pidettiin projektin aloituspalavaeri minun, ohjaavan opettajan sekä opinnäytetyöstä vastavan henkilön kanssa työpaikallani. Projektin aloituksessa tuotettiin lähtötietomuistio (liite 1). jonka perusteella opinnäytetyön toteutus aloitettiin.

Projektin suunnittelu aloitettiin määrittämällä projektin tavoitteet ja sisältö tarkemmin. Suunnitteluvaiheessa tuotettiin projektisuunnitelma, jossa otettiin kantaa myös projektin organisointiin, toteutussuunnitelmaan ja ohjaussuunnitelmaan. Dokumentissa määritettiin myös projektin päättäminen ja dokumentointisuunnitelma. Projektin määrittelyvaiheessa tuotettiin projektin kohteena olevan järjestelmän tekninen sekä toiminnallinen määrittelydokumentti, jotka sisälsivät kaiken tarpeellisen tiedon projektin toteuttamiseksi.

9.1 Järjestelmän suunnittelu

Automaattisen testausjärjestelmän suunnittelun lähtökohtana oli, että tuotettavan järjestelmän tuli kyetä suorittamaan tarvittavat ohjelmistokäännökset sekä moduuli- ja integraatiotestaukset yrityksen tuotekehityksen tarpeita mukaillen. Järjestelmän suunnittelussa ja hahmottamisessa hyödynnettiin UML-mallinnuskielen mukaisia kaavioita, joiden avulla testausjärjestelmän kokonaisuuden havainnointi oli helpompaa.

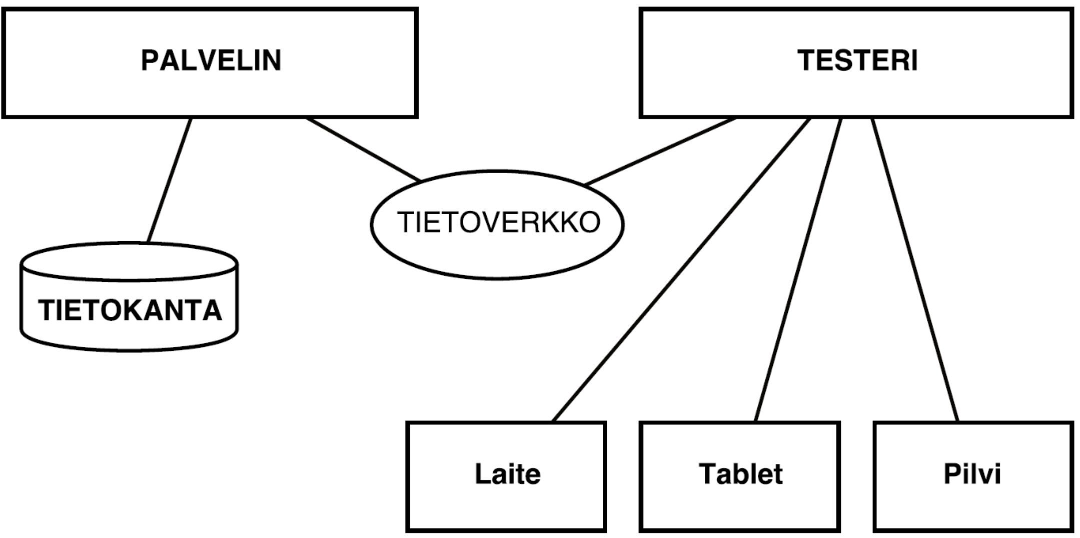
Suunnittelussa huomioitiin myös testausjärjestelmän jatkokehitettävyys projektin päättymisen jälkeen tuotannon testaukseen soveltuvaksi. Opinnäytetyön suunnitteluvaiheesta rajattiin pois järjestelmän järjestelmätestaus sekä järjestelmän jatkokehitys tuotannon testaukseen, koska ne eivät olleet yrityksen kannalta vielä ajankohtaisia vaiheita.

9.1.1 Järjestelmäkuvaus

Testausjärjestelmän kokonaisuutta hahmottaakseen järjestelmästä tuotettiin yleiskuvaus, jossa oli esitettynä testausjärjestelmään liittyvät osa-alueet mahdollisimman korkealla tasolla.

Testausjärjestelmän yleiskuvaus suunniteltiin siten, että se olisi mahdollisimman yksinkertainen ja helposti skaalattavissa. Järjestelmään suunniteltiin palvelin, joka sisältää testauksen tietokannat ja lähdekoodit. Järjestelmään suunniteltiin testeritietokone, jonka tehtävänä oli suorittaa ohjelmistokäännökset ja testitapausten suorittaminen.

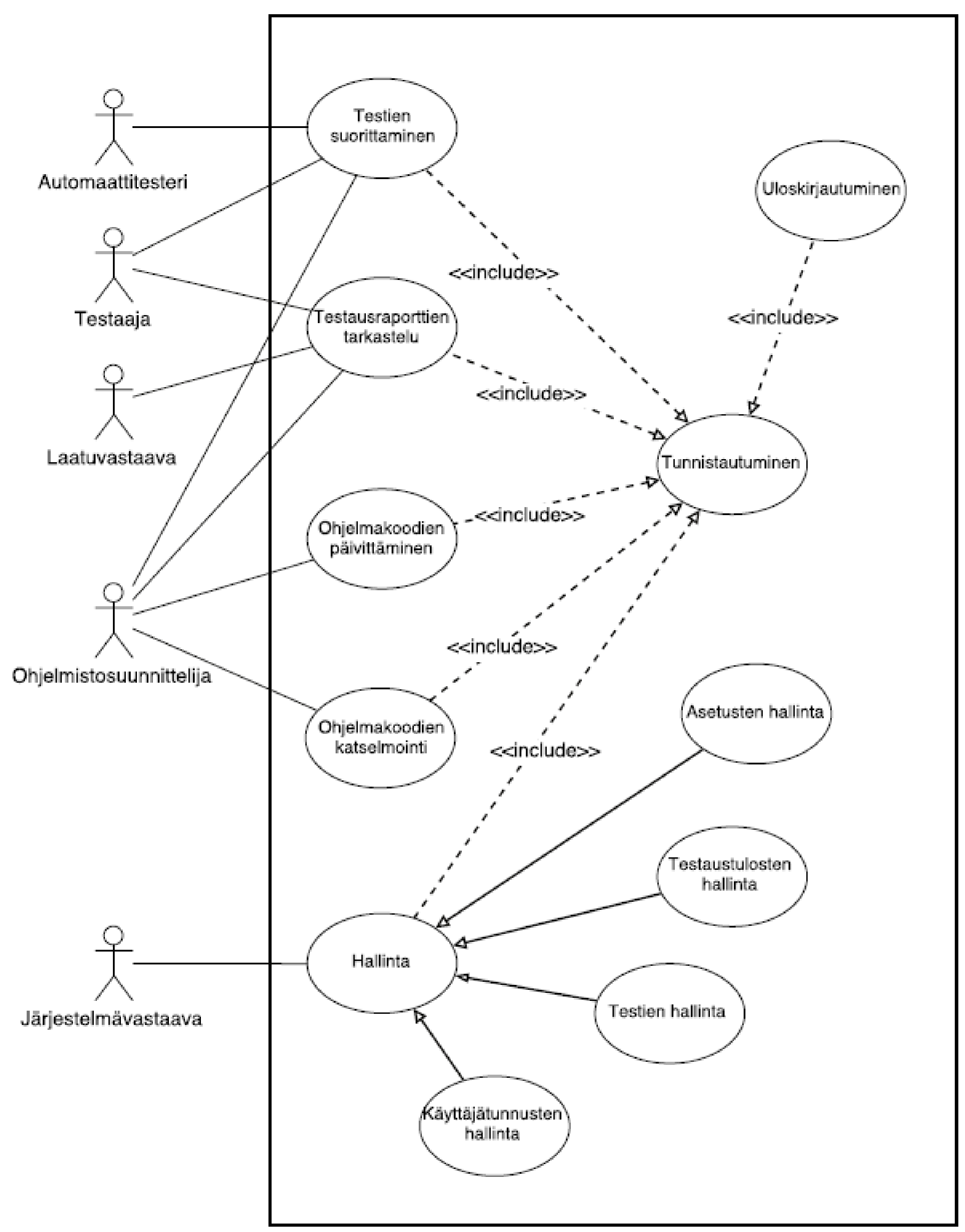
Automaattisen testausjärjestelmän järjestelmäkuvaus on kuvattuna UML-mallinnuskielen järjestelmäkaaviolla (kuva 4).



KUVA 4. Automaattisen testausjärjestelmän järjestelmäkaavio

9.1.2 Käyttötapausten suunnittelu

Kuten normaalissa ohjelmistokehitysprojektissa, myös testausjärjestelmän kehittämisessä oli tarpeen suunnitella järjestelmän eri käyttötapaukset. Käyttötapausten suunnittelussa määritettiin järjestelmän käyttäjät (actors) ja hahmoteltiin niiden tarpeet. Suunnittelutyön tuloksena tuotettiin UML-mallinnuskielen mukainen käyttötapauskaavio (kuva 5).



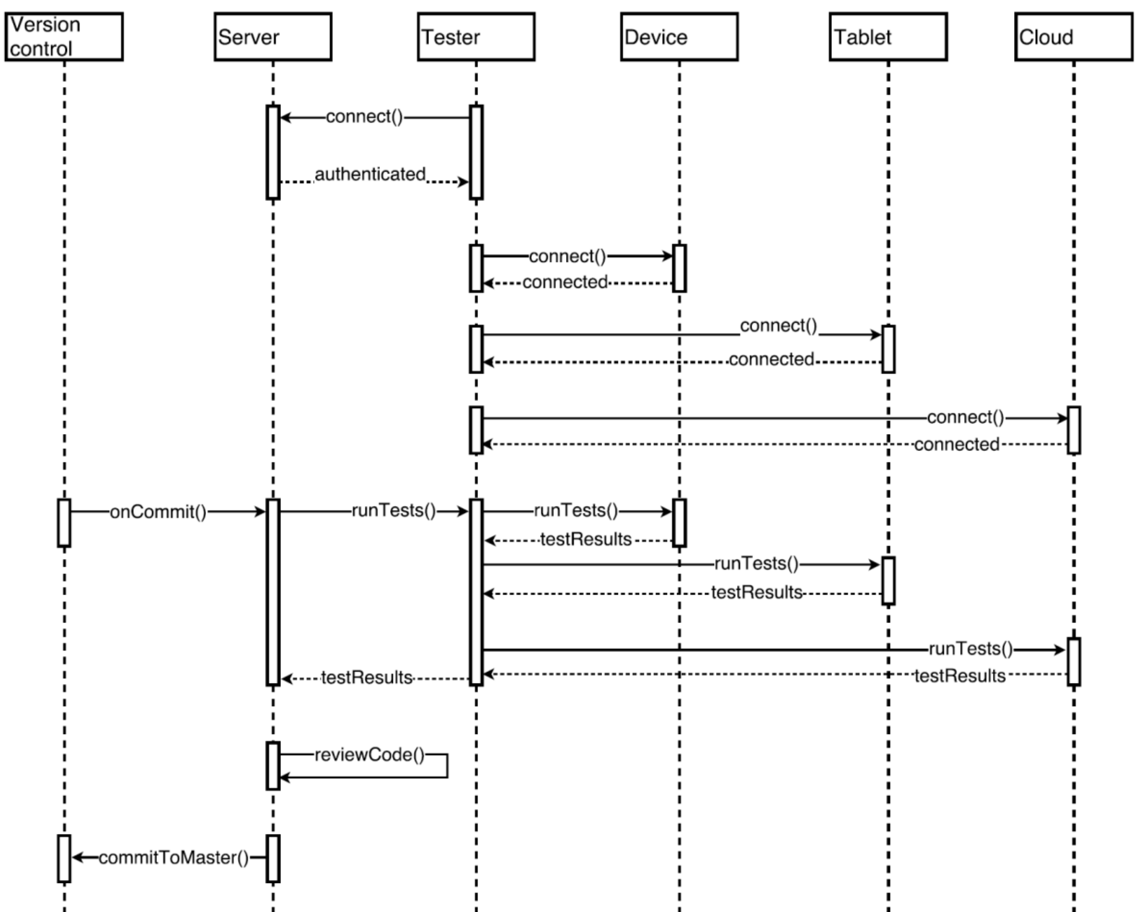
KUVA 5. Automaattisen testausjärjestelmän käyttötapauskaavio

Käyttötapauskaavion pohjalta laadittiin käyttötapauskortit, joiden avulla ohjelmiston testausta ja vaatimustenmukaista toimintaa voitiin testata. Käyttötapauskortit ovat salassa pidettäviä dokumentteja, jonka vuoksi niiden julkaisu on jätetty pois tästä opinnäytetyöstä.

9.1.3 Toiminnan suunnittelu

Projektin määrittelyvaiheessa suunniteltiin testausjärjestelmän toimintaa, jonka pohjana hyödynnettiin aikaisemmin tuotettua järjestelmäkaaviota (kuva 4). Toimintaa suunniteltaessa päädyttiin hyödyntämään versionhallintajärjestelmää testausjärjestelmän alustana.

Testausjärjestelmän suunnittelussa määritettiin järjestelmäosien välinen kommunikointi. Järjestelmän kommunikoinnissa suunniteltiin versionhallintajärjestelmän, palvelimen, testerilaitteen, diagnostiikkalaitteen, tablettitietokoneen ja pilvipalvelujen väliset viestiyhteydet. Suunnittelun tuloksena tuotettiin UML-mallinnuskielen mukainen viestiyhteyskaavio. (Kuva 6.)



KUVA 6. Automaattisen testausjärjestelmän viestiyhteyskaavio

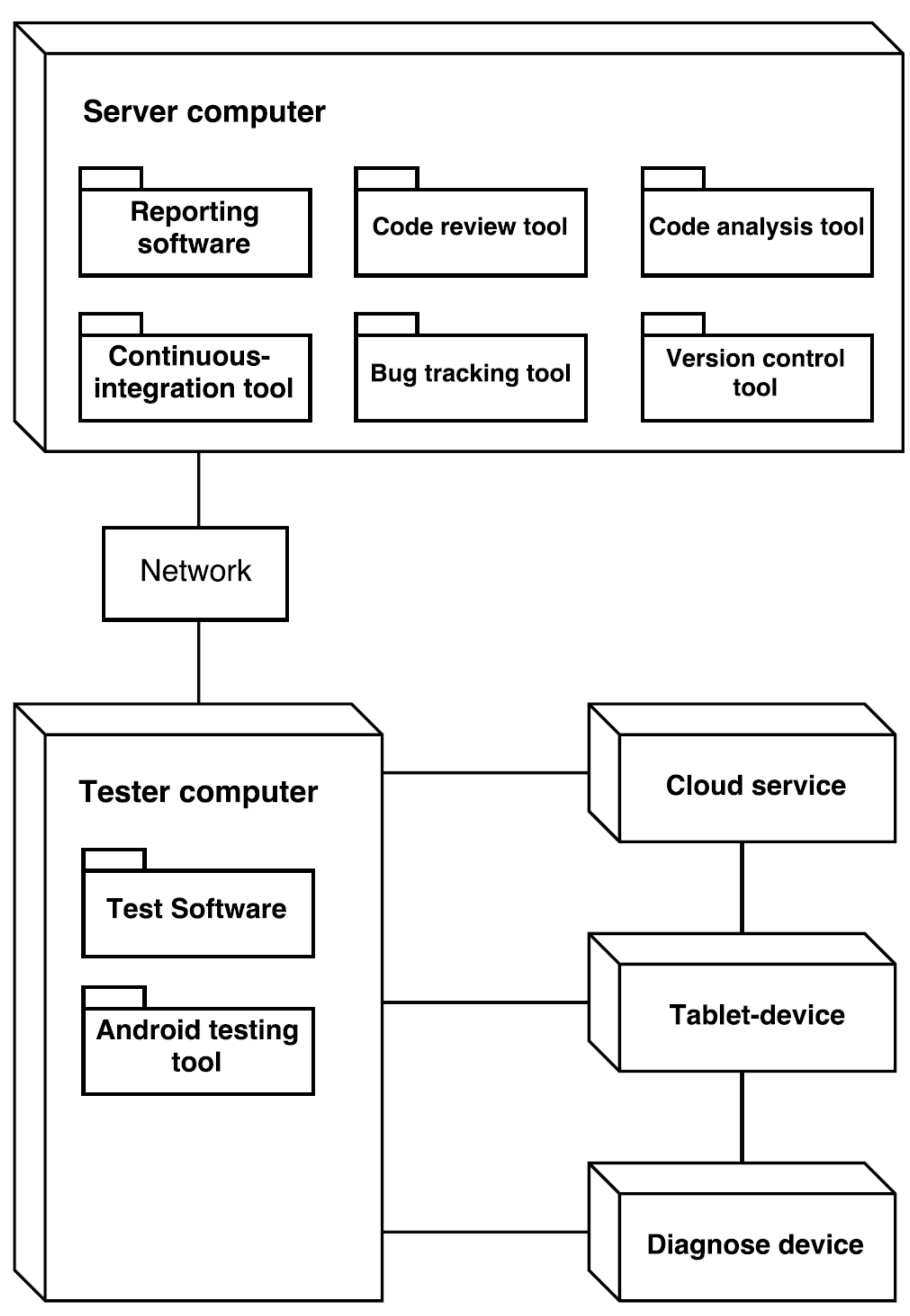
9.1.4 Järjestelmäarkkitehtuurin suunnittelu

Automaattisen testausjärjestelmän arkkitehtuurin suunnittelu aloitettiin määrittämällä ohjelmistokomponentit ja -työkalut, joita järjestelmän tulisi sisältää. Tässä vaiheessa versionhallintajärjestelmiä oli tutkittu riittävän syvällisesti, jotta voitiin todeta niiden käyttökelpoisuus testausjärjestelmän alustana.

Versionhallintajärjestelmiin on integroitu ominaisuudet, joilla voidaan välittää komentoja ja myös kuunnella muiden järjestelmien lähettämiä syötteitä ja käskyjä automatisoidusti ennalta määritettyjen toimenpiteiden mukaisesti.

Arkkitehtuurin suunnittelussa tutkittiin työkaluja, joita voitaisiin hyödyntää ohjelmistojen käännös- ja testaustoimenpiteisiin. Järjestelmään päätettiin lisätä versionhallintajärjestelmän lisäksi koodianalysointi-, koodinkatselmointi-, vikaraporttien hallinta-, raportointi- ja jatkuvan integraation työkalut.

Testeritietokoneen tuli voida suorittaa ohjelmistojen käännös- ja testaustoimenpiteet itsenäisesti. Testauksesta saadut tulokset tuli raportoida ja tallentaa testausjärjestelmään. Järjestelmän suunnittelun tuloksena tuotettiin UML-mallinnuskielen mukainen järjestelmäarkkitehtuurikaavio (kuva 7).



KUVA 7. Automaattisen testausjärjestelmän järjestelmäarkkitehtuuri

9.2 Työkalujen valinta

Järjestelmän suunnittelun jälkeen tutkittiin mahdollisia työkaluja, joita voisi hyödyntää järjestelmän toteutuksessa. Työkalujen valinnassa ehdoton kriteeri oli, että työkalut voitaisiin asentaa yrityksen paikallisille palvelintietokoneille.

Järjestelmän alustaksi soveltuvan versionhallintajärjestelmän valinnassa tutkittiin yrityksen käytössä olevaa Gogs-järjestelmää ja toisena vaihtoehtona GitLab CE-järjestelmää.

Lähdekoodiin tehtyjen muutosten katselmointiin tutkittiin entuudestaan tuttua Gerrit-työkalua, mutta sen käyttöönottoa ei voitu toteuttaa, koska se olisi vaatinut ulkoisen tunnistautumispalvelun. GitLab CE -järjestelmässä oli jo sisäänrakennettu katselmointityökalu, joten Gerrit-työkalun käyttöönotosta luovuttiin heti alkuvaiheessa.

Järjestelmän automatisointiin tutkittiin Jenkins-automatisointijärjestelmää. Tarkemman tutkimuksen perusteella voitiin todeta, että GitLab CE -järjestelmään sisäänrakennettu CI/CD-työkalun kykenee suorittamaan kaikki tarvittavat ohjelmistokäännökset sekä testauksen eri vaiheet. Tämän seurauksena Jenkins-järjestelmän käyttöönotosta voitiin luopua.

Järjestelmän vaatimuksena oli käännösten ja testausympäristön riippumattomuus testauslaitteistosta. Jotta tämä vaatimus voitiin toteuttaa, tutkittiin kokonaisen tietokoneympäristön virtualisointia sekä Docker-työkalua. Vaihtoehtojen tarkastelussa päädyttiin käyttämään Docker-työkalun tarjoamia palveluita sen monipuolisuuden, laajojen levykuvakirjastojen ja selkeiden ohjeiden vuoksi.

9.3 Järjestelmän toteutus

Järjestelmän toteutus aloitettiin asentamalla yrityksen käytössä olevalle Ubuntu-palvelimelle tarvittavat ohjelmistot ja työkalut. Ensimmäisenä tavoitteena oli saada siirrettyä Gogs-järjestelmään lisätyt kehitteillä olevien ohjelmistojen lähdekoodit GitLab CE -järjestelmään.

Palvelintietokoneelle asennettiin openssh-palvelin, cURL-ohjelmisto ja ca-sertifikaatit käyttämällä alla olevia komentoja.

```
$ sudo apt-get update && apt-get install -y curl openssh-server ca-certificates
```

9.3.1 GitLab CE

Testausjärjestelmän ensimmäisenä työkaluna asennettiin GitLab CE -versionhallintajärjestelmä. Asennus voitiin suorittaa bash-komentokehotteen kautta seuraavalla tavalla.

```
$ curl -sS https://packages.gitlab.com/install/repositories/gitlab/gitlab-
ce/script.deb.sh | sudo bash

$ sudo apt-get install gitlab-ce -y
```

Asentamisen jälkeen ohjelmisto konfiguroitiin testausjärjestelmän käyttöön soveltuvaksi muuttamalla ohjelman asetustiedostoa (/etc/gitlab/gitlab.rb). Tiedostoon määritettiin koneen IP-osoite ja käytettävä portti. Järjestelmän käytössä suositeltiin käytettäväksi unicorn daemon -työkalua, jotta GitLab CE -järjestelmä kykeni paremmin hyödyntämään palvelintietokoneen prosessoriytimiä.

```
external_url 'http://<IP.OSOITE>:<PORTTI>'

unicorn['port'] = 3001
```

Asetusten muutoksen jälkeen GitLab-järjestelmä käynnistettiin uudelleen suorittamalla uudelleenkonfigurointi alla olevalla komennolla.

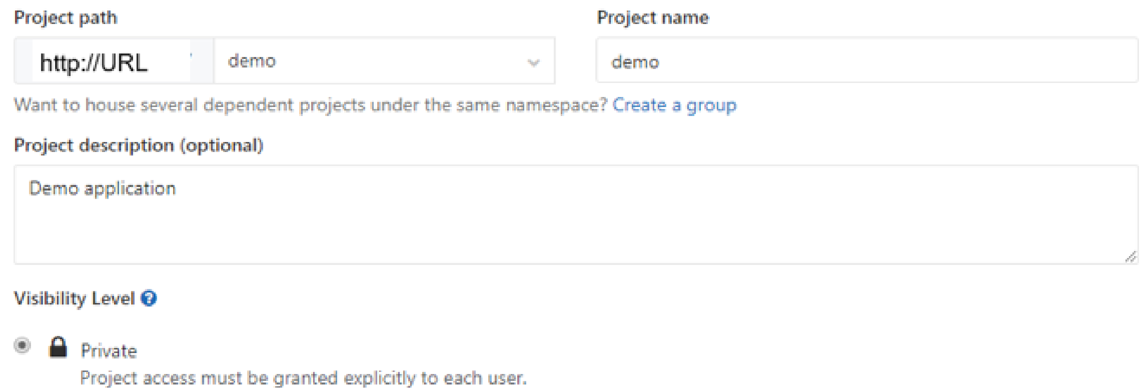
```
gitlab-ctl reconfigure
```

Järjestelmän asennuksen ja konfiguroinnin jälkeen järjestelmän viimeistely toteutettiin selaimen kautta. GitLab-järjestelmään asetettiin järjestelmävalvojan salasana ensimmäisellä käyttökerralla, jonka jälkeen järjestelmään kirjauduttiin järjestelmävalvojan tunnuksella ja salasanalla.

Lähdekoodien tietoturvan parantamiseksi järjestelmän asetuksiin tehtiin muutos, joka estää satunnaisia kirjautumissivuille tulleita käyttäjiä luomasta itselleen tunnuksia. Järjestelmäasetus (Sign-up enabled) asetettiin pois käytöstä.

9.3.2 Demo-ohjelma

Demo-ohjelman toteutuksen aluksi GitLab-järjestelmään lisättiin demo-niminen ryhmä, jonka alle luotiin uusi demo-niminen projekti. Projektin luonti tehtiin GitLab CE -järjestelmän käyttöliittymän kautta. (Kuva 8.)



KUVA 8. Demo-ohjelman projektin luonti GitLab CE -järjestelmään

Node.js-ohjelman luontia varten asennettiin sen vaatimat riippuvuudet alla olevan komennon mukaisesti.

```
$ apt-get install nodejs nodejs-legacy npm -y
```

Varsinainen demo-ohjelman konfigurointi suoritettiin hyödyntämällä npm-työkalua, joka kysyi ohjelmaan liittyviä perustietoja ja loi niistä asetustiedoston (package.json). Konfigurointi aloitettiin suorittamalla alla oleva komento.

```
$ npm init
```

Kun konfigurointi oli suoritettu, lisättiin ohjelman juurikansioon ohjelman aloitustiedosto (demo.js), joka suoritetaan ensimmäiseksi. Tiedostoon määritettiin tässä vaiheessa yksinkertainen komento, jonka tehtävänä oli tulostaa käyttäjälle teksti ”Application runned succesfully”. Ohjelman suorituskomento lisättiin tämän jälkeen asetustiedostoon seuraavalla tavalla.

```
"scripts": {
      "start": "node demo.js"
},
```

Tämän jälkeen ohjelman suorittamista voitiin testata seuraavalla komennolla.

```
$ npm start
```

Ohjelman suorittaminen tuotti alla olevan tulosteen konsoliin.

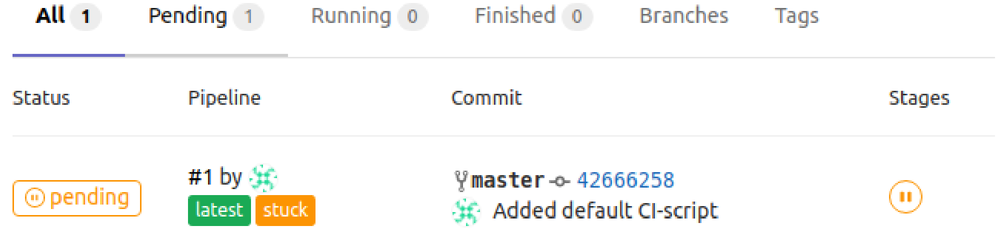
```
> demo@1.0.0. start ~/demo
> node demo.js
Application runned succesfully
```

Kun ohjelman suoritettavuus oli saatu testattua, lisättiin ohjelman lähdekoodit versionhallintajärjestelmään käyttämällä normaali git-komentoja.

9.3.3 GitLab CI -työkalun käyttöönotto

GitLab CI -työkalu voitiin ottaa käyttöön luomalla ohjelman juurikansioon tiedosto (.gitlab-ci.yml, liite 2). Tiedosto on ohje CI-työkalulle, jonne määritetään työjonon tehtävät ja missä järjestyksessä tehtävät suoritetaan.

Tiedoston luonnin jälkeen projektiin tehdyt muutokset päivitettiin versionhallintajärjestelmään. CI-työkalun toimivuutta tarkasteltiin GitLab CE -järjestelmän käyttöliittymän työjonolistan (CI/CD – Pipeline) kautta (kuva 9).



KUVA 9. CI-työkalun työjonoon lisätty tehtävä

Työjonon tarkemman tarkastelun yhteydessä huomattiin, että projektille ei ollut määritelty vielä CI-tehtävien suorittajaa (GitLab-Runner). Tehtävien suorittajatyökalun asennuksesta on kerrottu seuraavassa luvussa.

9.3.4 GitLab-Runner

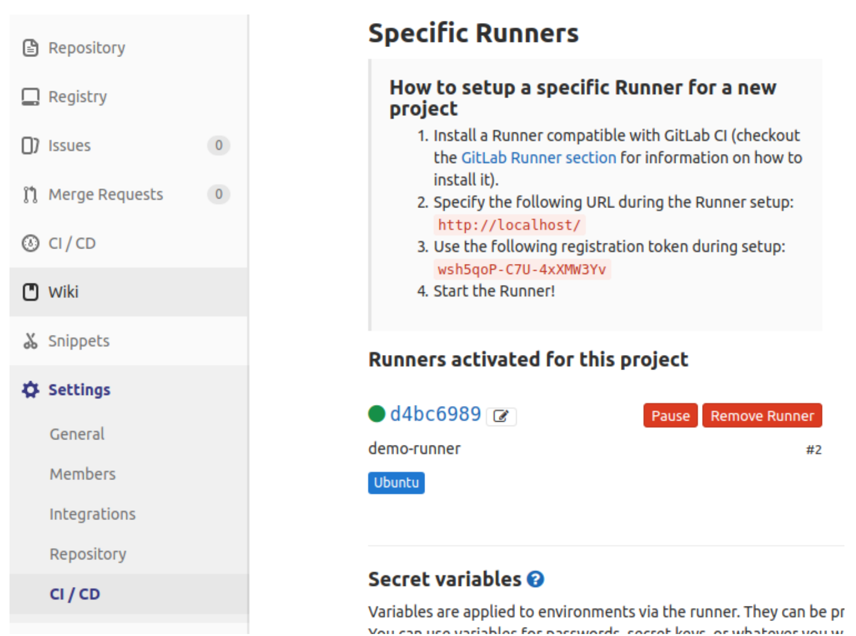
GitLab CI -työkalulle määritettyjä tehtäviä ei voida suorittaa ilman GitLab-Runner-työkalua. Työkalu asennettiin testaustietokoneelle, jotta se pystyi suorittamaan tarvittavat käännös- ja testaustoimenpiteet. Asennus suoritettiin alla olevan esimerkin mukaisesti.

```
$ curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | sudo bash

$ apt-get install gitlab-runner
```

GitLab-Runner-työkalu konfigurointi suoritettiin seuraavalla tavalla.

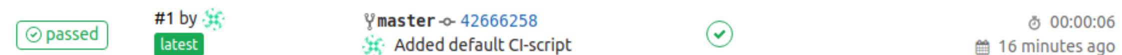
```
$ gitlab-runner register

>> http://<IP.OSOITE>:<PORT>          # CI/CD-työkalun osoite

>> <gitlab-ci token>                  # Haettu GitLab-käyttöliittymästä

>> demo-runner                        # Runner description

>> Ubuntu                             # Runner tags (comma separated)

>> true                               # Run untagged builds [true/false]

>> false                              # Lock runner to current project [true/false]

>> shell                              # Code executor [shell, ssh, virtualbox,
                                      docker+machine, docker-ssh+machine, docker-
                                      ssh, paralleles, kubernetes, docker]
```

Asetuksissa pyydetty varmenneavain voitiin hakea GitLab CE -järjestelmän käyttöliittymän kautta (Settings – CI/CD – Runners settings), jonka kautta myös linkityksen onnistuminen voitiin varmistaa (kuva 10).



KUVA 10. GitLab-Runner-työkalun varmenne ja onnistunut linkitys

Onnistuneen rekisteröinnin jälkeen aikaisemmin versionhallintajärjestelmään lisätty ohjelman automaattinen suorittaminen voitiin tarkistaa työjonojen osiosta, josta voitiin huomata, että ohjelmakoodi oli suoritettu onnistuneesti (kuva 11).



KUVA 11. Node.js-ohjelma suoritettu automaattisesti ilman ongelmia

9.3.5 Docker CE

Automaattinen testausjärjestelmän perustoiminto oli voitu todeta toimivaksi. Koska käännökset ja testaustoimenpiteet tuli voida suorittaa aina muuttumattomasti, suoritettiin manuaalinen testaus, jossa testaustietokoneelta poistettiin demo-ohjelman suorittamista varten tarvittavat nodejs-, nodejs-legacy- ja npm-paketit, jotta voitiin varmistua testausjärjestelmän toimivuus myös tällaisissa poikkeustilanteissa. Poiston jälkeen GitLab CE -järjestelmän työjonosivulta luotiin manuaalisesti uusi työjono, jota testaustietokoneen GitLab-Runner-työkalu ryhtyi suorittamaan.

Työkalujen poistaminen aiheutti testausjärjestelmässä odotettavissa olevan ongelman, jossa CI-työkalu ei kyennyt suorittamaan määriteltyjä toimenpiteitä onnistuneesti (kuva 12).

```
$ npm install
bash: line 50: npm: command not found
ERROR: Job failed: exit status 1
```

KUVA 12. Manuaalisesti määritetyn työjonon tuloste suoritusohjelmien poiston jälkeen

Docker CE -työkalun asennus

Testausjärjestelmään lisättiin Docker CE -työkalu, jotta vastaavilta ongelmatilanteilta voitiin välttyä. Työkalu kykeni luomaan eristetyn Docker-levykuvan, joka sisälsi tarvittavat ohjelmistot ja työkalut mitä tehtävän suorittaminen vaati. Työkalu asennettiin testauskoneelle seuraavalla tavalla.

```
$ sudo apt-get install apt-transport-https ca-certificates curl software-properties-
common

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -

$ add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
$(lsb_release -cs) stable"

$ apt-get update && apt-get install docker-ce -y
```

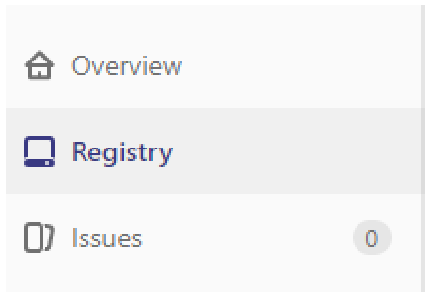
Docker-levykuvien rekisteri

Docker-levykuvien tehokasta hyödyntämistä ja projektikohtaista toimintaa varten GitLab CE -järjestelmään asetettiin käyttöön Docker-levykuvien rekisteri. Järjestelmässä rekisterit ovat projektikohtaisia, ja niissä voidaan säilyttää levykuvien eri versioita.

Käyttöönotto suoritettiin lisäämällä alla olevan mukainen asetus GitLab CE -järjestelmän asetustiedostoon, jonka jälkeen versionhallintajärjestelmälle suoritettiin uudelleenkonfigurointikäsky.

```
registry_external_url 'http://<IP.OSOITE>:3002'
```

Docker-levykuvien rekisterin käyttöönoton onnistuminen tarkistettiin GitLab CE -järjestelmän käyttöliittymästä projektin sivuilta (kuva 13).



KUVA 13. GitLab CE:n sisäänrakennettu Docker-levykuvien rekisteri

Jotta GitLab CE -järjestelmä kykeni hyödyntämään Docker-työkalua, lisättiin ohjelmalle asetustiedosto (/etc/docker/daemon.json). Tiedostoon määritettiin alla oleva sisältö, jotta Docker hyväksyi suojaamattoman http-yhteyden rekisteriin.

```
{ "insecure-registries": ["<IP.OSOITE>:3002"] }
```

Docker-levykuvan konfigurointi

Demo-ohjelman suorittamista ja testausta varten luotiin Docker-levykuva. Levykuvan määritystiedostoon (Dockerfile) määritettiin suoritettavat ohjelmakäskyt, jotta luotua levykuvaa voitiin hyödyntää ohjelman tarpeisiin. Tiedoston kuvaus liitteenä 3.

Levykuvan määritystiedoston syntaksin oikeellisuutta voitiin testata manuaalisesti ja luotu levykuva voitiin tallentaa projektin rekisteriin alla olevan komentosarjan avulla.

```
$ docker login <IP.OSOITE>:<PORTTI>

$ docker build -t <IP.OSOITE>:<PORTTI>/demo/demo:latest .

$ docker push <IP.OSOITE>:<PORTTI>/demo/demo
```

Luotua levykuvaa voitiin hyödyntää automaattisessa ohjelmatestauksessa muokkaamalla aikaisemmin määritettyä CI-ohjetiedostoa (.gitlab-ci.yml, liite 2) seuraavalla tavalla.

```
# image: node:6

image: <IP.OSOITE>:<PORTTI>/demo/demo:latest
```

Tehty muutos päivitettiin versionhallintajärjestelmään, josta CI-työkalu suoritti muuttuneiden suoritusohjeiden mukaisesti tehtävät onnistuneesti (kuva 14).

```
Running with gitlab-runner 10.4.0 (857480b6)
  on DemoRunner (7d10d54f)
Using Docker executor with image 10.0.2.6:3002/demo/demo:latest ...
Using docker image sha256:0c3214c6517167e526e8230998cf0ddc54c5989afbf780bc7453a33014cc7ece for predefined containe
r...
Pulling docker image 10.0.2.6:3002/demo/demo:latest ...
Using docker image 10.0.2.6:3002/demo/demo:latest ID=sha256:4d1f31e0842a4af3e4e21b6dac65f17341800c6e9262b5d929ea37f
48436fbd6 for build container...
Running on runner-7d10d54f-project-1-concurrent-0 via esa-VirtualBox...
Cloning repository...
Cloning into '/builds/demo/demo'...
Checking out 66c67d3a as master...
Skipping Git submodules setup
Checking cache for default...
Successfully extracted cache
$ npm install
npm WARN demo@1.0.0 No description
$ npm start

> demo@1.0.0 start /builds/demo/demo
> node demo.js

Application runned succesfully
Creating cache default...
node_modules/: found 1 matching files
Created cache
Job succeeded
```

KUVA 14. Demo-ohjelman automaattinen suorittaminen CI-työkalulla käyttäen kustomoitua Docker-levykuvaa.

Docker-levykuvan automatisointi

Ohjelmistokehitysten manuaalisia toimintoja vähentääkseen lisäämällä versionhallintajärjestelmään projektille uusi kehityshaara (docker_image), jota voitiin hyödyntää Docker-levykuvan luomisen automatisoinnissa.

Docker-levykuvan automatisointia varten kehityshaaran juurikansioon lisättiin CI-työkalun määritystiedosto (liite 4). Lisätty tiedosto tallennettiin versionhallintajärjestelmään, jonka jälkeen CI-työkalu suoritti Docker-levykuvan luonnin ja tallennuksen onnistuneesti (kuva 15).

```
Experimental: false
Insecure Registries:
 10.0.2.6:3002
 localhost:3002
 127.0.0.0/8
Live Restore Enabled: false

WARNING: No swap limit support
$ docker build -t $URL/$GROUP/$PROJECT:latest .
Sending build context to Docker daemon  4.096kB

Step 1/2 : FROM node:6
 ---> ecc41799ce17
Step 2/2 : RUN apt-get update && apt-get install -y     nodejs     nodejs-legacy     npm
 ---> Using cache
 ---> 4d1f31e0842a
Successfully built 4d1f31e0842a
Successfully tagged 10.0.2.6:3002/demo/demo:latest
$ docker push $URL/$GROUP/$PROJECT
The push refers to repository [10.0.2.6:3002/demo/demo]
4d6d87e72a4a: Preparing
29106d7f82b7: Preparing
31775b9047a3: Preparing
50599c766115: Preparing
d4141af68ac4: Preparing
8fe6d5dcea45: Preparing
06b8d020c11b: Preparing
b9914afd042f: Preparing
4bcdffd70da2: Preparing
8fe6d5dcea45: Waiting
06b8d020c11b: Waiting
b9914afd042f: Waiting
4bcdffd70da2: Waiting
31775b9047a3: Layer already exists
d4141af68ac4: Layer already exists
4d6d87e72a4a: Layer already exists
50599c766115: Layer already exists
29106d7f82b7: Layer already exists
8fe6d5dcea45: Layer already exists
06b8d020c11b: Layer already exists
b9914afd042f: Layer already exists
4bcdffd70da2: Layer already exists
latest: digest: sha256:d50d3485ab02cbfe4a30780ecf23cf9993403ab042ee7ba6da6998b5159ff55e size: 2219
Job succeeded
```

KUVA 15. Docker-levykuvan automatisoidun rakentamisen ja tallentamisen onnistunut toteutus CI-työkalulla.

9.4 Automatisoinnin toteutus

Opinnäytetyön automatisoinnin toteutus rajoittuu lähdekoodien salassapidon vuoksi vain demo-ohjelmien (Node.js ja Android) testaustoimenpiteiden määritykseen ja tuloksien tarkasteluun.

Automatisoinnin toteutuksessa esitetään malliesimerkit Node.js- ja Android-ohjelmien käännösten ja testausten automatisointiin.

9.4.1 Node.js

Node.js ohjelman pohjana hyödynnettiin aikaisemmin luotua demo-ohjelmaa. Ohjelman testausta varten asennettiin Mocha- ja Chai-moduulit, joita käytettiin ohjelman testauksessa. Asennus suoritettiin käyttämällä npm-työkalua.

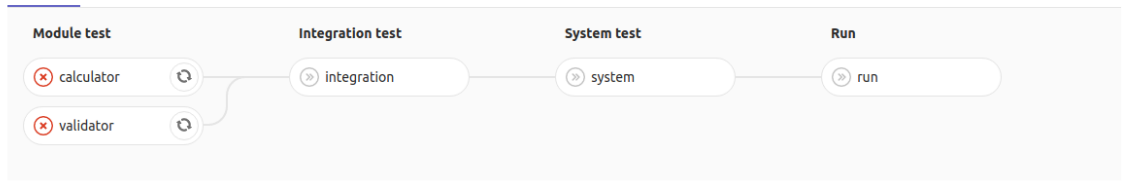
```
$ npm i mocha chai --save
```

Node.js-ohjelmaan lisättiin moduuli- ja integrointitestauksen koodit (liite 5). Ohjelmaan ei luotu tarkoituksenmukaisesti Calculator- ja Validator-moduuleita, jotta testausjärjestelmän virheen havaitsemista voitiin kokeilla. Tämän jälkeen testikäsky lisättiin ohjelman asetustiedostoon (package.json) seuraavalla tavalla.

```
"scripts": { "test": "mocha" }
```

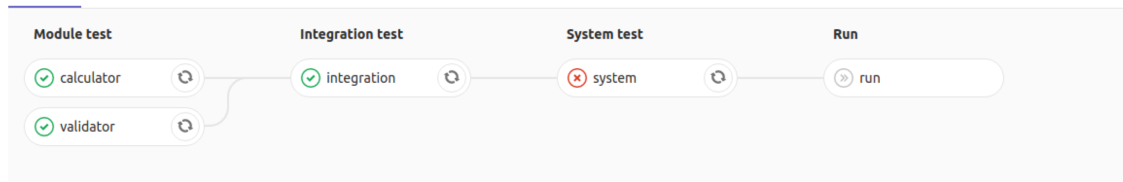
Testien suorittamista varten muokattiin aikaisemmin luotua GitLab CI-työkalun määritystiedostoa (liite 6), jotta halutut testaustoimenpiteet voitiin suorittaa testijärjestelmällä.

GitLab CI -työkalun suorittamat testit voitiin todeta kuvan 16 perusteella epäonnistuneen, koska testattavia moduuleita ei ollut vielä määritetty. Kuvasta voidaan havaita, että moduulitestauksen epäonnistuessa työkalu ei suorita seuraavia testausvaiheita, jos edellinen vaihe epäonnistuu.



KUVA 16. Tarkoituksellisesti aiheutettu moduulien vikatilanne

Kun testaus oli suoritettu, lisättiin ohjelmaan tarvittavat moduulit ja testattavat metodit. Moduulit lisättiin versionhallintajärjestelmään ja testituloksia tarkasteltiin uudelleen GitLab CE -järjestelmän työjonon kautta, josta voitiin todeta moduuli- ja integraatiotestien onnistunut suorittaminen (kuva 17).



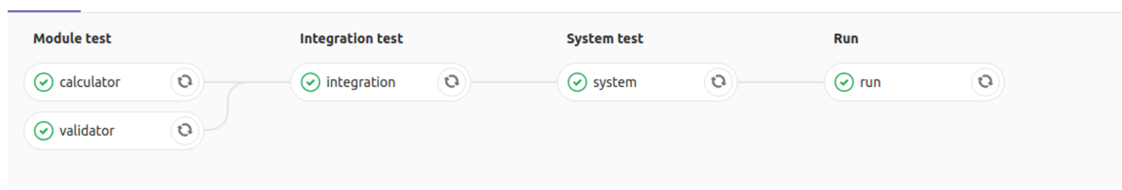
KUVA 17. Node.js-ohjelman automaattisesti suoritetut testit moduulien lisäyksen jälkeen

Koska testauksen tuloksena järjestelmätestauksen vaihe epäonnistui, tarkasteltiin testausraporttia tarkemmin. Raportista voitiin havaita syy, että ohjelman moduulien linkityksessä on virhe, eikä se voinut löytää lisättyä moduulia. (Kuva 18.)

```
1) Demo Node.js
     Run application without syntax error
       Should complete without issue:
   Error: Cannot find module './calculator'
    at require (internal/module.js:20:19)
    at Object.<anonymous> (lib/index.js:6:22)
    at require (internal/module.js:20:19)
    at Object.<anonymous> (index.js:3:28)
    at require (internal/module.js:20:19)
    at Context.it (test/system.js:11:4)
```

KUVA 18. Node.js-ohjelman järjestelmätestauksesta aiheutunut virheilmoitus automaattisen testausjärjestelmän tuottamana

Virheenkorjauksen jälkeen ohjelmistolle suoritettiin kaikki testitapaukset uudelleen, ja tuloksia tarkastellessa voitiin havaita, että kaikki testit suoritettiin onnistuneesti (kuva 19).



KUVA 19. Node.js-ohjelman testaukset suoritettu onnistuneesti GitLab-CI työkalun automatisoimana

9.4.2 Android-sovelluksen testaus

Kuten Node.js-ohjelman testauksessa, Android-sovellukselle luotiin versionhallintajärjestelmään tarvittava projekti. Projektiin lisättiin yksinkertainen sovellus (liite 10). Sovelluksen suorittamiseen CI-työkalun ohjetiedosto (liite 11).

Sovelluksen käännös- ja testaustoimenpiteitä varten luotiin uusi kehityshaara. Kehityshaaraa hyödynnettiin Docker-levykuvan luonnissa (liitteet 12 ja 13).

Esittelen tässä kohdassa vain ohjelman käännöksen suorittamisen, moduulitestien määrittelyn ja instrumentoidun integraatiotestauksen vaiheet.

Android-ohjelmaan lisättiin Node.js-ohjelman käyttämät moduulit. Moduulien testaukseen luotiin testauskoodit (liite 8), joilla moduulien toimintaa voitiin testata yksinkertaisilla vertailutoimilla.

Tarkoituksella viallinen moduuli

Testausjärjestelmän virheen havaitsemista voitiin kokeilla lisäämällä Calculator-moduuliin tarkoituksellisesti tuotettu virhe. Viallinen metodi näkyy alla olevassa koodissa.

```
public int divide( int val1, int val2 ) {

      return 0;

}
```

Moduulitestauksen tuloksena saatiin automaattisesti tuotettu testiraportti, josta voitiin tarkemmin tarkastella moduulitestausvaiheen testien suorittamista. Android-sovelluksen testauksesta saatiin automaattisen testausjärjestelmän lisäksi kääntäjän tuottama testiraportti (kuva 20).

Test	Duration	Result
testAdd	0s	passed
testDivide	0.005s	failed
testSubtract	0s	passed

KUVA 20. Viallisen moduulin kääntäjän tuottama testiraportin osa

Virheen havaitsemisen jälkeen Calculator-moduulin divide-metodi korjattiin ja muutokset tallennettiin versionhallintajärjestelmään. CI-työkalun tuottaman testauksen jälkeen voitiin todeta lähdekoodimuutoksen onnistuneen, eivätkä testaustoimenpiteet enää havainneet vikaa.

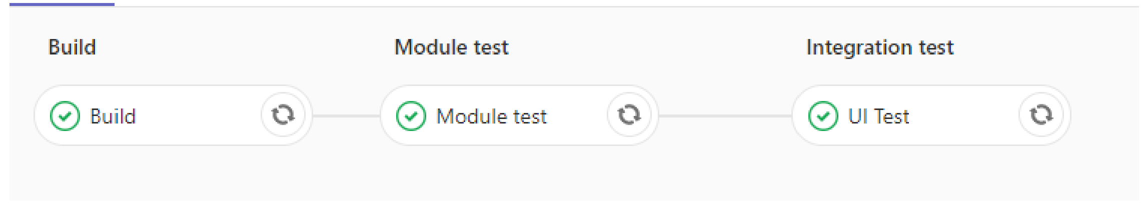
Android-sovelluksen instrumentoitu testaus

Android-sovellukseen lisättiin instrumentoituja integrointitestauksia, joita voitiin suorittaa testaustietokoneeseen kytketyn Android-laitteen avulla automatisoidusti. Testien lähdekoodit ovat liitteenä 9.

Tässä vaiheessa havaittiin ongelma Docker-kontin kanssa, koska kontista oli tarve päästä hallitsemaan ympäröivään käyttöjärjestelmään kytkettyä USB-laitetta. Ongelma voitiin ratkaista lisäämällä GitLab-Runner-työkalun asetustiedostoon alla oleva asetus.

```
devices = ["/dev/bus/usb:/dev/bus/usb:rwm"]
```

Asetusten muutoksen jälkeen sovelluksen käännös- ja testaustoimenpiteet suoritettiin onnistuneesti testausjärjestelmällä (kuva 21).



KUVA 21. Android-sovelluksen käännös ja testaustoimet suoritettu onnistuneesti

9.4.3 Sulautetun laitteen testaus

Testausjärjestelmään suunniteltiin sulautetun laitteen testauksen automatisointi. Toteutuksen yhteydessä suoritettiin laitteen ohjelmakoodille käännökset testerikoneella sekä käännetyn ohjelmakoodin lataaminen laitteistolle onnistuneesti.

Laitteen salassapidon vuoksi sulautettua laitetta ei esitellä tämän tarkemmin, eikä myöskään siihen liittyviä käännös- ja ohjelmointitoimenpiteitä.

9.4.4 Toteutuksen tarkastelu

Toteutuksen loppuvaiheessa järjestelmän toiminnot ja soveltuvuus varmistettiin suorittamalla useita testitapauksia yhtäaikaisesti ja useasti. Järjestelmään määritetyt testit voitiin suorittaa ilman ongelmia, eivätkä ulkoisen järjestelmän muutokset aiheuttaneet testeissä poikkeuksia.

Järjestelmän toteutuksessa harkittiin käytettäväksi GitLab-versionhallintajärjestelmän kaupallista versiota sen laajempien sisäänrakennettujen työkalujen ja ulkoisten ohjelmistojen tuen vuoksi, mutta opinnäytetyön toteutuksen kannalta siitä ei ollut merkittävää etua. Versionhallintajärjestelmä tullaan kuitenkin vaihtamaan jatkoprojektin aikana kaupalliseen versioon.

Opinnäytetyössä kerrottiin tietoturvatestauksesta ja sen automatisoinnista, mutta käytettävissä olevan resurssin sekä opinnäytetyön laajuuden vuoksi käytännön toteutusta ei kuitenkaan tässä työssä käsitelty.

10 YHTEENVETO

Opinnäytetyön päätarkoituksena oli suunnitella ja tuottaa työnantajan tarpeisiin soveltuva automaattinen testausjärjestelmä yrityksen ohjelmistokehityksen käyttöön. Kehitettävän testausjärjestelmän tavoitteena oli suorittaa jatkuvaa integraatiotestausta kehitettäville ohjelmistoille.

Opinnäytetyön aihetta mietittäessä havaittiin työnantajalla olevan selkeä tarve automaattiselle testausjärjestelmälle. Testausjärjestelmästä tulisi olemaan konkreettista hyötyä ohjelmistokehityksessä. Aihe oli minulle mieluinen ja teoreettiselta puolelta entuudestaan vähemmän tuttu. Testausjärjestelmän suunnittelutyön ja toteutuksen sain tehdä kokonaan itse Juho-Pekka Syrjälän avustamana ja se oli minulle sopivan haastava.

Automaattisen testausjärjestelmän toteutus onnistui hyvin, eikä suurempia vastoinkäymisiä tai ongelmia kohdattu. Järjestelmä kykenee toteuttamaan kaikki sille asetetut vaatimukset kokonaisuudessaan, ja sitä voidaan jatkojalostaa tuotannon testauksen käyttöön myöhemmässä vaiheessa.

Järjestelmä tulee vaatimaan jatkuvaa ylläpitoa, jotta järjestelmän toimivuus ja siitä saatava hyöty voidaan varmistaa. Järjestelmä on hyvin laajennettavissa, ja sinne voidaan määrittää uusia testitapauksia kohtuullisen vähäisellä vaivalla annettujen malliesimerkkien mukaisesti.

Projektia tehdessä opin erittäin paljon uusia asioita ohjelmistotestauksesta, tietoturvatestauksesta sekä automatisoinnista. Opinnäytetyön aikana minulle tuli myös tutuksi entuudestaan tuntematon Docker-työkalu, jota hyödynnämme tulevaisuudessa yrityksen muissakin hankkeissa.

LÄHTEET

1. Pohjolainen, Pentti 2003. Ohjelmiston testauksen automatisointi. Pro gradu -tutkielma. Kuopio: Kuopion yliopisto, tietojenkäsittelytieteen laitos. Saatavissa: http://www.cs.uku.fi/tutkimus/Teho/PenttiPohjolainen_Gradu.pdf. Hakupäivä 30.11.2017.

2. Rauhala, Eija 2010. Ohjelmistotestauksen suunnittelu - Case: A-lehdet Oy:n laskujen tulostusohjelma. Opinnäytetyö. Haaga-Helia, tietojenkäsittelyn koulutusohjelma. Saatavissa: https://www.theseus.fi/bitstream/handle/10024/23687/Rauhala_Eija.pdf?sequence=1&isAllowed=y. Hakupäivä 21.1.2018.

3. Dooley, John 2011. Software Development and Professional Practice. E-Kirja. apress. Saatavilla: https://www.apress.com/gp/book/9781430238027#otherversion=9781430238010. Hakupäivä 26.1.2018.

4. Lehtinen, Jesper 2016. Automated GUI Testing of Game Development Tools. Opinnäytetyö. Helsinki Metropolia University of Applied Sciences, information technology. Saatavissa: http://www.theseus.fi/bitstream/handle/10024/112962/Lehtinen_Jesper.pdf?sequence=1&isAllowed=y. Hakupäivä 25.1.2018.

5. Siitonen, Paula 2016. Ohjelmistotuotannon osa-alueet. PowerPoint esitys. Kaakkois-Suomen ammattikorkeakoulu. Saatavissa: http://cna.mamk.fi/public/Siitonen%20Paula/Ohjelmistosuunnittelu/Luennot/2.%20Ohjelmistotuotannon%20osa-alueet.ppt. Hakupäivä 25.1.2018.

6. Ahvamaa, Aleksi 2011. Käyttöliittymätason automaattinen testaus case: Enersize Oy. Opinnäytetyö. Satakunnan ammattikorkeakoulu, tietojenkäsittelyn koulutusohjelma. Saatavissa: https://www.theseus.fi/bitstream/handle/10024/34345/Ahvamaa_Aleksi.pdf?sequence=1&isAllowed=y. Hakupäivä: 25.1.2018.

7. Kautto, Tuomas 1996. Ohjelmistotestaus ja siinä käytettävät työkalut. Ohjelmistotekniikan seminaariesitelmä. Jyväskylä: Jyväskylän yliopisto, Informaatioteknologian tiedekunta. Saatavissa: http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/testaus/. Hakupäivä 26.1.2018.

8. Malmberg, Thomas 2014. Tietoturva ja IT-arkkitehtuuri. SlideShare. Mint Security. Saatavissa: https://www.slideshare.net/ThomasMalmberg/tietoturva-ja-itarkkitehtuuri. Hakupäivä 28.1.2018.

9. Kosten, Steve 2017. Are automated scans enough to detect all security problems in an application? Cypress Data Defence. Saatavissa: https://www.cypressdatadefense.com/app-security-tools/automated-scans-enough-detect-security-problems-application/. Hakupäivä 21.1.2018.

10. Vilkman, Vesa 2010. Jatkuva integraatio ohjelmistokehityksessä. Kanditutkielma. Jyväskylä: Jyväskylän yliopisto, tietojenkäsittelytieteen laitos. Saatavissa: https://jyx.jyu.fi/dspace/bitstream/handle/123456789/24657/Vesa.Vilkman.pdf?sequence=1. Hakupäivä 28.1.2018.

11. GitLab Continuous Integration & Deployment. GitLab Inc. Saatavissa: https://about.gitlab.com/features/gitlab-ci-cd/. Hakupäivä 14.2.2018.

12. Terveydenhuollon laitteiden lakisääteiset määräykset, 2015. Verkko-opas. Tekes. Saatavissa: http://reguloiko.fi/content/Terveydenhuollon_laitteiden_lakisaateiset_maaraykset/Html5_training/index.html. Hakupäivä 28.1.2018.

13. Laki terveydenhuollon laitteista ja tarvikkeista 629/2010. Naantali. Sosiaali- ja terveysministeriö. 24.6.2010. Saatavissa: https://www.finlex.fi/fi/laki/alkup/2010/20100629. Hakupäivä 28.1.2018.

14. A painless saelf-hosted Git service, 2017. Gogs. Saatavissa: https://gogs.io/. Hakupäivä 14.2.2018.

15. Build great things at any scale. Jenkins. Saatavissa: https://jenkins.io/. Hakupäivä 14.2.2018.

16. GitLab. Saatavissa: https://about.gitlab.com/products/. Hakupäivä 14.2.2018.

17. GitLab-Runner. GitLab Inc. Saatavissa: https://docs.gitlab.com/runner/. Hakupäivä 14.2.2018.

18. Docker Community Edition. Docker Inc. Saatavissa: https://www.docker.com/community-edition. Hakupäivä 14.2.2018.

LIITTEET

Liite 1 Lähtötietomuistio

Liite 2 Node.js-sovelluksen CI-määritystiedosto (.gitlab-ci.yml)

Liite 3 Node.js-sovelluksen Docker-imagen määritystiedosto (Dockerfile)

Liite 4 Node.js-sovelluksen Docker-imagen CI-määritystiedosto (.gitlab-ci.yml)

Liite 5 Node.js-sovelluksen testaustiedostot

Liite 6 Node.js-sovelluksen lopullinen CI-määritystiedosto (.gitlab-ci.yml)

Liite 7 Node.js-sovelluksen lähdekoodit

Liite 8 Android-sovelluksen moduulitestauksen lähdekoodit

Liite 9 Android-sovelluksen integraatiotestauksen lähdekoodit

Liite 10 Android-sovelluksen lähdekoodit

Liite 11 Android-sovelluksen CI-määritystiedosto (.gitlab-ci.yml)

Liite 12 Android-sovelluksen Docker-imagen määritystiedosto (Dockerfile)

Liite 13 Android-sovelluksen Docker-imagen CI-määritystiedosto (.gitlab-ci.yml)

LÄHTÖTIETOMUISTIO

Tekijä[1] Esa Hannila

Tilaaja[2] Nukute Oy

Tilaajan yhdyshenkilö ja yhteystiedot[3]

Juha-Pekka Syrjälä

Työn nimi[4] Ohjelmistojen automaattinen testausjärjestelmä

Työn kuvaus[5]

Testausjärjestelmän suunnittelu ja toteutus tuotekehityksen laadunvarmistamisen ja hyväksyntäprosessien vaatimusten mukaiseen automatisoituun testaamiseen. Toteutettava järjestelmän tulee suorittaa toistettavissa olevia testaustoimenpiteitä, sekä tuottaa ymmärrettäviä raportteja.

Tuotekehityksessä ei ole käytössä toimivaa testausjärjestelmää eikä testausmenetelmiä.

Järjestelmän tulee olla laajennettavissa.

Järjestelmän kehityksessä ratkaistaan ohjelmistotestaukseen liittyviä ongelmia, jotta testaustoimenpiteet ovat riittävän kattavat sekä toistettavissa olevat.

Työn tavoitteet[6]

Tavoitteena on toteuttaa tuotekehityksen tueksi automatisoitu integrointitestausjärjestelmä.

Tavoiteaikataulu[7]

Projektin tavoitteellinen valmistumisaika on helmikuu 2018 loppuun mennessä.

Päiväys ja allekirjoitukset[8]

---

[1] Tekijän nimi, puhelinnumero ja sähköpostiosoite.
[2] Työn teettävän yrityksen virallinen nimi.
[3] Sen henkilön nimi ja yhteystiedot, joka yrityksessä valvoo työn suoritusta.
[4] Työn nimi voi olla tässä vaiheessa työnimi, jota myöhemmin tarkennetaan.
[5] Työ kuvataan lyhyesti. Siinä esitetään muun muassa työn tausta, lähtötilanne ja työssä ratkaistavat ongelmat.
[6] Esitetään lyhyesti ja selvästi työn tavoitteet.
[7] Esitetään projektin tavoiteaikataulu. Silloin, kun työllä on välitavoitteita, myös ne merkitään aikatauluun. Tavoiteaikataulun ja oppilaitoksen yleisaikataulun perusteella tekijä laatii oman aikataulunsa.
[8] Lähtötietomuistio päivätään ja sen allekirjoittavat tekijä ja tilaajan yhdyshenkilö.

```
image: node:6


before_script:

  - npm install


cache:

  paths:

    - node_modules/


stages:

  - run


run:

  stage: run

  script:

    - npm start
```

```
FROM node:6


RUN apt-get update

RUN apt-get install -y \

    nodejs \

    nodejs-legacy \

    npm
```

```
image: docker:latest


variables:

  ACCESS_TOKEN: "<PÄÄSYAVAIN>"

  URL: <IP.OSOITE>:3002

  GROUP: "demo"

  PROJECT: "demo"


services:

  - docker:dind


stages:

  - build


before_script:

  - echo $ACCESS_TOKEN | docker login -u gitlab-ci-token --password-stdin $URL

  - docker info


build_image:

  stage: build

  script:

    - docker build -t $URL/$GROUP/$PROJECT:latest .

    - docker push $URL/$GROUP/$PROJECT
```

calculator.js

```
let assert                  = require('assert'),
    Calculator

describe('Calculator module test', () => {
    it( 'Should load calculator module without problems', () => {
        Calculator          = require('../../lib/modules/calculator')
    })
    it( 'Should calculate sum of two given values', () => {
        Calculator.add( 58, 24, res => {
            assert.equal( 82, res )
        })
    })
})
```

validator.js

```
let assert                  = require('assert'),
    Validator

describe('Validator module test', () => {
    it( 'Should load validator module without problems', () => {
        Validator          = require('../../lib/modules/validator')
    })
    it( 'Should return true when testing if value 5 is above zero', next => {
        if( Validator.isAboveZero( 5 ) )
            next()
        else
            next( '#isAboveZero')
    })
})
```

integration.js

```
let Validator,
    Calculator


describe('Basic integration test', () => {
    it( 'Should load modules without problems', () => {
        Validator           = require('../lib/modules/validator')
        Calculator          = require('../lib/modules/calculator')
    })
    it( 'Should return false on 140 subtracted by 150 is above zero', next => {
        if( !Validator.isAboveZero( Calculator.subtract( 140, 150 ) ) )
            next()
        else
            next( '#isAboveZero or #subtract unwanted result')
    })
})
```

system.js

```
describe('Demo Node.js', () => {
      it('Should complete succesfully', next => {
            next()
      })
      describe('Run application without syntax error', () => {
            it( 'Should complete without issue', () => {
                  require('../')
            })
      })
})
```

```
image: <IP.OSOITE>:3002/demo/demo:latest


before_script:
  - npm install


cache:
  paths:
    - node_modules/


stages:
  - "module test"
  - "integration test"
  - "system test"
  - run


calculator:
  stage: "module test"
  script:
    - npm test ./test/modules/calculator.js


validator:
  stage: "module test"
  script:
    - npm test ./test/modules/validator.js


integration:
  stage: "integration test"
  script:
    - npm test ./test/integration.js
```

```
system:

  stage: "system test"

  script:

    - npm test ./test/system.js



run:

  stage: run

  script:

    - npm start
```

./index.js

```
module.exports = exports = require('./lib')
```

./package.json

```
{
  "name": "demo",
  "version": "1.0.0",
  "main": "demo.js",
  "scripts": {
    "test": "mocha",
    "start": "node ./bin/demo"
  },
  "repository": {
    "type": "git",
    "url": "http://<IP.OSOITE>/demo/demo.git"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "chai": "^4.1.2",
    "mocha": "^5.0.0"
  }
}
```

./bin/demo

```
'use strict'
console.log('Loading demo Node.js application')
const App        = require('../')
let app               = new App()
app.start()
```

./lib/index.js

```
const      Calculator             = require('./modules/calculator'),
           Validator              = require('./modules/validator')


class App {
      constructor() {
            console.log('Creating application')
      }
      start() {
            console.log('Application started succesfully')
      }
}
module.exports = exports = App
```

./bin/modules/calculator.js

```
class Calculator {
    static add( val1, val2, next ) {
        if( typeof next === 'function' )
            next( val1 + val2 )
        return val1 + val2
    }
    static subtract( val1, val2, next ) {
        if( typeof next === 'function' )
            next( val1 - val2 )
        return val1 - val2
    }
}
module.exports = exports = Calculator
```

./bin/modules/validator.js

```
class Validator {

    static isAboveZero( value, next ) {

        if( typeof next === 'function' )

            next( value > 0 )

        return value > 0

    }

}

module.exports = exports = Validator
```

CalculatorModuleTest.java

```
public class CalculatorModuleTest {


    Calculator calc     = new Calculator();


    @Test

    public void testAdd() throws Exception {

        assertEquals( calc.add( 5, 10 ), 15 );

    }


    @Test

    public void testSubtract() throws Exception {

        assertEquals( calc.subtract( 65, 82 ), -17 );

    }


    @Test

    public void testDivide() throws Exception {

        assertEquals( calc.divide( 10, 2 ), 5 );

    }


}
```

ValidatorModuleTest.java

```
public class ValidatorModuleTest {


    Validator valid     = new Validator();


    @Test

    public void isAboveZero() throws Exception {

        assertEquals( valid.isAboveZero( 10 ), true );

        assertEquals( valid.isAboveZero( -6 ), false );

        assertEquals( valid.isAboveZero( 0 ), false );

    }


    @Test

    public void isBelowZero() throws Exception {

        assertEquals( valid.isBelowZero( -10 ), true );

        assertEquals( valid.isBelowZero( 10 ), false );

        assertEquals( valid.isBelowZero( 0 ), false );

    }


    @Test

    public void isZero() throws Exception {

        assertEquals( valid.isZero( -10 ), false );

        assertEquals( valid.isZero( 10 ), false );

        assertEquals( valid.isZero( 0 ), true );

    }


}
```

TestBase.java

```
public abstract class TestBase {

    @Before

    public void setup() {

        UiDevice device                 = UiDevice.getInstance(

                            InstrumentationRegistry.getInstrumentation() );

        try {

            device.wakeUp();

        } catch (RemoteException e) {

            e.printStackTrace();

        }


        final Activity activity         = getActivityRule().getActivity();

        Runnable wakeUpDevice           = new Runnable() {

            @Override

            public void run() {

                activity.getWindow().addFlags(

                    WindowManager.LayoutParams.FLAG_TURN_SCREEN_ON |

                    WindowManager.LayoutParams.FLAG_SHOW_WHEN_LOCKED |

                    WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON );

            }

        };

        activity.runOnUiThread( wakeUpDevice );

    }

    abstract ActivityTestRule<MainActivity> getActivityRule();

}
```

UiTester.java

```
public class UiTester extends TestBase {

    @Rule

    public ActivityTestRule<MainActivity> activityRule

                       = new ActivityTestRule<MainActivity>(MainActivity.class);

    @Override

    ActivityTestRule<MainActivity> getActivityRule() {

        return activityRule;

    }


    final int FIRST_VALUE              = 52;

    final int SECOND_VALUE             = 27;

    final int EXPECTED_ADD_RESULT      = FIRST_VALUE + SECOND_VALUE;

    final int EXPECTED_SUBTRACT_RESULT  = FIRST_VALUE - SECOND_VALUE;


    @Test

    public void addTest() throws Exception {

        onView( withId( R.id.first_value_input ) ).perform( typeText( String.valueOf(

                 FIRST_VALUE ) ), closeSoftKeyboard() );

        onView( withId( R.id.second_value_input ) ).perform( typeText(

                 String.valueOf( SECOND_VALUE ) ), closeSoftKeyboard() );


        onView( withId( R.id.add_btn ) ).perform( click() );


        onView( withId( R.id.result ) ).check( matches( withText( String.valueOf(

                 EXPECTED_ADD_RESULT ) ) ) );

    }
```

```
    @Test
    public void subtractTest() throws Exception {
        onView( withId( R.id.first_value_input ) ).perform( typeText( String.valueOf(
                FIRST_VALUE ) ), closeSoftKeyboard() );
        onView( withId( R.id.second_value_input ) ).perform( typeText(
                String.valueOf( SECOND_VALUE ) ), closeSoftKeyboard() );

        onView( withId( R.id.subtract_btn ) ).perform( click() );

        onView( withId( R.id.result ) ).check( matches( withText( String.valueOf(
                EXPECTED_SUBTRACT_RESULT ) ) ) );
    }
}
```

MainActivity.java

```
public class MainActivity extends Activity implements Button.OnClickListener {


    final public static String EXTRA_RESULT         = "MainActivity.Result";

    Calculator calculator;


    EditText firstValue;

    EditText secondValue;

    Button addBtn;

    Button subtractBtn;


    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);


        calculator          = new Calculator();


        firstValue          = (EditText) findViewById( R.id.first_value_input);

        secondValue         = (EditText) findViewById( R.id.second_value_input);

        addBtn              = (Button) findViewById( R.id.add_btn);

        subtractBtn         = (Button) findViewById( R.id.subtract_btn);


        addBtn.setOnClickListener( this );

        subtractBtn.setOnClickListener( this );

    }
```

```
    @Override
    public void onClick(View view) {


        int result          = 0;
        int val1            = Integer.parseInt( firstValue.getText().toString() );
        int val2            = Integer.parseInt( secondValue.getText().toString() );
        switch (view.getId()) {
            case R.id.add_btn:
                result          = calculator.add( val1 , val2 );
                break;
            case R.id.subtract_btn:
                result          = calculator.subtract( val1 , val2 );
                break;
        }


        Intent intent           = new Intent( this, ResultActivity.class );
        intent.putExtra( EXTRA_RESULT, result );
        startActivity( intent );
    }
}
```

ResultActivity.java

```
public class ResultActivity extends Activity {



    TextView resultView;



    @Override

    public void onCreate(Bundle savedInstanceState ) {

        super.onCreate( savedInstanceState );



        setContentView( R.layout.activity_result );



        resultView          = (TextView) findViewById(R.id.result);



        int result          = getIntent().getIntExtra( MainActivity.EXTRA_RESULT, 0
);



        resultView.setText( String.valueOf( result ) );

    }

}
```

Calculator.java

```
public class Calculator {

    public int add( int val1, int val2 ) { return val1 + val2; }

    public int subtract( int val1, int val2 ) { return val1 - val2; }

    public int divide( int val1, int val2 ) { return val1 / val2; }

}
```

```
image: 10.0.2.6:3002/demo/android_demo:latest


before_script:

  - chmod +x ./gradlew

  - adb start-server


stages:

  - Build

  - "Module test"

  - "Integration test"


cache:

  paths:

     - .gradle/wrapper

     - .gradle/caches

     - app/build/outputs


Build:

  stage: Build

  script:

    - ./gradlew clean assembleDebug

  artifacts:

    paths:

    - app/build/outputs
```

```
"Module test":
  stage: "Module test"
  script:
    - ./gradlew test

  artifacts:
    name: "reports_${CI_PROJECT_NAME}_${CI_BUILD_REF_NAME}"
    when: on_failure
    expire_in: 4 days
    paths:
      - app/build/reports/tests/

"UI Test":
  stage: "Integration test"
  script:
    - ./gradlew connectedAndroidTest

  artifacts:
    name: "reports_${CI_PROJECT_NAME}_${CI_BUILD_REF_NAME}"
    when: on_failure
    expire_in: 4 days
    paths:
      - app/build/reports/tests/

after_script:
  - adb kill-server
```

```
FROM openjdk:8-jdk


ENV ANDROID_COMPILE_SDK "26"

ENV ANDROID_BUILD_TOOLS "26.0.2"

ENV ANDROID_SDK_TOOLS "24.4.1"

ENV SDK_WEB_URL
      "https://dl.google.com/android/android-sdk_r${ANDROID_SDK_TOOLS}-linux.tgz"


ENV GRADLE_HOME /opt/gradle

ENV GRADLE_VERSION 4.1

ENV GRADLE_USER_HOME .gradle


ARG GRADLE_DOWNLOAD_SHA256=
            d55dfa9cfb5a3da86a1c9e75bb0b9507f9a8c8c100793ccec7beb6e259f9ed43


RUN apt-get -q -y update && \
    apt-get -q -y install wget tar unzip lib32stdc++6 lib32z1


RUN wget --quiet --output-document=android-sdk.tgz \
      https://dl.google.com/android/android-sdk_r${ANDROID_SDK_TOOLS}-linux.tgz


RUN tar --extract --gzip --file=android-sdk.tgz && \
    rm android-sdk.tgz && \
    echo y | android-sdk-linux/tools/android --silent update sdk --no-ui --all \
            --filter android-${ANDROID_COMPILE_SDK} && \
    echo y | android-sdk-linux/tools/android --silent update sdk --no-ui --all \
            --filter platform-tools && \
    echo y | android-sdk-linux/tools/android --silent update sdk --no-ui --all \
            --filter build-tools-${ANDROID_BUILD_TOOLS} && \
```

```
    echo y | android-sdk-linux/tools/android --silent update sdk --no-ui --all \
            --filter extra-android-m2repository && \
    echo y | android-sdk-linux/tools/android --silent update sdk --no-ui --all \
            --filter extra-google-google_play_services && \
    echo y | android-sdk-linux/tools/android --silent update sdk --no-ui --all \
            --filter extra-google-m2repository

ENV ANDROID_HOME=$PWD/android-sdk-linux
ENV PATH=$PATH:$PWD/android-sdk-linux/platform-tools/:$GRADLE_HOME:$GRADLE_USER_HOME



RUN echo "Downloading gradle" && \
    wget --quiet -O gradle.zip "https://services.gradle.org/distributions/gradle-
${GRADLE_VERSION}-bin.zip" && \
    \
    echo "${GRADLE_DOWNLOAD_SHA256} *gradle.zip" | sha256sum -c - && \
    \
    echo "Installing Gradle" && \
    unzip gradle.zip && \
    rm gradle.zip && \
    mv "gradle-${GRADLE_VERSION}" "${GRADLE_HOME}/" && \
    ln -s "${GRADLE_HOME}/bin/gradle" /usr/bin/gradle
```

```
image: docker:latest


variables:
  ACCESS_TOKEN: "<PÄÄSYAVAIN>"
  URL: <IP.OSOITE>:3002
  GROUP: "demo"
  PROJECT: "android_demo"


services:
  - docker:dind


stages:
  - build


before_script:
  - echo $ACCESS_TOKEN | docker login -u gitlab-ci-token --password-stdin $URL
  - docker info


build_image:
  stage: build
  script:
    - docker build -t $URL/$GROUP/$PROJECT:latest .
    - docker push $URL/$GROUP/$PROJECT
```