PROTOTYYPPILAITTEIDEN AUTOMATISOITU RAUTAPOHJAISEN SUORITUSKYVYN TESTAUS

Raili Peippo

Opinnäytetyö
Joulukuu 2009

Tietotekniikka

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) PEIPPO, Raili	Julkaisun laji Opinnäytetyö	Päivämäärä 03.12.2009
	Sivumäärä 68	Julkaisun kieli Suomi
	Luottamuksellisuus ( ) saakka	Verkkojulkaisulupa myönnetty (X)

Työn nimi
PROTOTYYPPILAITTEIDEN AUTOMATISOITU RAUTAPOHJAISEN SUORITUSKYVYN TESTAUS

Koulutusohjelma

Tietotekniikka

Työn ohjaaja(t)
SALMIKANGAS, Esa, VÄLIVERRONEN, Jukka

Toimeksiantaja(t)
Digia Oyj

Tiivistelmä
Tämän opinnäytetyön toimeksiantaja oli Digia Oyj. Digia Oyj on ohjelmistokehitysyritys, joka toimittaa tieto- ja viestintäteknologian ratkaisuja maailmanlaajuisesti. Digia Oyj on myös vahva toimija mobiililaitteiden ohjelmistotestauksessa.

Tämän opinnäytetyön tarkoituksena oli löytää toimiva konsepti rautapohjaisten vikojen löytämiseksi prototyyppilaitteista testaamalla enimmäkseen automatisoidusti laitteiden rautapohjaista suorituskykyä. Konseptin käytännön toteutuksen oli oltava sekä nopeasti että mahdollisimman vähällä työmäärällä toteutettavissa, jotta ohjelmistotestauksesta ei tarvitsisi kuluttaa liikaa resursseja. Myös prosessin, jonka perusteella rautapohjaiset viat erotetaan ohjelmistopohjaisista virhetiloista, oli oltava selkeä ja yhtenäinen.

Digia Oyj:n kohdetoimialue mobiililaitteiden testauksessa on ohjelmistopohjainen testaus. Yhden mobiililaitteen mallin kehityskaaren aikana tehdään kuitenkin useita versioita sekä rautatoteutuksesta että ohjelmistototeutuksesta. Koska ohjelmistotestauksessa käytetään vielä kehitysvaiheessa olevia laitteita, on mahdollista, että myös prototyyppilaitteen rautatoteutuksen valmistuksen yhteydessä on tapahtunut virheitä. Tästä johtuen on tärkeää löytää rautapohjaisesti rikkinäiset laitteet ennen niiden laittamista ohjelmistotestaukseen. Rikkinäisillä laitteilla tehdyn ohjelmistotestauksen tulosten paikkansapitävyys on kyseenalaista, joten kyseiset testitapaukset pitää suorittaa uudelleen toimivalla laitteella. Uudelleen testaus taas aiheuttaa turhaa työtä ja kuluttaa resursseja.

Testauskonseptiin suunniteltiin sekä manuaalista suorituskyvyn testausta että automatisoitua kestävyystestausta. Käytännön toteutuksesta saatujen tulosten perusteella voidaan todeta, että on mahdollista erottaa selkeästi rautapohjaiset viat ohjelmistopohjaisista käyttämällä testauskonseptissa suunniteltuja testausprosesseja. Tuloksien vahvistukseksi ajettu pidempi referenssiajo varmisti testauskonseptin tuloksien paikkaansapitävyyden ja toimivuuden. Myös vaatimukset mahdollisimman pienestä työmäärästä sekä ajan kulusta käytännön toteutuksen suhteen onnistuttiin täyttämään.

Avainsanat (asiasanat)
Testaus, automaatio, suorituskykytestaus, kestävyystestaus, ei toiminnallinen testaus

Muut tiedot

Author(s)
PEIPPO, Raili

Type of publication
Bachelor´s Thesis

Date
03.12.2009

Pages
68

Language
Finnish

Confidential

( ) Until

Permission for web publication
(X)

Title
AUTOMATED HARDWARE PERFORMANCE TESTING OF PROTOTYPE DEVICES

Degree Programme
Information Technology

Tutor(s)
SALMIKANGAS, Esa, VÄLIVERRONEN, Jukka

Assigned by
Digia Oyj

Abstract
This Bachelors Thesis was assigned by Digia Oyj. Digia Oyj is a software company, which delivers information and communication technology solutions worldwide. Digia Oyj is also oriented to software testing of mobile devices.

The purpose of this Bachelor's Thesis was to find a working concept for testing automatically hardware performance of prototype devices in order to find possible hardware based faults. The main requirements of the concept were that the hands-on implementation should be done as fast and with as little work as possible, due to the limited amount of recourses. In addition the process of isolating hardware based bugs from software based errors should be clear and coherent.

Digia's main scope in mobile testing is to find software bugs from the software versions. However there are several software and hardware versions developed during development process of one single model and therefore it is possible that errors have occurred during the making of hardware versions. Thus it is important to find faulty hardware versions before those are used in software testing. Testing results are not valid if faulty hardware has been used for testing and test cases need to be retested.

The testing concept was designed to use both manual performance and automated reliability testing. According to the results of the executed test concept it is possible to clearly isolate the hardware based faults from software based bugs by using the processes designed in the testing concept. The longer reference test execution verified that the results from the hands-on testing of the concept were valid. Also the requirements regarding the minimal time consumption and work efforts were met in the testing concept.

Keywords
Software testing, automated testing, reliability testing, performance testing, non-functional testing

Miscellaneous

SISÄLTÖ

LYHENTEET JA TERMIT 4

1 TYÖN LÄHTÖKOHDAT 6

1.1 Toimeksiantaja 6
1.2 Tehtävän kuvaus 6

2 OHJELMISTOTESTAUS 8

2.1 Testauksen tarkoitus 8
2.2 Testauksen V-malli ja tasot 12
2.3 Musta- ja lasilaatikkotestaus 18
2.4 Toiminnallinen testaus 20
2.5 Ei-toiminnallinen testaus 20

3 AUTOMATISOITU TESTAUS 22

3.1 Yleisesti 22
3.2 Testitapauksien automatisointi 24
3.3 Automatisoitu älypuhelintestaus 25

4 ÄLYPUHELINTESTAUS 28

4.1 Toimeksiantajan tekemä ei-toiminnallinen testaus 28
4.2 Kestävyystestaus 28
4.3 Suorituskyvyn testaus 30

5 TESTAUSKONSEPTIN TAVOITE JA SUUNNITTELU 32

5.1 Tavoite 32
5.2 Suunnittelu 35
5.3 Testausympäristö 39

6 TESTAUSKONSEPTIN TOTEUTUS 42

6.1 Testauskonseptin toteutuksen pohjatyö 42
6.2 Testauskonseptin toteutus 45
6.3 Tulokset 47
6.3.1 Käynnistysajat ja ajoajat 47
6.3.2 Testitapausjoukkokohtaiset tulokset 51
6.3.3 Testitapauskohtaiset tulokset 53

7 OPINNÄYTETYÖN JÄLKITARKASTELU 61

7.1 Pohdinta 61

7.2 Työn onnistuminen 63

LÄHTEET 65

KUVIOT

KUVIO 1. Ohjelmistovirheen kustannuksen kasvu ohjelmiston elinkaaren aikana (Hambling, Morgan, Samaroo, Thompson & Williams 2007). 10

KUVIO 2. V-malli ohjelmistosuunnitteluun (Hambling ym. 2007, 36). 12

KUVIO 3. Ylhäältä alas ja alhaalta ylös integraatiotestauksen strategiat (Hamblingin ym. 2007, 41 - 42). 15

KUVIO 4. Musta- vs. lasilaatikkotestaus 19

KUVIO 5. Ei toiminnallinen testaus V-mallissa 21

KUVIO 6. Automatisoidun testauksen kuusi askelta Li ja Wu (2004). 24

KUVIO 7. Automatisointiohjelmiston eri osat 40

KUVIO 8. Testausympäristö 41

KUVIO 9. Prototyyppilaitteiden käynnistysnopeudet 48

KUVIO 10. 17 testijoukkoiteraation ajoajat prototyyppilaitteittain 49

KUVIO 11. 60 testijoukkoiteraation ajoajat prototyyppilaitteittain 50

KUVIO 12. Kaikkien testiajojen kestot sekä ajoajat yhteensä prototyyppilaitteittain 50

KUVIO 13. Onnistuneet iteraatiot maksimi-iteraatiomäärän ollessa 204 . 52

KUVIO 14. Onnistuneet iteraatiot maksimi-iteraatiomäärien ollessa 720 53

KUVIO 15. Testitapaus ”Kontaktin lisääminen Puhelinluetteloon” iteraatiot 54

KUVIO 16. Testitapaus ”Kuvan ottaminen kameralla” iteraatiot 54

KUVIO 17. Testitapaus ”Internet-sivujen selailu WLAN-verkon yli” iteraatiot 55

KUVIO 18. Testitapaus ”Profiilin vaihto yleisestä äänettömään ja takaisin” iteraatiot 56

KUVIO 19. Testitapaus ”Tiedostoja sisältävän kansion kopioiminen ulkoiselta muistikortilta massamuistiin” iteraatiot. 56

KUVIO 20. Testitapaus Muistinkulutuksen sen hetkisen tilan taltioiminen iteraatiot 57
KUVIO 21. Testitapaus ”Muistikortilla sijaitsevan MP3-musiikkitiedoston toisto” iteraatiot 57
KUVIO 22. Testitapaus ”Hälytys kelloon” iteraatiot 58
KUVIO 23. Testitapaus ”SMS viesti 2G-verkon yli” iteraatiot 58
KUVIO 24. Testitapaus ”SMS viesti 3G-verkon yli” iteraatiot 59
KUVIO 25. Testitapaus ”Äänipuhelu 2G-verkon yli” iteraatiot 59
KUVIO 26. Testitapaus ”Äänipuhelu 3G-verkon yli” iteraatiot 60

Taulukot

TAULUKKO 1. Ohjelmistovirheen kustannuksen komparatiivinen nousu ohjelman elinkaaren aikana 9
TAULUKKO 2. Testitapaukset ja niiden testaamat prototyyppilaitteen komponentit. 36
TAULUKKO 3. Skriptien toiminnot testitapauksittain 43

LYHENTEET JA TERMIT

2G	Toisen sukupolven matkapuhelinteknologian digitaalinen standardi verkkoteknologialle, tässä tapauksessa tarkoitetaan eurooppalaista GSM standardia.
3G	Kolmannen sukupolven matkapuhelinteknologian digitaalinen standardi verkkoteknologialle, tässä tapauksessa tarkoitetaan Euroopan yleisintä UMTS standardia.
Aloitustila	Aloitustila on näkymä, johon prototyyppilaite jää käynnistyksen jälkeen ilman mitään käyttäjän tekemiä toimintoja.
Bluetooth, BT	Lyhyen kantaman radiotekniikkaan perustuva langaton tiedonsiirtomedia.
Dongle	PC:seen yleensä USB-liitännän kautta liitettävä lisälaitelaite, esimerkiksi BT vastaanotin tai ulkoinen muisti.
Emulaattori	Tietokoneohjelma tai laitteistolaajennus, joka mahdollistaa tietyn ohjelman tai laitteen käytön muunlaisella tietokoneella tai käyttöjärjestelmällä kuin mille se on alun perin tarkoitettu.
MMS	Multimedia Messaging Service, viesti johon on lisätty multimediaobjekti kuten kuva tai video.
MP3	MPEG-1 Audio Layer 3, hyvin yleinen MPEG-1 -standardiin perustuva häviöllinen äänenpakkausmenetelmä.
NDA	Non-Disclosure Agreement, Salassapitosopimus.
PC	Personal Computer. Henkilökohtainen tietokone.
Series 60	Alusta Symbian-käyttöliittymää käyttäville älypuhelimille. Esimerkiksi Nokia N95.
Skripti	Komentosarja eli lyhyt ohjelma, joka toimii rajatussa ympäristössä.
SMS	Short Message Service, matkapuhelinten tekstiviestijärjestelmä.
UI	User Interface, käyttöliittymä.

USB	Universal Serial Bus, sarjaväyläarkkitehtuuri usean laitteen välistä tiedonsiirtoa varten.
XML	eXtensible Markup Language, merkintäkieli tai standardi, jolla tiedon merkitys on kuvattavissa tiedon sekaan.
WLAN	Wireless Local Area Network, langaton lähiverkko.

1 TYÖN LÄHTÖKOHDAT

1.1 Toimeksiantaja

Digia Oyj / Smartphone Business Division

Suomesta kotoisin oleva Digia Oyj tunnetaan ketteränä ja innovatiivisena ohjelmistoyhtionä, joka toimittaa tieto- ja viestintäteknologiaratkaisuja maailmaanlaajuisesti. Digia Oyj on sijoittunut Pohjoismaihin, mutta toimipisteitä löytyy myös mm. Virosta, Venäjältä ja Kiinasta. Yritys on listattuna OMX Pohjoismaisessa Pörssissä Helsingissä (DIGI1V).

Digia työllistää tällä hetkellä yli 1300 ammattilaista, ja suuri osa yrityksen tietotaidosta on kanavoitu älypuhelimiin ja reaaliaikaisissa tietojärjestelmissä toimiviin mobiileihin ratkaisuihin. Mobiiliohjelmisto-osaamisen lisäksi Digia Oyj tarjoaa asiakkailleen palveluja ja tuotteita myös finanssin ja palveluiden, teollisuuden ja kaupan sekä telekommunikaation liiketoiminta-alueilla.

Telekommunikaation liiketoiminta-ala on jakautunut älypuhelin- ja operaattoridivisiooniin. Älypuhelindivisioonaan eli Smartphone Business Divisioniin kuuluu sekä älypuhelinohjelmistojen kehitys että testaus. Opinnäytetyö tehtiin Smartphone Business Divisioonalle. (Digia 2008.)

1.2 Tehtävän kuvaus

Opinnäytetyön tarkoituksena oli kehittää konsepti, jolla voidaan testata kaikki saapuneet prototyyppipuhelimet ennen virallisen testauksen aloittamista. Työn aloitushetkellä ei ollut mitään yleistä prototyyppien testausta, vaan saapuneet puhelimet otettiin suoraan käyttöön viralliseen testaukseen. Käytännössä tämä tarkoitti sitä, että rautatason erot eri puhelinten välillä jäivät huomaamatta tai sitten vaihtoehtoisesti löytyivät virallisen testauksen yhteydessä, jolloin tuloksien luotettavuus saattoi kärsiä.

Nykyisin toimeksiantajan puhelintestaus on painottunut lähinnä automatisoituun kestävyyden testaukseen sekä manuaalisesti tehtävään suorituskyvyn testaukseen. Molemmilla testausalueilla testauksen tarkoitus on painottunut hyvin pitkälti ohjelmistopohjaisten ongelmien ja virheiden löytämiseen. Valitettavasti prototyyppipuhelinten välillä on huomattavissa joskus jo rautatasolta lähteviä toiminnallisia eroja, joita ei voi selittää ohjelmistopohjaisilla ongelmilla. Tällaisten erojen löytäminen mahdollisimman aikaisessa vaiheessa on hyvin tärkeä varsinkin manuaalisesti tehtävän suorituskyvyn mittauksen kannalta.

Opinnäytetyön onnistuneen toteutuksen suurimpia haasteita oli erottaa ohjelmistovirheet rautapohjaisen suorituskyvyn aiheuttamista virheistä. Esimerkiksi perinteistä automatisoitua kestävyystestausta, jossa ajetaan hyvin pitkään samoja testitapauksia nimenomaan ohjelmistovikojen ja muistivuotojen löytämiseksi, ei voi sellaisenaan käyttää. Työn tarkoituksena oli kuitenkin löytää nimenomaan rautapohjaiset viat eikä ohjelmistopohjaisia vikoja. Manuaalisesti tehtävää suorituskyvyn testausta taas ei automaatiosta johtuvista rajoitteista – esimerkiksi kuvan tunnistukseen saattaa kulua tietokoneelta huomattavasti pidempi aika kuin ihmiseltä – johtuen voida myöskään käyttää, sillä testauskonseptin käytännön toteutuksen on oltava mahdollisimman pienellä työmäärällä toteutettavissa. Näin ollen opinnäytetyön käytännön osuus on toteutettava mahdollisimman pitkälti automatisoidusti.

Omat rajoitteensa käytännön toteutukselle asettavat resurssiongelmat, sillä projektissa työskentelee vain tietty määrä ihmisiä ja virallisen testauksen aikataulujen ollessa välillä hyvinkin kiireisiä, ylimääräistä aikaa ei juuri jää muiden tehtävien tekemiseksi. Koko konseptin toteutuksen on siis oltava mahdollisimman nopeasti ja hyvin pienillä muokkauksilla käyttöön otettavissa alun kehitysvaiheen jälkeen.

Oma lukunsa on myös automaatioympäristö, sillä puhelinprototyyppien kehityksen ohessa myös testausautomaatioympäristöä kehitetään jatkuvasti. Tästä seuraa valitettavasti myös silloin tällöin yhteensopivuusongelmia joko skriptien tai prototyyppien ja itse ympäristön välillä. Samoin välillä esiintyy regressiotestauksen jäädessä vähän vähemmälle uusia ongelmia aikaisemmin jo toimineissa ympäristön ominaisuuksissa.

Näin ollen opinnäytetyön tarkoituksena oli kehittää luotettava, mahdollisimman nopeasti ja vähällä työmäärällä suoritettava testauskonsepti, jonka avulla vialliset prototyypit myös oikeasti voidaan seuloa pois virallisesta testauksesta. Testitapaukset eivät tietenkään pysy samoina ajan kuluessa, sillä jo pelkästään älypuhelinten ominaisuuksien kehitys ja uusien ominaisuuksien mukaan liittäminen vaativat joustavuutta koko konseptilta.

2 OHJELMISTOTESTAUS

2.1 Testauksen tarkoitus

Blackin (2007) mukaan testauksen tarkoituksen kuvitellaan usein olevan sen toteennäyttäminen, että ohjelmisto toimii täydellisesti. Samoin uskotaan, että kun ohjelmisto on testauksen ja tuloksien analysoinnin jälkeen hyväksytty tuotantoon, siinä ei voi löytyä enää yhtään virhettä, koska testauksessa on käyty läpi kaikki mahdolliset virhetilanteet. (Black 2007, 6.)

Käytännössä tämä on täysin mahdotonta. Kuka tahansa yhtään kokenut ohjelmoija ja testaaja tietää, ettei täysin virheetöntä ohjelmistoa voi enää tehdä sillä nykyiset, ominaisuuksiltaan erittäin monimuotoiset laitteet käyttävät hyvin monimutkaisia ja laajoja ohjelmistokokonaisuuksia. Tiedon siirto, käyttö, tallennus ja uudelleen käyttö eri ohjelmistojen välillä sekä ohjelmistojen sisällä on nykyisin niin monimutkaista, että virhetilanteita tulee väkisin vastaan. Tähän auttaa tietysti paljon virhetilanteiden ennakointi ja testaus, mutta kaikkien mahdollisten vikatilojen ennakointi on mahdotonta. Ohjelmiston loppukäyttäjien kaikkia mahdollisia toimintoja ei voida ennustaa. Esimerkiksi tekniseltä osaamiseltaan heikompi loppukäyttäjä saattaa kokeilla syötettä, jota teknisten taitojensa puolesta kokeneempi testaaja ei voisi edes kuvitella. Täydellistä kaiken kattavaa testausta, jossa kaikki mahdolliset syötteet ja esiehtojen yhdistelmät testataan, ei olisi mahdollista toteuttaa, vaikka budjetti, käytettävissä oleva aika ja resurssit olisivat äärettömät. (Black 2007, 6.)

Ohjelmiston kehitysprosessissa ilmentyneet virheet korjataan seuraavaan versioon, mutta korjauksen mahdollisesti aiheuttamat muutokset ja uudet, ei vielä esiin tulleet virhetilanteet voivat taas vaikuttaa hyvinkin paljon koko ohjelmiston toimivuuteen. Käytännössä siis, jotta täydellisen testauksen määritelmä täyttyisi, pitäisi jokaisen uuden ohjelmistoversion testauksessa käydä läpi aina uudestaan kaikki mahdolliset syötteet, toiminnallisuudet ja esiehtojen yhdistelmät. Tämä taas vaatii suunnattomasti resursseja, aikaa ja rahaa, jolloin ohjelmiston taloudellinen hyöty ja mahdollinen innovatiivisuus muihin vastaaviin ohjelmistoihin nähden vähenisivät käänteisesti ajan ja rahan kulutukseen nähden.

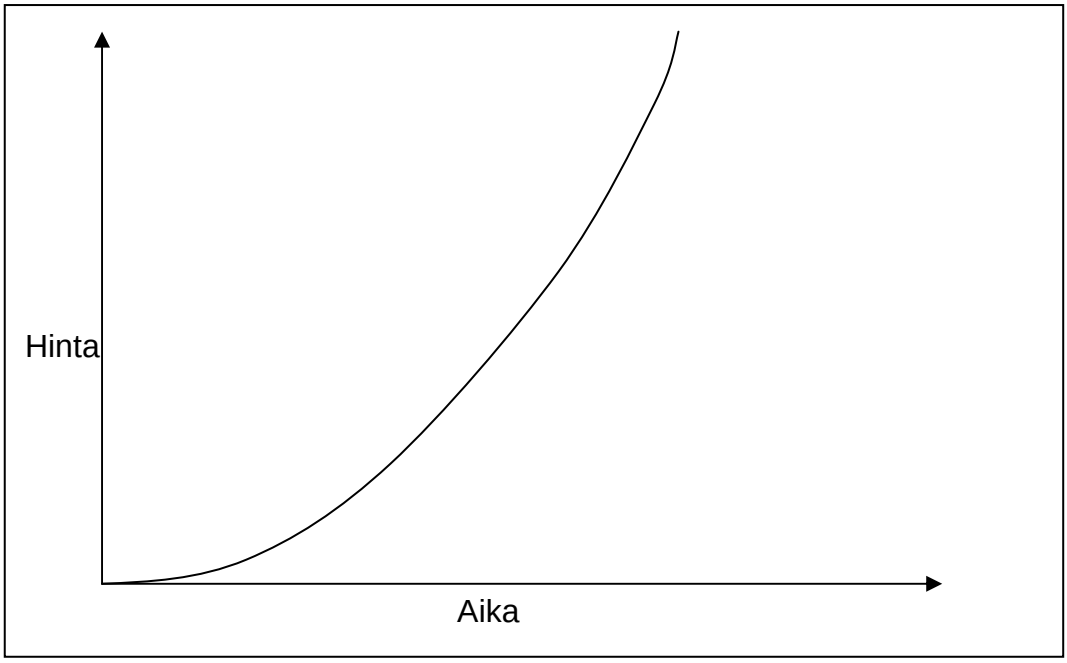
Testaus on kuitenkin hyvin tärkeä osa-alue ohjelmiston kehitysprosessissa. Mahdollisimman vähän virheitä sisältävä ohjelmisto vähentää muun muassa ylläpidon kustannuksia, kasvattaa yrityksen brändin positiivista mainetta luotettavana ja laatuun panostavana tekijänä alalla sekä vähentää ohjelmiston uusien versioiden tekemiseen kuluvaa aikaa ja resursseja perusteiden ollessa hyvässä kunnossa.

Taulukko 1 on kustannusten indeksisidonnainen malli (Cost escalation model), ja taulukossa on esitetty yhden ohjelmistovirheen suhteellinen ja komparatiivinen hinta ohjelmiston kehityksen eri tasojen aikana (Hambling, Morgan, Samaroo, Thompson & Williams 2007, 18).

TAULUKKO 1. Ohjelmistovirheen kustannuksen komparatiivinen nousu ohjelman elinkaaren aikana

Vaihe jolloin virhe löytyy.	**Komparatiivinen kustannus**
Määrittely	$1
Ohjelmointi	$10
Moduulitestaus	$100
Järjestelmätestaus	$1,000
Hyväksymistestaus	$10,000
Ohjelmisto on julkisessa käytössä	$100,000

Toki aina yksi yksittäinen virhe ei maksa näin suuria summia, mutta mitä pidemmälle ohjelmiston kehityksessä mennään, sitä enemmän löydettyjen virheiden korjaaminen tulee maksamaan. Esimerkiksi jo järjestelmätestauksessa löydettyihin virheisiin tarvitaan erillinen testaaja, joka tutkittuaan virhettä tekee siitä virheraportin. Tämän jälkeen virhe korjataan, mutta tarvitaan lisää testausta sekä moduulitasolla että järjestelmätasolla, jotta voidaan varmistaa, ettei virheen korjaaminen aiheuttanut muita virheitä moduulitasolla tai korjatun ohjelmiston ominaisuudesta riippuvaisissa moduuleissa. Tämä kaikki vie resursseja, aikaa ja työtä kaikilta virheen löytämiseen, korjaamiseen ja uudelleen testaukseen osallistuvilta työntekijöiltä. Kuvio 1 esittää miten virheiden kustannus kasvaa sen mukaan missä vaiheessa ohjelmiston elinkaarta ne löydetään ja korjataan.



KUVIO 1. Ohjelmistovirheen kustannuksen kasvu ohjelmiston elinkaaren aikana (Hambling, Morgan, Samaroo, Thompson & Williams 2007).

Kuten aikaisemmin mainittu, täydellinen kaiken kattava testaus ei ole mahdollista, joten ohjelmiston testauksessa on löydettävä tasapaino ajan, resurssien ja rahan käytön välillä laadun kuitenkaan kärsimättä siitä. Käytännössä tämä toteutetaan niin, että ensiksi varmistetaan ohjelmiston kriittisten ominaisuuksi-

en toimivuus, minkä jälkeen keskitytään pienempien mahdollisten virhetilanteiden testaamiseen.

Vaikka kaiken kattava testaus ei olekaan mahdollista, testauksen toteutuksen tulee olla hyvin suunniteltua ja järjestelmällistä resurssien käytön, testauksen laadun ja kulujen optimoimiseksi. Järjestelmällinen testaus toteutetaan suunnittelemalla ensiksi testitapausmassa, joka määritellään valmiiksi ennen käytännön testauksen aloittamista. Testitapauksien määrittämiseen käytetään testattavan järjestelmän vaatimusanalyysia, josta selviävät vaatimukset, joista testattavan järjestelmän on suoriuduttava täyttääkseen sille asetetut laatuvaatimukset. Hyvän testitapausmassan määrittäminen vaatii yleensä myös aikaisempaa kokemusta vastaavan järjestelmän testauksesta, jolloin testitapausmassaan osataan liittää myös aikaisempiin kokemuksiin perustuvia testitapauksia.

Testitapaukset määritellään kirjallisesti ennen testauksen aloittamista siksi, että kaikki testitapaukset tulisivat suoritettua järjestelmällisesti samalla tavalla kuin testitapauksien suunnittelija on vaatimusanalyysin perusteella ne määritellyt. Tämä käytäntö lisää testauksen tuloksien luotettavuutta ja oikeellisuutta, koska kaikki testitapaukset suoritetaan kirjallisten ohjeiden mukaan, jolloin yksittäisestä testaajasta johtuvat erot testitapauksien suorittamisessa minimoituvat. Testauksen luotettavuuden lisäämiseksi testaajan on hyvä olla itsenäinen ja riippumaton järjestelmän kehitysprojektista, jolloin testaus on mahdollisimman objektiivista. Näin ollen kehitysprojektin muiden jäsenten mielipiteet oman työnsä laadusta eivät pääse vaikuttamaan testaajan toimintaan. (Pääkkönen 2005, 16.)

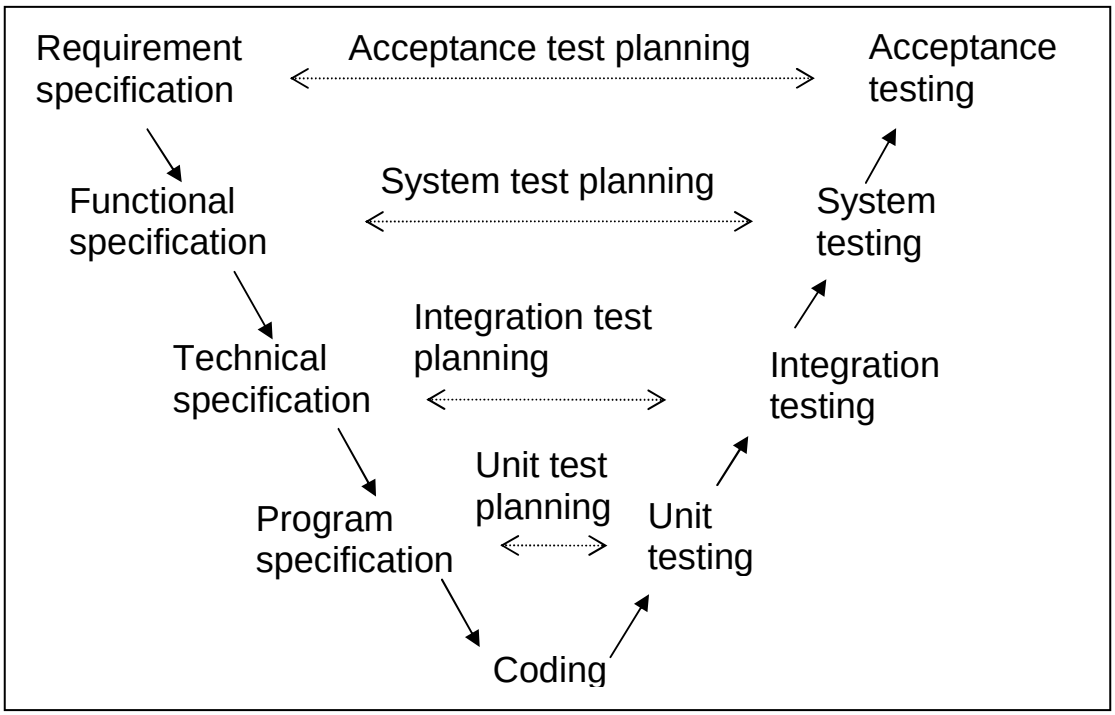
Black on listannut lähestymistapoja, joiden avulla pystyy määrittelemään tärkeät testauskohteet ohjelmistossa (mts. 74):

- Millä ohjelmiston ominaisuuksia on merkitystä asiakkaille, käyttäjille ja osakkaille?
- Mitkä ohjelmiston virheet aiheuttavat eniten ongelmia?
- Mitkä ohjelmiston virheet vaikuttavat ohjelmiston loppukäyttöön?

Ohjelmiston kehitysprosessi ja sen osa-alueena oleva testausprosessi ovat loppuvaiheessa, ja ohjelmisto on valmis julkaisuun vasta, kun kaikki löydetyt tärkeät ohjelmistoviat on korjattu ja ohjelmisto täyttää sille asetetut laatuvaatimukset. Toki sekä ohjelmiston kehitys että sen myötä testaus jatkuvat vielä jonkin aikaa julkaisun jälkeen ohjelmiston ylläpidon yhteydessä, mutta julkaisun jälkeen tähän käytetään yleensä huomattavasti vähemmän aikaa ja resursseja.

2.2 Testauksen V-malli ja tasot

V-mallia eli vesiputousmallia on käytetty jo hyvin pitkään, ja se on yksi testausmallien peruskivistä. V-mallin kaikki tasot toimivat pareina, jolloin ohjelmiston eri suunnitteluvaiheessa suunnitellaan myös vastaavat testauksen eri vaiheet. Käytännössä tämä tarkoittaa sitä, että vaatimusanalyysin yhteydessä suunnitellaan myös hyväksymistestaus, toiminnallisen määrittelyn yhteydessä järjestelmätestaus, teknisen määrittelyn yhteydessä integrointitestaus ja moduulisuunnittelun yhteydessä moduulitestaus. Kuviossa 2 näkyvät kaikki vesiputousmallin tasot.



KUVIO 2. V-malli ohjelmistosuunnitteluun (Hambling ym. 2007, 36).

V- mallin mukainen testauksen suunnittelu alkaa ylimmältä tasolta eli vaatimusanalyysin yhteydessä tehtävän hyväksymistestauksen suunnittelusta ja jatkuu jokaisen suunnitteluvaiheen kautta alas viimeisenä suoritettavaan tasoon eli moduulisuunnitteluun yhteydessä tehtävän moduulitestaukseen suunnitteluun. Näin toimittaessa on mahdollista löytää ohjelmistosuunnittelun aikana tulleet virheet jo testauksen suunnittelun yhteydessä, jolloin niiden korjaaminen maksaa huomattavasti vähemmän ja on helpompaa toteuttaa kuin ohjelmoinnin ollessa jo käynnissä.

Hamblingin ja kumppaneiden (2007) mukaan vesiputousmallin vasemman puoleinen haara tarkentuu ja laajentuu ohjelmistolle asetetuista alkuvaatimuksista teknisesti seikkaperäisemmäksi alaspäin mentäessä (mts. 36):

- Vaatimusanalyysi kattaa allensa loppukäyttäjien tarpeet ja samalla asettaa vähimmäisvaatimukset ja rajat toiminnalliselle
- Toiminnalliseen määrittelyyn kuuluu itse toiminnallisuuksien määrittely, joita tarvitaan loppukäyttäjien tarpeiden tyydyttämiseen.
- Tekniseen määrittelyyn kuuluu toiminnallisessa määrittelyssä suunniteltujen toiminnallisuuksien tekninen suunnittelu.
- Moduulisuunnittelun alle kuuluvat tarkat ja yksityiskohtaiset suunnitelmat jokaisesta yksittäisestä moduulista, jota tarvitaan teknisen määrittelyn toiminnallisuuksien toteuttamisessa.

V-mallin jokaiseen tasoon kuuluu neljä eri työvaihetta, jotka ovat testauksen suunnittelu, testausympäristön luominen, testien suorittaminen ja tulosten arvioiminen. Edellä mainitusta neljästä vaiheesta vain suunnittelu etenee vesiputousmallissa lineaarisesti ylhäältä alaspäin, loput kolme vaihetta tulevat tasoittain käänteisessä järjestyksessä, eli lineaarisesti alhaalta ylöspäin. Testauksen käytännön toteutus alkaa siis V-mallin mukaan alimmalta tasolta nousten ohjelmiston kannalta vaativimpiin ja laajempiin kokonaisuuksiin. Toki virheiden korjauksen yhteydessä palataan tarvittaessa alkuun, eli komponenttitasolla korjattu virhe testataan ensiksi yleensä virheen korjaajan toimesta moduulitestauksella ja sen jälkeen integrointitestauksella ennen kuin ohjelmiston korjattu versio laitetaan ylemmän tason testaukseen. V-mallin mukaisessa

testauksessa alemmat tasot toteutetaan mustalaatikkotestausmallin mukaisesti ja ylemmät tasot lasilaatikkotestausmallin mukaisesti.

Moduulitestaus

V-mallin alimman tason testaus eli moduulitestaus suoritetaan ohjelmiston kehityksen alkuvaiheessa, ja se keskittyy virheisiin, jotka ovat syntyneet yksittäisten komponenttien ohjelmoinnin yhteydessä. Tässä vaiheessa testattava ohjelmisto on vielä hyvin kaukana lopullisesta muodostaan, ja moduulitestauksen tarkoituksena on löytää virheet komponenttien sisältä liittyen niiden sisäiseen toimintaan, äärimmäistilanteisiin ja virhetilanteisiin. Moduulitestauksen suorittavat yleensä komponenttien kehittäjät itse, eikä siinä oteta kantaa eri komponenttien toimintaan kokonaisuutena. (Mts. 40.)

Integraatiotestaus

Integrointitestaus suoritetaan komponenttitestauksen jälkeen ja se tehdään siinä vaiheessa, kun yksittäisiä komponentteja ruvetaan liittämään suuremmiksi kokonaisuuksiksi. Integrointitestauksen tarkoitus on löytää virheitä moduulien rajapinnoista ja niiden keskinäisestä toiminnasta. Integraatiotestauksen suorittavat yleensä komponenttien kehittäjät itse. Suurimmaksi osaksi integraatiotestaus kuuluu lasilaatikkotestauksen alle, koska testaaja on yleensä itse kehittänyt testattavat komponentit ja tietää näin ollen hyvin paljon ohjelmiston toiminnoista.

On olemassa kolme eri strategiaa suorittaa integraatiotestausta: Big bang, ylhäältä alas ja alhaalta ylös integraatiotestaus. Big bang -strategian mukaan kaikki ohjelmistoon kuuluvat komponentit lisätään kerralla kokonaisuudeksi, minkä jälkeen testaus aloitetaan. Tätä strategiaa pidetään yleisesti huonona, koska sen avulla on hankalaa löytää ja eristää virheiden alkuperäiset aiheuttajat. Virhe saattaa esiintyä täysin eri komponenttien toiminnan yhteydessä kuin mistä se on alun perin aiheutunut. (Hamblingin ym. 2007, 42.)

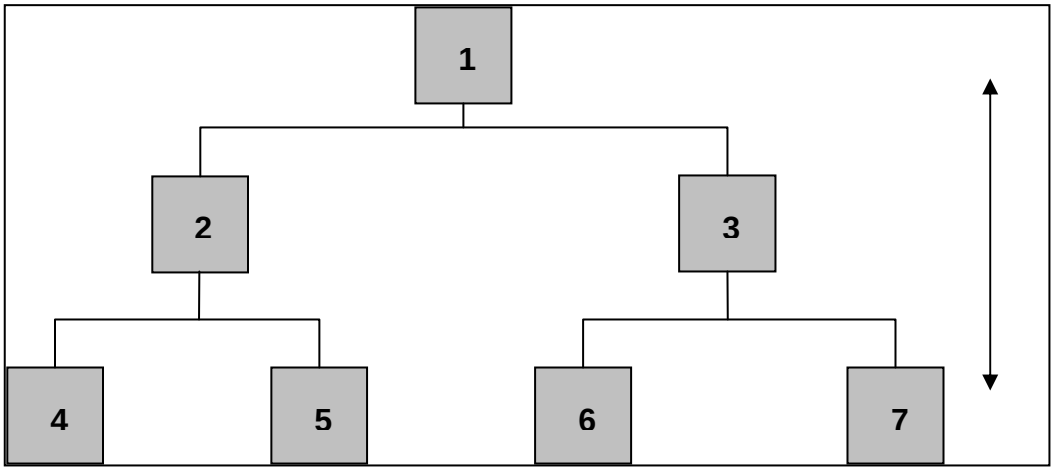
Integraatiotestauksen eri strategiat

Ylhäältä alas -strategiassa ohjelmisto on rakennettu vaiheissa aloittaen moduuleista, jotka kutsuvat muita moduuleja. Yleensä muita moduuleja kutsuvat komponentit myös sijoitetaan prioriteetiltaan muita ylemmäs. Ylhäältä alas

strategiassa testaus aloitetaankin näistä eli ylemmistä komponenteista ja siirrytään yksittäisten moduulien rajapintojen ja keskinäisen toimivuuden testaamisen jälkeen aina alemmille tasoille. Alhaalta ylös -strategia toimii muuten samalla tavalla paitsi, että rakennettavien komponenttien ohjelmointi ja testaus aloitetaan alhaalta ja siirrytään ylöspäin.

Sekä ylhäältä alas että alhaalta ylös -strategioissa kaikki komponentit eivät välttämättä ole vielä integrointivalmiita, jolloin puuttuvista komponenteista joudutaan tekemään korvaavat tynkäkomponentit, jotka toimivat ulospäin rajapintaan nähden samalla tavalla kuin mitä oikeat puuttuvat komponentitkin. Tynkämoduulit ovat passiivisia ohjelmiston osia, joita jo valmiit komponentit voivat kutsua tai ne voivat kutsua valmiita moduuleja, ja näin ollen jo ohjelmoitujen testauksen alla olevien komponenttien rajapintojen toimivuus voidaan testata. Jokaisen lisättävän komponentin rajapinta ja kyseisen komponentin yhteensopivuus muiden komponenttien kanssa testataan erikseen. Näin löytyneiden virheiden eristäminen on helpompaa ja halvempaa, koska sen voi rajata tiettyyn komponenttiin tai kokonaisuuden koko valmiin ohjelmiston sijasta. (Mts. 40-43.)

Kuviossa 3 esitellään sekä alhaalta ylös -testausstrategia että ylhäältä alas -testausstrategia. Molemmissa strategioissa on samat perusperiaatteet, mutta testauksen aloitussuunnat ovat vain täysin päinvastaisia.



KUVIO 3. Ylhäältä alas ja alhaalta ylös integraatiotestauksen strategiat (Hamblingin ym. 2007, 41 - 42).

Järjestelmätestaus

Kun komponenttien rajapintojen ja niiden keskinäinen toimivuus on testattu, aloitetaan järjestelmätestaus. Järjestelmätestauksen tarkoituksena on varmistaa ohjelmiston toimivuus kokonaisuutena sen jälkeen, kun komponenttien keskinäinen toimivuus on testattu integraatiotestauksessa. Järjestelmätestauksessa varmistetaan, että ohjelmisto toimii kokonaisuutena samassa käyttöympäristössä kuin missä loppukäyttökin toteutuu sekä komponenttien keskinäisten rajapintojen toimivuus kokonaisuutena. Järjestelmätestauksen suorittaa yleensä ohjelmiston kehityksestä erillinen työryhmä, joka on täysin riippumaton ohjelmiston kehitysprosessista. Tällä varmistetaan se, että testaus tapahtuu objektiivisesti määritettyjen testitapauksen mukaan eikä itse kirjoitetun koodin toiminnallisuutta mukaillen.

V-mallissa järjestelmätestaus perustuu toiminnalliseen määrittelyyn, ja se määritelläänkin toiminnallisen määrittelyn yhteydessä kattamaan sekä toiminnallisen että ei toiminnallisen testauksen. Järjestelmätestaus on mustalaatikkotestausta, koska testaajat eivät yleensä tiedä testaamansa ohjelmiston sisäisesti teknisestä rakenteesta kovin paljoa, eikä kyseistä tietoa myöskään tarvita järjestelmätestauksessa. (Mts. 43.)

Hyväksymistestaus

Hyväksymistestaus on V-mallin mukaisen ohjelmistotestauksen viimeinen vaihe ja se suoritetaan järjestelmätestauksen jälkeen. Hyväksymistestauksen tarkoitus on varmistaa, että ohjelmisto toimii loppukäyttäjien odotuksien mukaisesti ja se perustuu ohjelmiston suunnittelun alussa tehtyyn vaatimusanalyysiin. Hyväksymistestaus on itsenäistä verrattuna aikaisempiin testausvaiheisiin, ja sen tarkoituksena on testata, noudattaako ohjelmisto asiakkaan vaatimuksia. Hyväksymistestauksen suorittavat yleensä ohjelmiston tilannut asiakas tai vastaavasti pieni ryhmä ohjelmiston loppukäyttäjiä, tosin myös ohjelmistoprojektiin kuuluvia testaajia saattaa olla mukana testauksessa.

Hamblingin ja kumppaneiden mukaan tyypilliseen hyväksymistestauksen kuuluvat seuraavat asiat (2007 44 – 46):

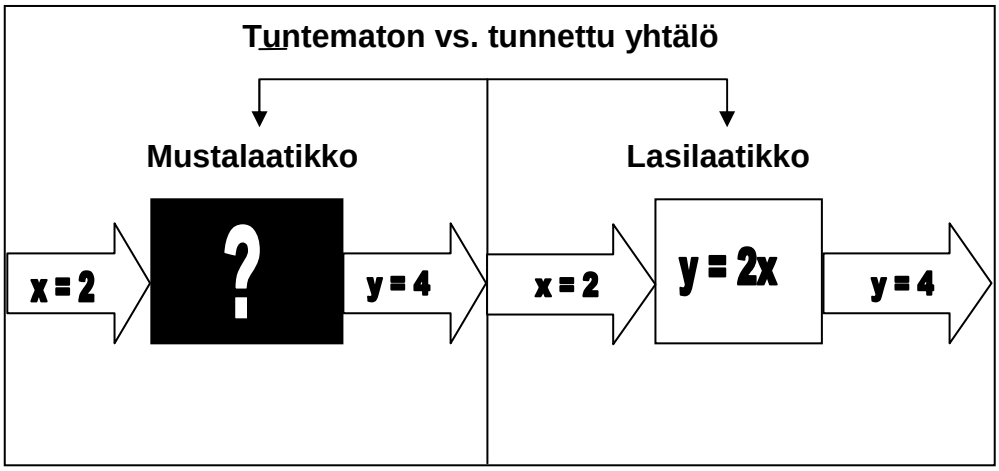
- Käyttäjien hyväksymistestaus, joka suoritetaan loppukäyttäjien toimesta. Näin varmistetaan, että ohjelmisto täyttää heidän yrityksensä tarpeet. Käyttäjien hyväksymistestaus voidaan suorittaa sekä ohjelmistokehitysprojektin tekemässä ympäristössä että lopullisessa käyttöympäristössä.
- Toimintakuntoisuuden testaus, johon kuuluu ohjelmiston prosessien ja toimenpiteiden toimivuuden testaus, jossa varmistetaan, että ohjelmistoa voidaan käyttää ja ylläpitää. Tällaisia toimenpiteitä ja prosesseja ovat varmuuskopiointi, toiminnot onnettomuudesta tms. selviytymiseksi, koulutus loppukäyttäjille, ylläpidon toiminnot sekä turvallisuusprosessit.
- Sopimus ja säännösten hyväksymistestaus. Joskus ohjelmistoprojektin sopimuksessa on määritetty tietyt kriteerit, joiden täyttyminen testataan, ennen kuin ohjelmisto hyväksytään käyttöön. Samoin joissakin ohjelmistoissa vaaditaan tiettyjen valtion määräämien, laillisuus tai turvallisuussäännösten toteutumista. Tällaisia ohjelmistoja ovat esimerkiksi puolustusvoimien, pankkien tai lääketeollisuuden ohjelmistot.
- Alfa- ja betatestaus. Alfatestaus suoritetaan ohjelmistoprojektin tiloissa ohjelmistokehitysprojektin työntekijöiden toimesta, kun taas betatestaus suoritetaan asiakkaan tiloissa asiakkaan ja loppukäyttäjien toimesta ennen ohjelmiston lopullista käyttöönottoa.

Vesiputousmallin heikkoutena pidetään sitä, että se on yleensä aikataulu- ja budjettiriippuvainen. Testaus on se projektin elämänkaaren osa-alue, jota helposti vähennetään resurssien säästämiseksi, jos aikataulusuunnitelmat eivät toteudu alkuperäisen suunnitelman mukaisesti ja projektin rahoitus loppuu kesken. Kuten edellä mainittu V-mallin mukainen käytännön testaus tapahtuu käänteisessä järjestyksessä suunnitteluun nähden, joten rahoituksen loppuessa kesken on olemassa vaara, että ylemmän vaiheen testaustasot jäävät kunnolla suorittamatta. V-mallissa ylemmän tason testausta ei taas tehdä, ennen kuin alemman tason testaus on suoritettu loppuun, joten pahimmassa tapauksessa esimerkiksi hyväksymistestaus jää pintapuoliseksi. (Black 2007, 24.)

2.3 Musta- ja lasilaatikkotestaus

Musta- ja lasilaatikkotestaus ovat kaksi eri testaustyyppiä, joita käytetään ohjelmistotestauksen eri vaiheissa. On olemassa myös paljon muita testaustyyppejä, mutta nämä kaksi ovat käytännössä aina tavalla tai toisella mukana ohjelmistotestauksessa. Lyhyesti mustalaatikkotestaus on testausta, jossa testataan ohjelmiston toimivuus perehtymättä sen tarkemmin ohjelmiston sisältämään koodiin, kun taas lasilaatikkotestauksessa testataan itse koodin toimivuutta ilman, että panostetaan koko ohjelmiston toimivuuteen kokonaisuutena. Mustalaatikkotestauksessa siis testataan suuria ohjelmiston kokonaisuuksia ja koko ohjelmistoa, kun taas lasilaatikkotestauksessa testataan pieniä ohjelmiston osia.

Kuviossa 4 on kuvattu molemmat testaustyypit. Mustalaatikkotestauksessa ei tiedetä mitä itse ohjelmiston koodi tekee, mutta vaatimusanalyysin ja toiminnallisen määrittelyn perusteella tiedetään mitä testitapauksen suorituksen jälkeen ohjelmiston pitäisi tehdä. Lasilaatikkotestauksessa taas yleensä testaaja tietää myös mitä ohjelmiston testauksen alla oleva osa tekee koodin tasolla, koska on yleensä myös itse tehnyt testattavat moduulit. Lasilaatikkotestauksessa löytyneet virheet ovat kieltämättä helpompia korjata, koska virheen löytäjä myös tietää yleensä, mikä sen on aiheuttanut. Lasilaatikkotestausta ei kuitenkaan voida käyttää suurien ohjelmistokokonaisuuksien testaukseen, koska kukaan ei voi sisäistää suurien ohjelmistojen koko koodia. Samoin myös eri komponenttien rajapintojen yhteensopivuusongelmat ovat hankalaa löydettävää suurissa ohjelmiston kokonaisuuksissa, vaikka testaaja tuntisikin ohjelmiston hyvin myös koodin tasolla.



KUVIO 4. Musta- vs. lasilaatikkotestaus

Hamblingin ja kumppaneiden (2007) mukaan tärkein mustalaatikkotestauksen ominaisuus on se, että siinä ei oteta kantaa, miten haluttuun tulokseen päästään koodin tasolla tai miten ohjelmisto sisäisesti suorittaa annetut komennot. Mustalaatikkotestauksessa tärkeintä on testitapauksien suorittaminen ja näistä saatavien tuloksien yhdenmukaisuus vaatimusanalyysin ja toiminnallisen määrittelyn yhteydessä tehtyihin hyväksymistestauksen ja järjestelmätestauksen määriteltyihin tavoitteisiin. On tärkeää erottaa määritys siitä mitä ohjelmiston pitäisi tehdä ja siitä miten se sen tekee. Tämä mahdollistaa sen, että ohjelmistokehitysprojektin kaksi eri ryhmää – testaajat ja ohjelmoijat – voivat toimia itsenäisesti työssään.

Testaajat varmistavat mustalaatikkotestauksella, että ohjelmisto tekee sen, mitä on määritelty ja ohjelmoijat lasilaatikkotestauksella sen, että ohjelmisto tekee asiat niin kuin ne on määritelty. Näillä kahdella eri testaustyypillä pitäisi lopulta päätyä samoihin tuloksiin, vaikka testaustyylit eroavatkin toisistaan paljon. Jos ohjelmoijat ovat ymmärtäneet väärin ohjelmistolta vaadittavat ominaisuudet tai toiminnallisuudet, virheet löytyvät viimeistään mustalaatikkotestauksessa, koska siinä tuloksia verrataan vain ja ainoastaan dokumentoituihin määrityksiin ohjelmiston ominaisuuksista ja toiminnallisuuksista. (Hambling ym. 2007, 78 - 79.)

Lasilaatikkotestausmenetelmää käytetään ohjelmiston rakenteen testaukseen. Sen sijaan, että ohjelmistoa ajetaan ja verrataan saatujen testitapauksien tuloksia määriteltyihin tuloksiin, testataan, että tietyt osat itse ohjelmistosta tule-

vat ajetuksi ja toimivat niin kuin teknisessä määrittelyssä ja moduulisuunnittelussa on määritetty. Tällä testaustyylillä käydään läpi kaikki koodin osat varmistaen, että jokainen lauseke koodissa suoritetaan edes kerran ja että ne toimivat, niin kuin pitääkin. Lasilaatikkotestauksessa on helppo unohtaa suuren ohjelmiston toimivuus kokonaisuutena, mutta näin varmistetaan, että koodi toimii sisäisesti oikein, eikä mitään ylimääräisiä osia jää turhaan mukaan ohjelmistoon. Lasilaatikkotestauksessa testitapaukset määritellään suoraan koodin perusteella, joten lasilaatikkotestauksessa on osattava myös lukea ja analysoida koodia. (Mts. 93.)

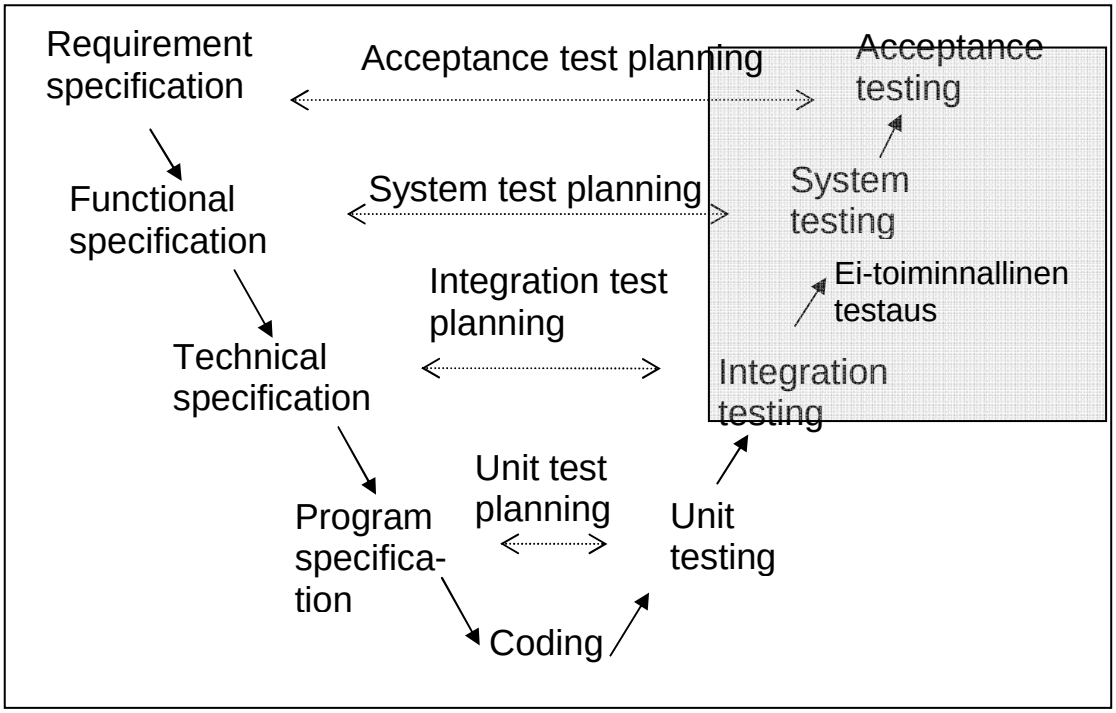
2.4 Toiminnallinen testaus

Testauksen osa-alueena funktionaalinen eli toiminnallinen testaus kuuluu järjestelmätestauksen alle ja sen tarkoitus pähkinäkuoressa on vastata kysymykseen, toimiiko järjestelmä. Toisin sanoen sen tehtävänä on varmistaa toimiiko ohjelmisto sille asetettujen vaatimusten mukaisesti. Toiminnallisessa testauksessa jokainen toiminto testataan yksitellen läpi, jolloin saadaan kattavat tulokset testattavien ominaisuuksien toimivuudesta. Toisaalta funktionaalisen testauksen yhteydessä toiminnallisuuksien riippuvuussuhteista aiheutuvat virheet saattavat jäädä löytymättä. Esimerkiksi toiminnallisessa testauksessa voidaan testata tekstiviestin lähetystä ja Internet-selailua erillisinä toimintoina, mutta pystyykö tekstiviestiä lähettämään samaan aikaan, kun selataan Internet-sivuja, jää testaamatta.(Pääkkönen 2005, 21.)

2.5 Ei-toiminnallinen testaus

Ei funktionaalisen eli ei-toiminnallisen testauksen tavoitteena on testata, miten hyvin järjestelmä toimii. Ei-toiminnallinen testaus ei siis ota kantaa suoranaisesti tiettyjen ohjelmiston ominaisuuksien ja prosessien toimivuuteen, vaan testauksen tarkoitus on selvittää, miten hyvin ohjelmiston toiminnallisuudet ja prosessit toimivat. Ei-toiminnallisen testauksen vaatimukset ovat tyypiltään melko yleisiä ja niitä voidaan soveltaa hyvin erilaisiin ohjelmistoihin. Ei-toiminnallinen testaus esimerkiksi testaa, vastaako ohjelmisto syötteisiin järkevässä ajassa tai voiko tiettyä toimintaa suorittaa useita kymmeniä kertoja

ohjelmiston hajoamatta. Ei-toiminnalliseen testaukseen kuuluu myös sekä ohjelmiston normaalien toimintojen testaus että virhetilanteista palautumisen testaus. Kuviossa 5 on kuvattu ei-toiminnallisen testauksen yleinen sijainti V-mallin tasoissa.



KUVIO 5. Ei toiminnallinen testaus V-mallissa

Yleisimpiä ei-toiminnallisen testauksen osa-alueita ovat seuraavat (Hambling ym. 2007, 44):

- Asennusprosessit – miten sujuvasti ja oikein ohjelmiston asennus onnistuu sekä pystyykö ohjelmistoa käyttämään määritetyllä tavalla asennuksen jälkeen.
- Ohjelmiston toimivuus eri ympäristöissä – toimiiko ohjelmisto kaikissa vaatimusanalyysissä määritetyissä ympäristöissä.
- Ylläpidettävyys – miten hyvin ohjelmisto selviää järjestelmän muutoksista.
- Suorituskyky – toimiiko ohjelmisto odotetusti käyttäen toimintojen suorittamiseen hyväksyttävissä olevan ajan.
- Kuormitus – miten ohjelmisto toimii kuormituksen kasvaessa jatkuvasti.

- Rasituskestävyys – miten ohjelmisto toimii, kun lähestytään järjestelmän kapasiteetin ylärajoja.
- Siirrettävyys – miten ohjelmisto toimii erilaisissa käyttöjärjestelmissä.
- Vikatiloista palautuminen – miten ohjelmisto palautuu virhetiloista.
- Luotettavuus – miten ohjelmisto suoriutuu toiminnallisuuksistaan ja prosesseistaan ajan kuluessa.
- Käytettävyys – helppous, jolla käyttäjä pystyy käyttämään ohjelmistoa.

3 AUTOMATISOITU TESTAUS

3.1 Yleisesti

Kuten aikaisemmin mainittiin, ohjelmistot ovat nykyisin niin laajoja kokonaisuuksia, että kaiken kattava täydellinen testaus on käytännössä mahdotonta saavuttaa. Lin ja Wun (2004, 2 – 9) mukaan testaukseen kuluu ohjelmistoprojektin budjetista 25 - 50 prosenttia ja siihen sisältyy yleensä sekä manuaalista testausta tekevät testaajat, automaattista testausta käyttävät testaajat että testaustyökalujen kehittäjät. Tästä huolimatta on laskettu, että julkaisun jälkeen ohjelmistoissa löytyneet virheet maksavat Yhdysvaltojen taloudelle 59,5 biljoonaa dollaria vuodessa.

Testauksen kattavuuden kasvattamiseksi onkin otettu käyttöön aina useammin automatisoitu testaus, jolla voidaan testata suurempia testitapausmassoja pidempiä aikoja, kuin mitä on mahdollista suorittaa manuaalisesti ihmisvoimin. Testausta ei kuitenkaan ole mahdollista suorittaa kokonaan automaattisesti, vaan testauksen suunnittelu ja itse testauksen automatisointi skriptaamalla sekä testaustuloksien analysointi ja tarkistus on aina pakko tehdä ihmisvoimin. Tähän taas kuluu aikaa ja resursseja, jolloin ohjelmistojen testauksessa on hyvin tärkeää suunnitella etukäteen, mitkä osiot ja alueet testauksesta edes kannattaa suorittaa automaattisesti.

Testaus kannattaa automatisoida, jos samoja testitapauksia joudutaan suorittamaan useita kertoja usealle ohjelmistoversiolle, kuten esimerkiksi tehdään

kestävyystestauksessa. Vaikka yhden skriptin kirjoittamiseen saattaa kulua melkein yhtä paljon aikaa kuin esimerkiksi saman testitapauksen suorittamiseen manuaalisesti viisikymmentä kertaa, on pitkällä aikavälillä taloudellisesti kannattavampaa automatisoida kyseinen testitapaus. Saman testitapauksen skriptiä voi käyttää useita kertoja uudelleen myös tulevien ohjelmistoversioiden testauksessa, jolloin manuaalisiin iteraatioihin kuluva työtä ei tarvitse tehdä ja aikaa ja resursseja jää muuhun työhön. Toki ohjelmistoversioiden kehityksen aikana tulleet muutokset ohjelmiston ominaisuuksiin ja toimintoihin vaativat joskus isojakin muutoksia myös tehtyihin skripteihin, mutta käytännössä tähän kuluva aika on huomattavasti pienempi, kuin jos kaikki testaus tehtäisiin manuaalisesti.

Inhimillisesti on myös mahdotonta jo ajallisesti suorittaa useita satoja ja jopa tuhansia iteraatioita yhtäjaksoisesti ilman mitään taukoja. Automaattisesti tämä onnistuu helposti eikä testaajaan tarvitse kuin varmistaa skriptien tarkoituksen mukainen toimivuus, laittaa automatisoidut testitapaukset ajoon ja analysoida tulokset automatisoidun testauksen ajon loputtua. Ensimmäiseen testausajoon kuluu kieltämättä aina paljon aikaa johtuen skriptien kirjoittamisesta ja niiden toimivuuden varmistamisesta, mutta kokonaisuuteen nähden seuraaviin automatisoituihin testiajoihin kokonaisuuksina kuluu huomattavasti vähemmän aikaa, kuin jos koko toteutus tehtäisiin manuaalisesti. Tällä on suuri vaikutus myös onnistuneisiin tiukoissa aikatauluissa pysymisiin, sillä automaation ei tarvitse nukkua tai toimia vain 7,5h arkipäivisin. Automatisoitu testaus myös poistaa manuaalisessa testauksessa esiintyvät inhimilliset virheet kuten virhepainallukset.

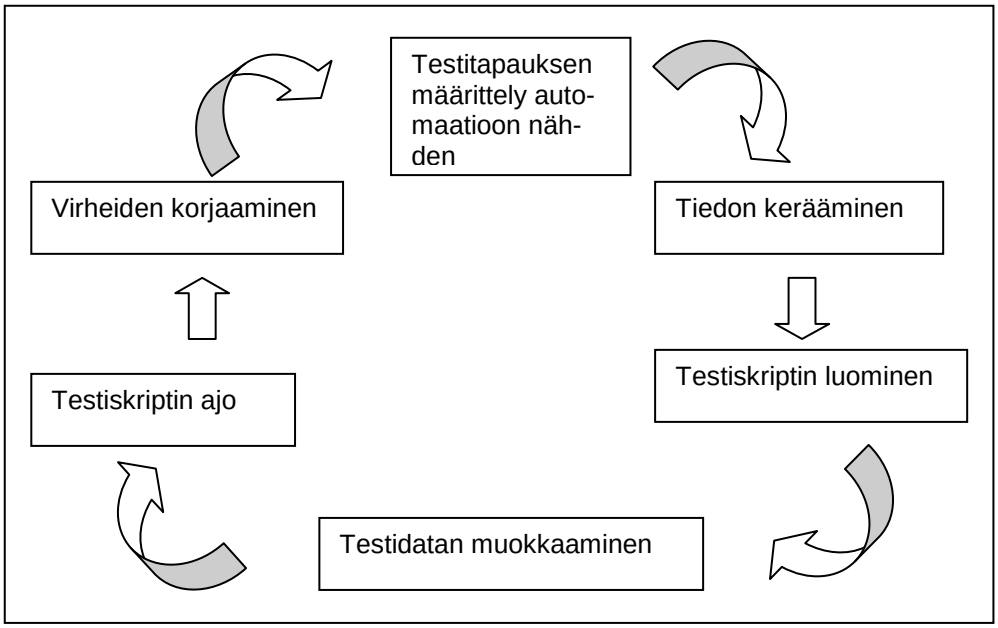
Automatisoitu testaus ei kuitenkaan ole vain laakereilla lepäämistä vaan siinäkin on omat haittapuolensa. Li ja Wu ovat listanneet suurimpia automatisoidun testauksen ongelmakohtia (mts. 4):

- Testitapauksien skriptit vaativat usein uudelleen testausta, sillä pienetkin ohjelmistojen ominaisuuksien tai toiminnallisuuksien muutokset saattavat aiheuttaa ongelmia automatisoitujen testitapauksien ajossa.

- Mikään automatisoinnin työkalu ei voi kokonaan suorittaa itsenäisesti testausta, vaan testaajaan on pakko tehdä osa työstä, kuten suunnittelu, verifiointi ja analysointi manuaalisesti.
- Automatisoinnin työkalut saattavat lisätä testaukseen prosesseja, jotka eivät ole yhteneväisiä testattavan ohjelmiston kanssa.
- Takaisinmallinnuksen prosessi on eroteltu testiskriptien kirjoittamisesta, jolloin virhetilanteiden syyt saattavat olla hankalia selvittää syvätasoisesti.

3.2 Testitapauksien automatisointi

Ennen testauksen aloitusta on yleensä määritelty tietty testitapausryhmä, joka on tarkoitus suorittaa kokonaan testiajon aikana. Testitapausryhmän automatisoinnin aikana testaaja käy kaikki ryhmän testitapaukset läpi yksitellen seuraten automatisoidun testauksen kuutta askelta, jotka näkyvät kuviossa 6.



KUVIO 6. Automatisoidun testauksen kuusi askelta Li ja Wu (2004).

Käytännössä testaaja valitsee jonkun testitapauksen ryhmästä, implementoi määritetyt toiminnallisuudet tarkemmin automaatioon nähden ja vertaa testitapauksen toteutusmahdollisuuksia itse ohjelmiston toiminnallisuuksiin nähden. Joitakin muutoksia joudutaan yleensä tekemään, sillä testitapaukset ovat

usein määritelty yleisellä tasolla johtuen jo ohjelmiston tarkkojen toiminnallisuuksien jatkuvasta muuttumisesta kehityksen aikana. Esimerkiksi jo sovellusten nimet saattavat muuttua ohjelmistokokonaisuuksien eri versioissa.

Tämän jälkeen alkaa testiskriptin kirjoittaminen ja tarvittaessa testidatan muokkaaminen. Testiskriptin tarkoituksen mukainen toimiminen verifioidaan jatkuvasti ajamalla skriptin se osuus, joka on saatu kirjoitettua. Virheet korjataan sitä mukaan, kun ne tulevat esille. Kaikki kuusi askelta käydään mahdollisesti monta kertaa läpi, ennen kuin oikein toimiva testiskripti on valmistunut ja näin toimitaan testitapausryhmän jokaisen testitapauksen kohdalla.

3.3 Automatisoitu älypuhelintestaus

Opinnäytetyö tehtiin asiakkaan omalla älypuhelintestauksen automatisointityökalulla, josta ei valitettavasti voi kertoa yksityiskohtia Non-disclosure agreement – eli salassapitosopimuksen – perusteella.
Tässä kappaleessa esitellään Digian oma älypuhelintestauksen automatisointityökalu AppTest. Molempien työkalujen toimintaperiaatteet ovat hyvin pitkälti samoja, joten AppTestin kuvaus kattaa hyvin pitkälti myös opinnäytetyön tekemisessä käytetyn ohjelmiston toiminnot. Suurin ero näiden kahden ohjelman välillä on se, ettei AppTest:ssä ei ole mukaan rakennettua tekstintunnistusta, mikä taas parantaa huomattavasti tehtyjen skriptien uudelleenkäytettävyyttä.

AppTest on Digian kehittämä testauksen automatisointityökalu ja se kuuluu Digian Quality Kit pakettiin. Quality Kit paketti on tarkoitettu helpottamaan Symbian käyttöjärjestelmien parissa, varsinkin Series 60 tuoteperheen kanssa, töitä tekevien projektipäälliköiden, ohjelmistokehittäjien ja ohjelmistotestaajien työtä. Quality Kit tukee ohjelmistotestauksen kaikkia vaihteita aina moduulitestauksesta järjestelmätestaukseen asti. (Digia 2009, b.)

AppTestin lisäksi Quality Kit pakettiin kuuluvat myös EUnit-työkalu, joka on suunniteltua moduuli- ja integrointitestauksen automatisointiin. Quality Kitin

pakettiin kuuluva kolmas työkalu on TestManager, joka on tarkoitettua testauksen hallintaan ja suoritukseen projektinhallinnan yhteydessä.

AppTest on mustalaatikkotestaukseen tarkoitettua automatisointityökalu ja sillä voidaan automatisoida ohjelmistotestauksessa olevat rutiininomaiset ja useasti toistettavat testitapaukset. Kaikkia testitapauksia ei voida automatisoida, koska osassa tarvitaan esimerkiksi testaajan fyysistä panosta kuten muun muassa kuulokkeiden kaapelin kytkemistä testattavaan puhelimeen.

AppTestin käyttöönotto vaatii ohjelmiston asentamisen tietokoneelle ja AppTestin agentti asennetaan testattavaan puhelimeen. AppTestilla on toimiakseen joitakin laitevaatimuksia eli PC:n käyttöjärjestelmän tulee olla Windows 2000 tai uudempi sekä PC:hen tulee myös olla asennettuna Symbian Series 60 software development kit (SDK). Muita laitevaatimuksia ei AppTestin käyttöön ole, koska ohjelmiston toiminta sujuu käyttöliittymän ja puhelimeen asennetun agentin välillä. (Pääkkönen, 2005, 43 – 44.)

AppTestin skriptit voidaan luoda joko testattavassa puhelimessa olevan agentin kautta tai emulaattorissa. Skriptit luodaan AppTestilla nauhoittamalla kaikki testaajan tekemät toiminnot puhelimessa, minkä jälkeen AppTestin agentti generoi niistä skriptin. AppTestin skriptit voidaan tehdä myös PC:n puolella olevan emulaattorin avulla, jolloin testaaja painelee emulaattorin imitoiman puhelimen kuvassa olevia näppäimiä suorittaakseen testitapauksen, josta halutaan skripti. Agentin kautta tehdyt skriptit voidaan lähettää PC:lle lähes kaikkia Series 60 älypuhelimissa olevia kommunikaatiokeinoja käyttämällä, eli esim. Bluetoothin, USB:n, sähköpostin tai MMS:n välityksellä. (Mts. 44.)

AppTest generoi skriptit puhtaaseen XML-koodiin, mutta niitä voidaan myös siirtää Exceliin, mikä helpottaa skriptin lukemista. AppTest myös tallentaa skriptin tekijän pitämät tauot toimintojen välillä, eikä niitä siten tarvitse erikseen määritellä tehtyihin skripteihin. Luotuja skriptejä joudutaan kuitenkin välillä muokkaamaan testattavan ohjelmiston muutoksien myötä tai esimerkiksi skriptin tekijän virhepainalluksen seurauksena, mikä taas on mahdollista missä tahansa tekstieditorissa. AppTestin skriptien muoto on melko selkeää, eikä ohjelmoinnin syvällistä taitoa tarvita niiden muokkaukseen. (Mts. 45.)

AppTest toimii kuvantunnistuksen perusteella, eli skriptiä luodessa se ottaa kuvakaappauksen tilanteesta ja vertaa olemassa olevia kuvakaappauksia puhelimesta otettaviin kuvakaappauksiin testin suorituksen aikana. Jos skriptiin määritettyä kuvakaappausta ei löydy testin suorituksessa siinä vaiheessa kun sen pitäisi näkyä, testitapauksen suoritus on epäonnistunut.
Useita kertoja suoritettavat testitapaukset toimivat skripteissa samalla periaatteella, paitsi että skriptin loppuun lisätään kuinka monta kertaa se on suoritettava. AppTest on myös suunniteltu tunnistamaan virhetilanteita testiajon suorituksen aikana ja se osaa esimerkiksi erottaa tilanteen, jossa puhelin on nollannut itsensä käynnistymällä uudelleen jonkun virhetilanteen seurauksena. AppTest osaa myös emuloida tilannetta, jossa puhelimen muisti loppuu, eli tällä ominaisuudella voidaan testata miten puhelin käyttäytyy ja miten se toipuu tilanteissa, jossa muisti loppuu kesken yllättäen.

Kuten skriptit myös testiajoista tulevat lokit generoidaan XML-muotoon. Myös lokit pystytään siirtämään Exceliin, mikä lisää lokien luettavuutta ja helpottaa testituloksien raportointia. AppTest perustuu kuvatunnistukseen, joten lokeissa näkyvät myös testiajossa talteen otetut kuvat puhelimen näytöltä, mikä taas helpottaa automaattisesti ajetun testauksen oikeellisuuden varmistamista sekä auttaa mahdollisten virhetilanteiden syiden selvittämisessä. (Mts. 44 – 46.)

4 ÄLYPUHELINTESTAUS

4.1 Toimeksiantajan tekemä ei-toiminnallinen testaus

Toimeksiantajan ei-toiminnallinen älypuhelinten testaus on jakautunut lähinnä automaattisesti ajettavaan kestävyystestaukseen sekä manuaalisesti tehtävään suorituskyvyn testaukseen. Kestävyystestauksen ja suorituskyvyn testauksen lisäksi testataan myös prototyyppilaitteiden toipumiskykyä esimerkiksi irrottamalla akku kesken puhelun tai Internetin selailun ja varmistetaan siten, että puhelin toipuu yllättävistä virhetilanteista ja palautuu täysin toimintakykyiseksi. Yksi ei-toiminnallisen testauksen osa-alue on myös rasitustestaus, jossa testataan puhelimen toimintakykyä useiden samaan aikaan toiminnassa olevien ohjelmistojen rasittaessa puhelimen prosessoria. Esimerkiksi videon katselu vie hyvin paljon puhelimen rajallisia tehoja ja jos saman aikaan myös Internet-selain on auki sekä useita muita ohjelmistoja kuten kalenteri, laskin ja niin edelleen, testattavan laitteen suorituskyky voi olla hyvinkin alhainen.

Seuraavaksi hieman tarkemmin molemmista testauksen osa-alueista, joita käytettiin opinnäytetyön tekemisessä.

4.2 Kestävyystestaus

Automaattisesti ajettavia kestävyystestausta on kahta eri tyyppiä, jotka ovat pitkäkestoinen ja iteraatiopohjainen testaus. Pitkäkestoisessa testauksessa suoritetaan nimensä mukaisesti samaa testitapausta useita tunteja ja tarkoitus on varmistaa prototyyppilaitteen tietyn toiminnallisuuden toimivuus pitkänkin yhtäjaksoisen käytön jälkeen. Esimerkiksi kuusi tuntia kestävä Internet-sivujen selailu WLAN-yhteyden yli on hyvä esimerkki pitkäkestoisesta testitapauksesta. Ymmärrettävistä syistä pitkäkestoista testausta ei voida suorittaa käsin vaan testitapaukset on voitava automatisoida. Manuaalinen testaus olisi resurssien tuhlausta, koska itse skriptin kirjoittamiseen ja ympäristön asentamiseen menisi huomattavasti vähemmän aikaa kuin esimerkiksi kuusi tuntia.

Useamman tunnin kestävä saman yksinkertaisen toiminnan toisto – esimerkiksi sen varmistaminen, että äänipuhelu on toiminnassa ja ääni kulkee puhelinten välillä – olisi myös testin suorittajan näkökulmasta älyllisesti haasteetonta sekä turhauttavaa.

Iteraatiopohjainen kestävyystestaus jakautuu edelleen kahteen osa-alueeseen, joissa toisessa testataan kestävyyttä suorittamalla tietty iteraatiomäärä peräkkäin samaa testitapausta ja toisessa tietyn ajan verran useita testitapauksia satunnaisessa järjestyksessä. Molemmissa iteraatiopohjaisissa kestävyystestauksen tyypeissä on tarkoitus suurella määrällä iteraatioita testata, että puhelimen ohjelmisto toimii oikein myös pidemmän käytön jälkeen. Molemmissa tapauksissa testaaja kirjoittaa ensiksi skriptit joiden mukaan testitapaukset ajetaan automaattisesti. Skriptien kirjoittaminen ja sen varmistaminen, että tehdyt skriptit myös toimivat määrittelyn mukaisesti, on automatisoidun testauksen aikaa vievin osa. Toki tuloksien raportoiminen ja testitapauksien ajon aikainen seuraaminen vievät oman osansa työajasta, mutta käytännössä skriptien verifioiminen on kaikkein hankalin ja aikaa vievin osa.

Ensimmäisessä tyypissä ennalta määritellyt testitapauksista ajetaan jokainen yksitellen automaattisesti useita iteraatiota, joiden määrä vaihtelee yleensä kolmenkymmenen ja sadan välillä. Testitapaus voi olla esimerkiksi videon avaaminen prototyyppilaitteesta ja tekstiviestin lähetys videon katselun aikana ja tämä suoritetaan sitten automaattisesti esimerkiksi viisikymmentä kertaa peräkkäin.
Kaikkia testitapauksia ei voida automatisoida johtuen automaation rajoituksista, sillä esimerkiksi kuulokkeiden kytkeminen ja irrottaminen automaattisesti eivät ole vielä mahdollisia. Nämä testitapaukset on pakko suorittaa suuresta iteraatiomäärästä huolimatta manuaalisesti.

Toisessa automatisoidussa kestävyystestauksen tyypissä muutoin samat periaatteet, paitsi että tietty testitapausten massa ajetaan satunnaisessa järjestyksessä esimerkiksi noin viikon ajan yhtäjaksoisesti. Tämä on prototyyppilaitteen kannalta kaikkein raskain testaustyyli ja yleensä se tuleekin mukaan vasta ohjelmistokehityksen loppuvaiheessa.

4.3 Suorituskyvyn testaus

Yleensä manuaalisesti suoritettavassa suorituskyvyn testauksessa mitataan muun muassa eri tapahtumien kestoa ajallisesti, eli esimerkiksi kuinka paljon kuluu aikaa selaimen avautumiseen. Suorituskyvyn testauksessa mitataan myös muun muassa virran kulutusta eri tilanteissa ja datansiirtonopeuksia. Suorituskyvyn testauksen osa-alueita ovat myös vasteajan mittaus ja tiedonsiirtonopeuden mittaus.

Suorituskyvyn testauksen osa-alueille on ominaista tarkkaan määritelty testidata sekä testattavan laitteen alkutila asetuksineen. Nämä molemmat on hyvin tärkeää säilyttää samoina läpi testattavan laitteen testauselinkaaren, sillä vähäpätöisimmiltäkin vaikuttavat muutokset saattavat vaikuttaa testituloksiin ja näin saattaa sekä tuloksien oikeellisuuden että vertailukelpoisuuden kyseenalaiseksi. Esimerkiksi prototyyppilaitteen muistikortilla olevan testidatan määrä voi vaikuttaa paljonkin siihen, miten paljon aikaa kuluu tietyn tiedoston siirtämiseen tietokoneelta testattavaan laitteeseen. Tyhjään muistiin voi siirtää huomattavasti nopeammin tiedostoja, sillä laitteen ei tarvitse käydä koko muistia läpi löytääkseen riittävästi tyhjää tilaa. Lähes täyteen muistiin siirrättäessä laite taas saattaa joutua käymään koko muistin läpi ennen kuin löytää riittävästi tyhjää tilaa. Tämä taas vaikuttaa hyvin paljon tiedoston siirtoon kuluvaan aikaan. Tarkkaan määritelty testidata ja prototyyppilaitteen alkutila tietyin määritellyin asetuksin myös lisää suorituskyvyn testauksen validia toistettavuutta sillä näin vähennetään riskiä muiden ennalta määrittämättömien asioiden – kuin testattavan laitteen suorituskyvyn – vaikutuksesta mittaustuloksiin.

Suorituskyvyn testauksen osa-alueille on myös ominaista tarkkaan määritellyt mittauksen aloitus ja lopetuspisteet. Kaikille testitapauksille on määritetty omat raja-arvonsa, joista riippuen testitapaus joko hyväksytään läpimenneeksi tai sitten hylätyksi, jos mittaus tulokset eivät jää raja-arvojen väliin. Raja-arvot ovat yleensä hyvin pieniä lukuja ja tästä johtuen testaajan on tarkkaan tiedettävä missä kohtaa mittaus on aloitettava ja mihin lopetettava, sillä parin sekunnin viive voi johtaa testitapauksen tuloksen hylätyksi tulemiseen.

Vasteajan mittauksessa mitataan kuinka pitkä aika tietyn ohjelman tai sen toiminnon käynnistykseen kuluu. Esimerkiksi kuinka paljon aikaa kuluu siitä kun painetaan kalenteriohjelman ikonia siihen kunnes kalenterin avausnäkymä on kokonaan piirtynyt alavalikon kuvakkeita myöten laitteen ruudulle. Yleisimmät keinot mitata kuluvaa aikaa ovat sekuntikello tai sitten tapahtuma kuvataan videokameralla ja tallennettu video käydään kuva kuvalta läpi siihen asti kunnes haluttu näkymä on kokonaisuudessaan piirtynyt näytölle. Vasteajan mittauksia tehdään sekä käyttöliittymän että komentoliittymän kautta. Edellä mainittu esimerkki kalenterista tehdään käyttöliittymän kautta, mutta komentoliittymän testauksessa taas mitataan ohjelmiston sisäisiin toimintoihin kuluvaa aikaa. Esimerkkinä tästä kuinka kauan kuluu aikaa lisätyn kalenterimerkinnän tallentamiseen laitteen tietokantaan. Komentoliittymän mittaukset tehdään koodin instrumentoinnilla, eli koodiin lisätään apuohjelmaa käyttäen pisteitä, ja mitataan aikaa, joka kuluu pisteiden välillä olevan koodin suorittamiseen.

Toinen suorituskyvyn testauksen osa-alue on tiedon siirtonopeuden mittaus. Tämäkin voidaan testata sekä käyttöliittymän että komentoliittymän tasolla samoilla tavoilla kuin mitä vasteajan mittauksessa käytetään. Siirtonopeuden mittauksessa mitataan kuinka kauan tiedoston siirtoon menee eri yhteystavoilla tai prototyyppilaitteen sisäisten muistien sekä muistikortin välillä. Kun tietyn tiedoston koko ja siirtämiseen kuluva aika on tiedossa, voidaan laskea siirtonopeus esimerkiksi bitteinä sekunnissa. Myös siirtonopeuksia mittaaviin testitapauksiin on määritelty omat raja-arvonsa ja vertaamalla saatuja tuloksia näihin arvoihin saadaan joko hyväksytty tai hylätty tulos kyseiseen testitapaukseen.

5 TESTAUSKONSEPTIN TAVOITE JA SUUNNITTELU

5.1 Tavoite

Opinnäytetyön tavoitteena oli kehittää käytännön toteutuksen kautta mustalaatikkotestaustyyliin pohjautuva testauskonsepti prototyyppilaitteiden rautapohjaisten virheiden löytämiseksi. Tarkoitus oli siis suunnittelemalla ja käytännössä kokeilemalla löytää toimiva tapa testata ja erottaa mahdolliset prototyyppilaitteiden rautatoteutuksesta johtuvat virheet testauksen yhteydessä esiin tulevista ohjelmistopohjaisista virheistä. Tällä tavalla virallisten testituloksien paikkaansa pitävyys varmistuisi ja myös testituloksien laatu paranisi. Ohjelmistopohjaisen testauksen tarkoitus on kuitenkin vain löytää ohjelmistossa mahdollisesti olevat ohjelmistopohjaiset virheet eikä kiinnittää huomiota niinkään mahdollisiin rautapohjaisiin virheisiin, joita saattaa esiintyä vain joissakin, vielä kehitysvaiheessa olevissa, prototyyppilaitteissa. Toki myös rautatoteutus testataan, mutta se suoritetaan erikseen ja eri tavalla kuin ohjelmistopuolen testaus, eikä näin ollen ohjelmistotestauksessa edes ole työkaluja tai toimivia käytäntöjä kattavaa rautatoteutuksen testausta varten.

Omat rajoitteensa prototyyppilaitteiden rautapohjaisen testauksen konseptin kehittämiseen aiheuttivat resurssien vähyys sekä testausprojektin rajallinen budjetti. Testausprojektissa on yleensä vain tietty määrä työntekijöitä ja heidän työpanoksensa on laskettu hyvin tarkkaan vain ja ainoastaan ohjelmistopohjaisen testauksen työtehtäviin. Jokaisella projektilla on tietty budjetti, jonka rajoja ei saisi ylittää ja varsinkin alihankintana tehtävien projektien budjetit ovat hyvin tarkkaan määriteltyjä asiakkaan puolelta. Näin ollen ohjelmistotestausprojekteissa ei käytännössä ole juurikaan aikaa ja resursseja ns. ylimääräiseen työhön, eli tässä tapauksessa prototyyppilaitteiden rautapohjaiseen testaukseen. Jokaisen työntekijän työpanos on suunniteltu ohjelmistotestauksen tehtäviin, johon taas ei kuulu rautapohjaista testausta. Tämä loi tiukat rajat siihen miten paljon opinnäytetyönä olevaan automatisoituun prototyyppilaitteiden rautapohjaisen suorituskyvyn testaukseen kuluu aikaa käytännössä. Konsepti olisi ollut hyödytön, jos siihen olisi kulunut useamman päivän verran työ-

tunteja testauksen suorittajalta, sillä projektilla ei olisi ollut resursseja toteuttaa konseptia. Tästä johtuen mahdollisimman helposti ja nopeasti toteuttavissa oleva tapa testata prototyyppilaitteiden rautapohjaista suorituskykyä automatisoidusti oli ehdoton testauskonseptin vaatimus. Automatisoitu rautapohjainen testaus ei saanut myöskään varata liian pitkäksi ajaksi testausympäristöjä, sillä niitä on rajallinen määrä ja viralliset asiakkaan antamat ohjelmistotestauksen työtehtävät ovat tärkeysjärjestykseltään korkeammalla. Jos testausympäristöt loppuisivat kesken, niin rautapohjaisen suorituskyvyn testaus jouduttaisiin keskeyttämään projektin pääasiallisen toimen – ohjelmistotestauksen – tieltä, jolloin rautapohjaisen suorituskyvyn testaus jouduttaisiin aloittamaan myöhemmin alusta tulosten oikeudellisuuden takaamiseksi.

Automatisoidun rautapohjaisen suorituskyvyn testauskonseptin nopean toteutuksen puolesta vaikutti myös se seikka, että testattavaksi tulevat prototyyppilaitteet tarvitaan yleensä myös mahdollisimman nopeasti ohjelmistotestaukseen. Ohjelmistoa ja prototyypin fyysistä rautatoteutusta kehitetään yleensä samanaikaisesti, eikä olisi järkevää testata ohjelmistoa muussa ympäristössä kuin siinä mihin se on alun perin suunniteltu ja jossa se tulee lopulta myös sijaitsemaan. Muutoin saadut tulokset muuttuisivat radikaalisti ympäristön vaihtuessa, sillä ohjelmisto on suunniteltu toimimaan tietyssä rautapohjaisessa ympäristössä. Tämän kokonaisuuden testauksesta saatuihin tuloksiin pohjautuu ohjelmiston jatkokehitys ja jos ympäristö vaihtuisi jostakin syystä, jouduttaisiin tekemään huomattavasti lisää kehitys- ja testaustyötä saadakseen ohjelmiston ja fyysisen rautatoteutuksen muodostama kokonaisuus toimimaan alun perin suunnitellulla tavalla. Tästä johtuen myöskään prototyyppejä ei voi varata ns. ylimääräiseen testaukseen pitkäksi aikaa, sillä kuten aikaisemmin mainittiin, viralliset ohjelmistotestauksen työtehtävät ovat tärkeysjärjestykseltään korkeammat ja virallinen testaus ohittaisi tämän opinnäytetyön kontekstissa suunnitellun rautapohjaisen testauksen.

Testauksen laadun parantaminen on kuitenkin aina tärkeää. Varsinkin tulosten paikkaansa pitävyys ja toistettavuus ovat äärimmäisen tärkeitä ohjelmistotestausprojekteissa. Aikaisemmin virallisen testauksen yhteydessä – varsinkin suorituskyvyntestauksessa – on tullut vastaan prototyyppipuhelimia, joissa on ollut rautapohjainen vika. Tämä ilmeni yleensä sen jälkeen, kun virallista tes-

tausta oli jo tehty useamman työtunnin verran. Testauksen myöhemmässä vaiheessa esiin tullut virhe oli tarkistettu toisella prototyyppipuhelimella, jolloin huomattiin, ettei virhetilannetta toistu muilla prototyypeillä. Tämän jälkeen testattava ohjelmisto asennettiin uudestaan prototyyppiin, ja jos virhetilanne toistui, kyseessä oli rautapohjainen vika. Tästä taas seurasi se, että viallisella prototyypillä testattujen tuloksien paikkansapitävyys oli kyseenalaista ja viallisella prototyypillä tehdyt testitapaukset piti suorittaa uudelleen toimivalla prototyyppipuhelimella. Tähän kului turhaan aikaa ja resursseja, joten oli tärkeää, että prototyyppipuhelinten rautapohjaista suorituskykyä testattaisiin ennen virallisen testauksen aloittamista, sillä se varmistaisi ja parantaisi tuloksien laatua ja paikkansapitävyyttä. Viallisten prototyyppipuhelinten seulominen pois testattavien prototyyppien joukosta taas vähentäisi rautapohjaisista vioista johtuvaa turhaa ajan ja resurssien tuhlausta.

Automatisoidun rautapohjaisen suorituskyvyn testauskonseptista olisi myös se hyöty, että samalla kun automatisoitu rautapohjaisen suorituskyvyn testaus tehtäisiin kaikille uusille prototyyppipuhelimille, saataisiin tuloksien ohella myös yleiskatsaus prototyyppipuhelinten yleisestä rautapohjaisesta kypsyydestä. Tämä taas antaa hieman näkyvyyttä mahdollisiin tuleviin testituloksiin ja näin ollen auttaa suunnittelemaan testausaikatauluja.

Testauskonseptin suurimpia ongelmia oli ohjelmistopohjaisten virheiden ja rautapohjaisten vikojen erottaminen toisistaan. Varsinkin sekä ohjelmiston että prototyyppilaitteiden kehityskaaren alkuvaiheessa virhetilanteita esiintyy hyvin usein. Suurin osa näistä on ohjelmistopohjaisia, mutta testaajan näkökulmasta sekä ohjelmistopohjaiset että rautapohjaiset virhetilanteet näyttävät samalta sillä prototyyppilaite toimii molemmissa samantyyppisesti väärin. Testauskonseptin tavoitteena oli löytää tapa erottaa nämä tilanteet mahdollisimman hyvin toisistaan, jotta oikeasti vialliset laitteet voitaisiin siirtää pois virallisesta ohjelmistotestauksesta, tai riippuen rautapohjaisen vian vaikutuksesta laitteen toimintaan, ainakin pois virallisesta suorituskyvyn testauksesta.

Testauskonseptin vaatimuksina olivat siis mahdollisimman nopeasti ja pienellä työmäärällä toteutettava tapa löytää mahdolliset rautapohjaiset viat prototyyppilaitteista. Myös löytää selkeä tapa erottaa rautapohjaiset viat ohjelmistopoh-

jaisista virhetilanteista oli tärkeä osa testauskonseptia, jonka suurin tavoite oli parantaa testauksen laatua.

5.2 Suunnittelu

Testauskonseptin automaattisesti ajettavat testitapaukset suunniteltiin hyvin yksinkertaisiksi, jotta testiskriptien kirjoittaminen olisi mahdollisimman helppoa ja nopeaa eikä niiden ylläpitäminen ja muokkaaminen vaatisi paljon aikaa tai resursseja. Testauskonseptiin kuluvan työmäärän vähentämiseksi siihen suunniteltiin myös asetusskripti, jolla pystyttiin laittamaan suurin osa automaattiseen testiajoon tarvittavista asetuksista automatisoidusti prototyyppilaitteisiin. Näin prototyyppilaitteet saataisiin mahdollisimman nopeasti ajokuntoon ja myös asetukset olisivat kaikissa identtisiä, mikä parantaisi tulosten luotettavuutta.

Johtuen tarpeesta suorittaa testauskonsepti nopeasti, siihen ei suunniteltu myöskään kahden prototyyppilaitteen keskenäistä vuorovaikutusta sisältäviä testitapauksia, sillä niihin tarvittaviin testausskripteihin tulisi molemmille prototyyppilaitteelle omat skriptinsä ja myös testausympäristön konfigurointiin tarvittava työmäärä kaksinkertaistuisi. Kaksi prototyyppilaitetta vaativa testitapaus olisi esimerkiksi tekstiviestin kirjoittaminen ja lähettäminen yhdestä prototyyppilaitteesta testattavaan prototyyppilaitteeseen, joka ottaa viestin vastaan, avaa sen ja tarkistaa, että kyseessä on juuri lähetetty viesti. Testitapaukset suunniteltiin yksinkertaisiksi myös siksi, että useamman ohjelman käyttö samanaikaisesti yhdessä testitapauksessa olisi raskasta prototyyppilaitteen ohjelmistopuolelle, jolloin vaara ohjelmistopohjaisten virhetilojen esiintymisestä kasvaisi.

Testauskonseptin automatisoituun testijoukkoon suunniteltiin kaksitoista testitapausta, joiden tarkoitus oli testata prototyyppilaitteen rautatoteutuksen eri komponentteja. Kaikki kaksitoista testitapausta on esitelty testattavine komponentteineen taulukossa 2.

TAULUKKO 2. Testitapaukset ja niiden testaamat prototyyppilaitteen komponentit.

	Testitapaus	**Testattava komponentti**
1	Äänipuhelu 2G-verkon yli	Testasi prototyyppilaitteen radiotaajuuskomponenttien toiminnallisuutta äänipuhelun osalta 2G-verkossa.
2	Äänipuhelu 3G-verkon yli	Testasi prototyyppilaitteen radiotaajuuskomponenttien toiminnallisuutta äänipuhelun osalta 3G-verkossa.
3	SMS viesti 2G-verkon yli	Testasi prototyyppilaitteen radiotaajuuskomponenttien toiminnallisuutta tekstiviestin lähetyksen osalta 3G-verkossa.
4	SMS viesti 3G-verkon yli	Testasi prototyyppilaitteen radiotaajuuskomponenttien toiminnallisuutta tekstiviestin lähetyksen osalta 3G-verkossa.
5	Internet-sivujen selailu WLAN-verkon yli	Testasi prototyyppilaitteen WLAN-komponenttien langattoman lähiverkon toiminnallisuutta.
6	Hälytys kelloon	Testasi prototyyppilaitteen käyttöjärjestelmän kello-komponentin toimivuutta. Kello on käyttöjärjestelmien tärkeimpiä toiminnallisuuksia ja siihen perustuu yleensä hyvin paljon muita toiminnallisuuksia.
7	Kuvan ottaminen kameralla	Testasi prototyyppilaitteen kamera-komponentteja.
8	Tiedostoja sisältävän kansion kopioiminen ulkoiselta muistikortilta massamuistiin	Testasi prototyyppilaitteen datansiirron toiminnallisuutta tietojärjestelmän sisällä.
9	Muistinkulutuksen sen hetkisen tilan taltioiminen	Muistinkulutuksen seurantaan suunniteltu ohjelma haki prototyyppilaitteen sen hetkisen muistinkulutuksen ja tallensi sen lokiin.
10	Profiilin vaihto yleisestä äänettömään ja takaisin	Testasi prototyyppilaitteen käyttöjärjestelmän toimivuutta koko järjestelmään vaikuttavien

		asetuksien vaihdon yhteydessä. Ohjelmisto on suunniteltu tietylle rautatoteutukselle, jolloin myös tämän kokonaisuuden toimivuus oli tärkeää testata.
11	Muistikortilla sijaitsevan MP3-musiikkitiedoston toisto	Testasi prototyyppilaitteen äänentoistokomponenttien toimintaa sekä tietojärjestelmän sisäistä datan käyttöä ulkoiselta fyysiseltä muistilta.
12	Kontaktin lisääminen Puhelinluetteloon	Testasi prototyyppilaiteen tietojärjestelmän sisäiseen fyysiseen muistiin tallennuksen toiminnallisuutta.

Automatisoitujen kestävyystestauksen tyyppisten testitapauksien lisäksi konseptiin suunniteltiin myös suorituskykytestauksen tyyppinen testitapaus, jossa mitattiin laitteen käynnistysnopeutta. Testauskonseptiin ei suunniteltu enempää suorituskykytestauksen tyyppisiä testitapauksia, sillä tämän testityypin testitapauksien suoritus on manuaalista ja näin ollen testauskonseptin suorittamiseen kuluva aika ja työmäärä kasvaisivat oleellisesti. Prototyyppilaitteen käynnistysnopeuden mittaaminen aloitettiin virtanapin painamista seuranneesta prototyyppilaitteen värähtelystä aina siihen pisteeseen kun kaikki ikonit ovat piirtyneet laitteen näytölle aloitustilassa. Testauskonseptissa käynnistysnopeus suunniteltiin mitattavaksi jokaisesta prototyyppilaitteesta ensimmäisen kerran heti ohjelmiston asennuksen jälkeen. Toisen kerran käynnistysnopeus mitattaisiin jokaisesta prototyyppilaitteesta sen jälkeen kun automatisointiin tarvittavat asetukset on laitettu laitteeseen. Kolmannen kerran käynnistysnopeus mitattaisiin testiajon jälkeen.

Kuten aikaisemmin on usein mainittu, automatisoidun rautapohjaisen suorituskyvyn testauskonseptin yksi tärkeimpiä vaatimuksia oli mahdollisimman vähän resursseja vievä ja mahdollisimman nopea testauksen toteutus. Tästä johtuen yhden prototyypin testaukseen kuluvaksi ajaksi suunniteltiin noin puoli vuorokautta kestävä ajojakso. Tähän ajokestoon päädyttiin siksi, että testaaja pystyy laittamaan prototyyppilaitteet yön yli testiajoon ja ottamaan tulokset talteen seuraavana aamuna, jolloin sekä testattavat prototyyppilaitteet että

testausympäristöt ovat vapaasti virallisen testauksen käytettävissä jo seuraavana päivänä. Yksi testijoukon testiajo kesti noin neljäkymmentä minuuttia, jolloin samaa testijoukkoa ajettiin yhteensä seitsemäntoista kertaa samassa järjestyksessä. Yhden kokonaisajon kestoksi muodostui silloin keskimäärin runsaat yksitoista tuntia. Sama pysyvä testitapauksien ajojärjestys kaikilla testattavilla prototyyppilaitteilla teki prototyyppilaitteiden ajokestoista pätevät vertailukohteet.
Jos yhdellä prototyyppilaitteella kestäisi huomattavasti pidempään suoriutua samasta määrästä samassa järjestyksessä ajettavista testitapausiteraatioista kuin muilla testattavilla prototyyppilaitteilla, olisi siinä selkeästi jokin lisätutkimusta vaativa vika.

Testauskonseptin suunnittelussa oli testikäytössä seitsemän saman kypsyystason prototyyppilaitetta, jotta testaustilanne ja tulokset vastaisivat mahdollisimman paljon todellista prototyyppilaitteiden rautapohjaisen suorituskyvyn testaustilannetta. Ainut tapa erottaa rautapohjainen vika on testata sama testitapausjoukko usealla muulla prototyyppipuhelimella ja jos tietty virhetilanne toistuu vain ja ainoastaan yhdellä laitteella myös ohjelmiston uudelleen asennuksen jälkeen, voidaan varmuudella sanoa, että kyseessä on rautapohjainen vika. Näin ollen rautapohjaisia vikoja löytyy sitä todennäköisemmin, mitä enemmän testattavia prototyyppilaitteita on. Rautapohjaisen suorituskyvyn testauskonseptin tavoitteena olikin se, että uuden rautatoteutuksen version tullessa kaikki prototyyppilaitteet testattaisiin kerran läpi ennen virallisen ohjelmistotestauksen aloittamista.

Rautapohjaisten vikojen olemassaolo ei kuitenkaan ole itsestäänselvyys, sillä oletusarvoisesti kaikkien prototyyppilaitteiden oletetaan toimivan oikein. Fyysistä rautatoteutusta testataan kuitenkin pitkään ennen ei-toiminnallisen ohjelmistotestauksen aloittamista. Silti on mahdollista, että fyysisten laitteiden valmistusprosessissa on tapahtunut virhe yksittäisten prototyyppilaitteiden kohdalla.

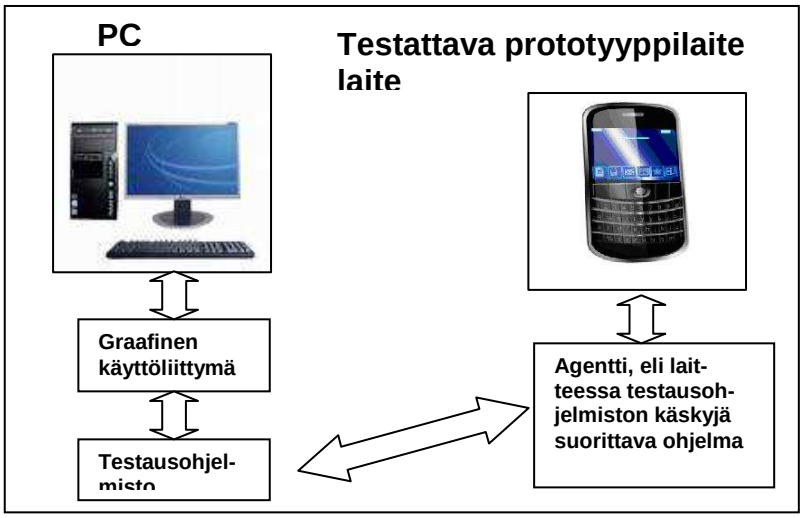
Myös ohjelman asennuksen aikana jokin voi mennä vikaan, jolloin ainut tapa varmistaa virhetilanteen johtuvan ohjelmistosta on asentaa testattavana oleva ohjelmisto uudelleen ja tehdä testitapaus, jossa virhetilanne esiintyi, uudel-

leen. Näin ollen oli tärkeää, että kaikki käytössä olevat prototyyppilaitteet testattaisiin samalla kertaa käyttäen samaa ohjelmistoversiota, jolloin mahdolliset rautapohjaiset viat erottuisivat selkeämmin koko testattavien prototyyppilaitteiden joukosta. Tästä syystä myös testauskonseptissa testauksen aikana yksittäisessä prototyyppilaitteessa tapahtuneen virhetilanteen – jota ei esiintynyt muilla prototyyppilaitteilla – selvittämisen seuraava askel on ohjelmiston uudelleen asentaminen. Sitä kautta virhetilanteen alkuperä selviää joko rautapohjaiseksi tai ohjelmistopohjaiseksi.

5.3 Testausympäristö

Testausohjelmisto koostuu kolmesta komponentista - PC:llä sijaitsevista graafisesta käyttöliittymästä ja moottorista sekä prototyyppilaitteeseen asennettavasta agentista. Näistä käyttäjälle näkyvin osa on graafinen käyttöliittymä, jolla hallitaan koko testausympäristöä. Graafisella käyttöliittymällä valitaan ajettavat skriptit ja sen kautta ne laitetaan myös ajoon. Käyttöliittymästä myös voi nähdä sen hetkisten ajojen tilanteen sekä pystytään myös generoimaan HTML-muotoisen lokin ajon aikaisista tapahtumista.

Graafisen käyttöliittymän alla on komentovalikon tyyppinen testausohjelmiston moottori. Tämä osa ottaa graafiselta käyttöliittymältä vastaan skriptit, muotoilee niissä olevat käskyt testattavalle prototyyppilaitteelle lähetykseen sopiviksi, lähettää ne testattavassa laitteessa olevalle ohjelmistolle, eli agentille, ja kuittaa testattavasta laitteesta takaisin tulevat tulokset jokaisen käskystä. Jokaisen käskyn kuittauksen jälkeen moottori vertaa tullutta vastausta käskyn onnistumiseen vaadittaviin kriteereihin ja riippuen vertailun tuloksesta laittaa käskyn joko onnistuneeksi tai epäonnistuneeksi. Kokenut käyttäjä pystyy muokkaamaan testausohjelmiston moottorin asetuksia, mutta yleensä niitä ei tarvitse muuttaa vaan testaaja pystyy perusasetuksilla käyttämään ohjelmistoja testauksen automatisoinnissa ja testitapauksien suorittamisessa. Kuviossa 7 on esitetty testausympäristön automatisointiohjelmiston osat tietokoneen ja testattavan laitteen välillä.



KUVIO 7. Automatisointiohjelmiston eri osat

Testattava prototyyppilaite voidaan liittää tietokoneeseen useammalla eri tavalla. Yleisemmät näistä ovat USB ja Bluetooth, joista ensiksi mainittua käytetään eniten vakaan toimivuutensa ja helppokäyttöisyytensä ansiosta. Molemmat yhteysmuodot toimivat testausympäristön näkökulmasta samalla periaatteella, eikä yhteystyypillä ole mitään vaikutusta testausohjelmiston toimintaan. Toimiakseen yhteysmuoto pitää valita samaksi sekä testausohjelmiston moottorissa että testattavassa laitteessa olevassa vastakappaleessa.

Käytännössä koko toimiva fyysinen testiympäristö tietokoneineen ja testattavine laitteineen pitää yleensä sisällään seuraavat kolme komponenttia – tietokoneen, yhteysmuodon sekä testattavan prototyyppilaitteen. Tietokone on yhdistetty joko datakaapelia tai BT-yhteyttä pitkin testattavaan laitteeseen, johon on kytketty myös laturi, koska testiajot saattavat kestää useita vuorokausia ja testattavan laitteen akun on kestettävä koko testiajon aika. Tietokoneeseen on yhdistetty myös kamera, jota säätelee testausohjelmisto. Kameralla kuvataan kaikki yksittäiset testitapauksien iteraatiot, mutta vain epäonnistuneiden iteraatioden videomateriaali tallennetaan. Näin ollen testiajon suorittavan testaajan on helpompaa analysoida epäonnistuneiden iteraatioiden taustalla olevia syitä. Testilaboratoriossa on yleensä valmiina useita fyysisiä ympäristöjä, koska usein tulee samanaikaisia pidempiä testiajoja. Kuviossa 8 esitellään perustestiympäristön fyysiset komponentit.

KUVIO 8. Testausympäristö

6 TESTAUSKONSEPTIN TOTEUTUS

6.1 Testauskonseptin toteutuksen pohjatyö

Testattaviksi prototyyppilaitteiksi valittiin seitsemän laitetta, joiden virallinen testaus oli jo loppunut ja sekä rautatoteutuksen että ohjelmiston kehityselinkaari oli loppuvaiheessa. Valinta johtui siitä, että testauskonseptin käytännön toteutus olisi jouduttu keskeyttämään virallisen testaustyötehtävän tieltä, jos testattavina laitteina olisi käytetty vielä virallisessa testauksessa olevia prototyyppilaitteita. Testauskonseptin toteutuksessa käytettiin myös viimeisintä ohjelmistoversiota, joka valittiin vakautensa ja kypsyytensä takia, sillä nämä ominaisuudet vähentäisivät ohjelmistopohjaisten virhetilanteiden esiintymisen todennäköisyyttä.

Testauskonseptin toteutus aloitettiin tekemällä asetusskripti. Asetusskriptilla laitettiin mm. prototyyppilaitteen näytön valojen ja näppäinlukon asetukset sekä määriteltiin myös joitakin testitapauksiin tarvittavia asetuksia kuten yhteysosoitteet ja tekstiviestien lähettämiseen ja vastaanottamiseen tarvittava viestikeskuksen numero. Asennusskriptillä nopeutettaisiin huomattavasti prototyyppilaitteiden testiajovalmiuteen laittoja ja varmistettaisiin testattavana olevien prototyyppilaitteiden asetusten olevan mahdollisimman identtisiä keskenään. Tämä vähentäisi mahdollisuutta, että laitteet käyttäytyisivät keskenään poikkeavasti toisistaan eroavista asetuksista johtuen.

Asetusskriptin jälkeen aloitettiin testauskonseptiin suunniteltujen automatisoitujen testitapauksien (ks. kappale 5.2) skriptaaminen. Kuten aikaisemmin mainittiin, kahden prototyyppilaitteen testitapauksia ei suunniteltu testauskonseptiin vaan viestien lähetykseen ja äänipuheluihin käytettiin testinumeroa, joka vastaanottaa puheluita ja viestejä. Testitapauksiin skriptatut toiminnat on esitelty taulukossa 3.

TAULUKKO 3. Skriptien toiminnot testitapauksittain

	Testitapaus	**Skriptin toiminnat**
1	Äänipuhelu 2G-verkon yli	Skripti laittaa prototyyppilaitteen 2G-verkkoon minkä jälkeen soitetaan testinumeroon ja tarkistaa, että puhelu vastaanotetaan. Odotetaan kaksi minuuttia ja suljetaan puhelu.
2	Äänipuhelu 3G-verkon yli	Skripti laittaa prototyyppilaitteen 3G-verkkoon minkä jälkeen soitetaan testinumeroon ja tarkistaa, että puhelu vastaanotetaan. Odotetaan kaksi minuuttia ja suljetaan puhelu.
3	SMS viesti 2G-verkon yli	Skripti laittaa prototyyppilaitteen 2G-verkkoon, minkä jälkeen avataan Viestit-ohjelman ja kirjoitetaan uusi tekstiviesti. Viesti lähetetään testinumeroon, suljetaan Viestit-ohjelma ja palataan aloitustilaan.
4	SMS viesti 3G-verkon yli	Skripti laittaa prototyyppilaitteen 3G-verkkoon, minkä jälkeen avataan Viestit-ohjelman ja kirjoitetaan uusi tekstiviesti. Viesti lähetetään testinumeroon, suljetaan Viestit-ohjelma ja palataan aloitustilaan.
5	Internet-sivujen selailu WLAN-verkon yli	Avataan Internet selain, valitaan siirry Web-osoitteeseen valikko, kirjoitetaan osoite ja ladataan testisivu. Skripti tarkistaa, että sivu on latautunut, minkä jälkeen painetaan nuolinäppäintä alaspäin neljä kertaa sivun selausvaikutelman aikaansaamiseksi. Tämän jälkeen kirjoitetaan toinen web-osoite, ja prototyyppilaite aloittaa sen lataamisen. Skripti tarkistaa, että myös toinen web-sivu on latautunut, minkä jälkeen painetaan taas nuolinäppäintä alas neljä kertaa. Tämän jälkeen selain suljetaan ja palataan aloitustilaan.

6	Hälytys kelloon	Avataan prototyyppilaitteen Kello-ohjelma ja valitaan Uusi hälytys toiminto. Skripti laskee kellon ajan neljän minuutin verran eteenpäin ja lisää hälytyksen sille hetkelle. Kello-ohjelma suljetaan ja palataan aloitustilaan, jossa odotetaan hälytyksen alkamista. Skripti tarkistaa, että prototyyppilaite hälyttää oikealla hetkellä, minkä jälkeen hälytys lopetetaan. Prototyyppilaite jää aloitustilaan.
7	Kuvan ottaminen kameralla	Avataan Kamera-ohjelma ja tarkistetaan, että kamera on kuvanottotilassa. Skripti painaa kameranappia ja prototyyppilaite ottaa kuvan, minkä jälkeen skripti odottaa, että kuva on prosessoitu ja tallennettu. Kamera-ohjelmisto suljetaan ja palataan aloitustilaan.
8	Tiedostoja sisältävän kansion kopioiminen ulkoiselta muistikortilta massamuistiin	Avataan Tiedostonhallinta-ohjelma ja navigoidaan muistikortin kansiorakennelmaan. Valitaan kaksi tiedostoa sisältävä testikansio ja kopioidaan se massamuistin juureen. Tämän jälkeen navigoidaan massamuistille kopioidun tiedoston sijaintiin varmistaakseen, että tiedosto on kopioitunut oikein ja poistetaan testitiedosto. Lopuksi suljetaan Tiedostonhallinta-ohjelma ja palataan aloitustilaan.
9	Muistinkulutuksen sen hetkisen tilan taltioiminen	Muistinkulutuksen seurantaan suunniteltu ohjelma tallentaa prototyyppilaitteen sen hetkisen muistinkulutuksen tilan lokiin. Käytännössä muistinkulutuksen ohjelma avataan ja logiin kirjaus pistetään hetkeksi päälle, minkä jälkeen ohjelma suljetaan
10	Profiilin vaihto yleisestä äänettömään ja takaisin	Avataan Profiilit-ohjelma ja valitaan Äänetön profiili. Tämän jälkeen palataan aloitustilaan ja tarkistetaan että Äänetön profiili on käytössä. Palataan Profiilit-ohjelmaan, vaihdetaan Yleinen profiili käyttöön ja palataan aloitustilaan.

11	Muistikortilla sijaitsevan MP3-musiikkitiedoston toisto	Avataan Tiedostonhallinta-ohjelma ja navigoidaan muistikortilla sijaitsevan testattavan MP3-tiedoston sijaintiin. Avataan MP3-tiedosto, jolloin samalla avautuu myös musiikkisoitin. Tarkistetaan että prototyyppilaite toistaa testikappaletta ja odotetaan kolmekymmentä sekuntia. Tämän jälkeen toisto keskeytetään, suljetaan musiikkisoitin sekä Tiedostonhallinta-ohjelma ja palataan aloitustilaan.
12	Kontaktin lisääminen Puhelinluetteloon	Avataan Puhelinluettelo-ohjelma ja valitaan Lisää uusi nimi -toiminto. Lisätään kontaktille etunimi, sukunimi ja puhelinnumero, minkä jälkeen lisätään uusi kontakti puhelinluetteloon. Skripti tarkistaa, että juuri luotu kontakti on lisättynä kontaktilistaan, minkä jälkeen kontakti avataan ja tarkistetaan, että lisätyt tiedot ovat oikein. Tämän jälkeen suljetaan kontakti sekä Puhelinluettelo-ohjelma ja palataan aloitustilaan.

6.2 Testauskonseptin toteutus

Prototyyppilaitteiden testaus aloitettiin jokaisen prototyyppilaitteen kohdalla käynnistysnopeuden ensimmäisellä mittauksella heti ohjelmiston asennuksen jälkeen. Prototyyppilaite nro seitsemän erosi selkeästi muista testattavista prototyyppilaitteista ensimmäisessä käynnistysnopeuden mittauksessa, sillä se oli keskimäärin runsaat kymmenen sekuntia hitaampi kuin muut laitteet. Laitteen nro seitsemän käynnistysaika mitattiin vielä kaksi kertaa sen varmistamiseksi, ettei kyseessä olisi vain hetkellinen vikatila, mutta käynnistysnopeus pysyi edelleen noin kymmenen sekuntia hitaampana verrattuna muihin laitteisiin. Tämän jälkeen laitteeseen nro seitsemän asennettiin ohjelmisto uudestaan, jotta voitaisiin sulkea pois ohjelmiston asennuksessa mahdollisesti tapahtunut virhe. Tästä huolimatta laitteen nro seitsemän käynnistysnopeus pysyi edelleen noin kymmenen sekuntia hitaampana verrattuna muihin laitteisiin, joten kyseessä oli rautapohjainen vika. Laite nro seitsemän päätettiin kuitenkin lait-

taa testiajoon, jotta voitaisiin tutkia kuinka laajamittaisesta rautapohjaisesta viasta on kyse ja miten paljon se vaikuttaa laitteen toimintaan.

Ohjelmiston asennuksen ja ensimmäisen käynnistysnopeuden mittauksen jälkeen prototyyppilaitteisiin laitettiin automaattiseen ajoon tarvittavat manuaaliset asetukset kuten agentin konfigurointi käytettävään yhteysmuotoon, mikä oli tässä tapauksessa USB. Myös testidata, kuten toistettava MP3-tiedosto, lisättiin tässä vaiheessa muistikortille. Tämän jälkeen jokaiseen testattavaan prototyyppilaitteeseen laitettiin tarvittavat asetukset ajamalla asetusskripiti. Asetusten jälkeen jokaisesta prototyyppilaitteesta mitattiin käynnistysnopeus uudestaan ja edelleen vain nro seitsemän käynnistysaika erottui muiden laitteiden käynnistysnopeudesta.

Ennen jokaisen prototyyppilaitteen testiajon aloittamista todettiin laitteen ajovalmius ajamalla testitapausjoukko kerran läpi. Näin varmistettiin, että kyseisessä laitteessa olisivat kaikki tarvittavat asetukset oikein laitettu ja että testitapauksissa tarvittava testidata löytyisi muistikortilta. Tämän jälkeen prototyyppilaitteet laitettiin testiajoon, jossa kahdentoista testitapauksen joukko ajettiin jokaisella laitteella samassa järjestyksessä 17 kertaa läpi. Jokaisen prototyyppilaitteen testiajo suoritettiin yksitellen, jotta varattaisiin käyttöön vain yksi testiympäristö kerrallaan, eikä näin ollen testauskonseptin testiajoja jouduttaisi keskeyttämään virallisen testauksen tieltä. Testiajon jälkeen jokaisesta laitteesta mitattiin käynnistysnopeus, joissa ei edelleenkään ollut muuta poikkeavaa kuin laitteen nro seitsemän hidas käynnistysaika.

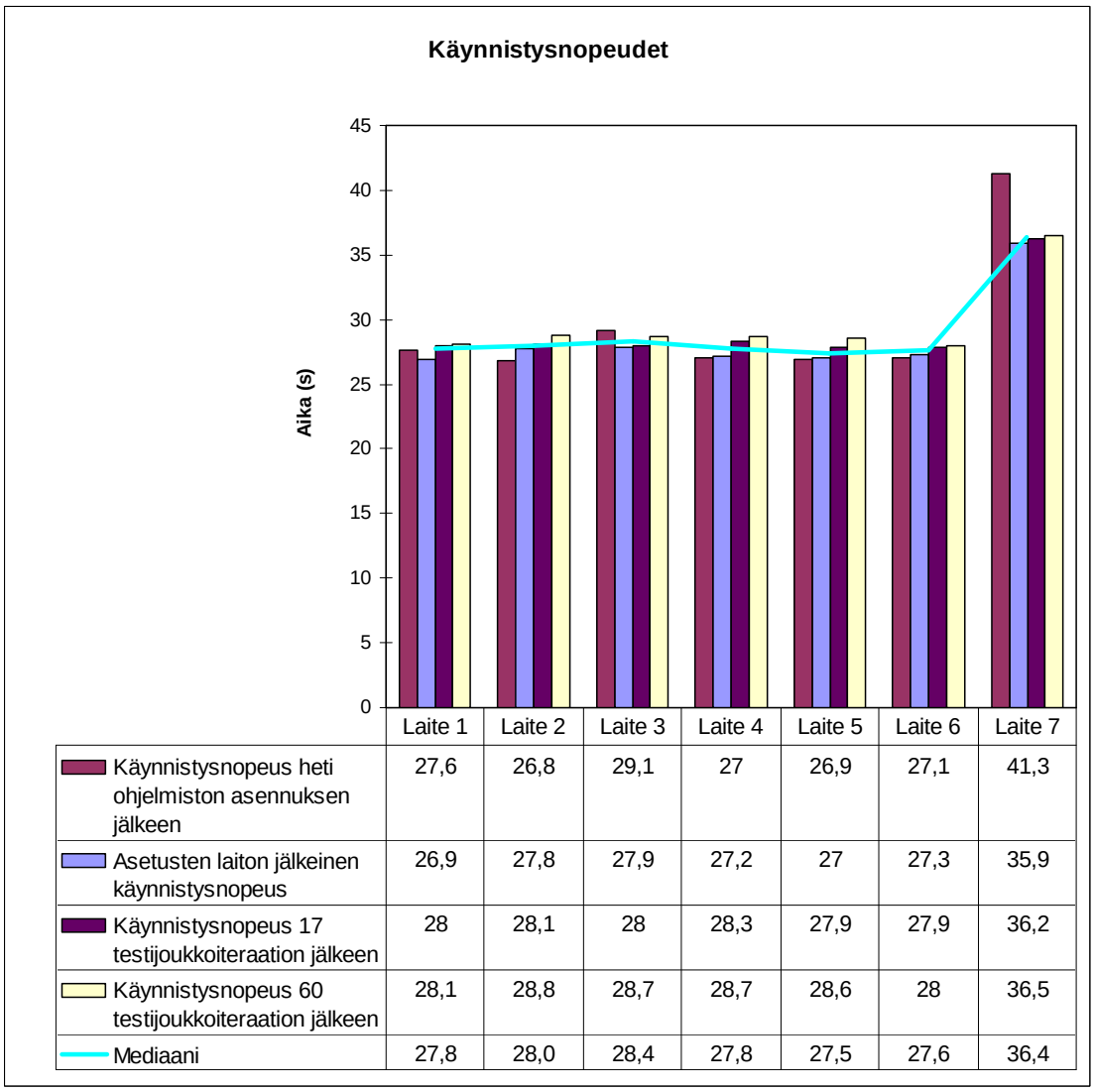
Testiajojen jälkeen käytiin saadut tulokset läpi ja todettiin, että muita rautapohjaisia vikoja kuin prototyyppilaitteen nro seitsemän hidas käynnistysaika, ei testiajon aikana esiintynyt. Tämä saattoi hyvin johtua siitä, että testattavana olevien laitteiden rautatoteutuksen kehityselinkaari oli lähes lopussa, jolloin rautapohjaisia virheitä ei yleensä esiinny runsaasti.
Jotta voitaisiin olla varmoja tulosten oikeellisuudesta, päätettiin kaikki laitteet laittaa pidempään testiajoon. Näin selviäisi myös se, että onko kontekstissa tarpeen kasvattaa testiajoajan pituutta prototyyppilaiteen kehityselinkaaren mukaan, siten että prototyyppilaitteiden testiajon kestoa kasvatetaan porrastetusti laitteen kypsyyden mukaan.

Pidemmässä referenssiajossa kahdentoista testitapauksen joukko laitettiin jokaisella prototyyppilaitteella samassa järjestyksessä ajoon kuusikymmentä kertaa. Yhdelle prototyyppilaitteelle tulisi näin ajoajaksi keskimäärin kolmekymmentä yhdeksän tuntia, mikä olisi testauskontekstin kannalta mahdollista toteuttaa myös virallisen testauksen ohella, sillä testaaja voi laittaa prototyyppilaitteet testaukseen viikonlopun yli. Testiajojen jälkeen mitattiin kaikkien laitteiden käynnistysnopeus viimeisen kerran. Käynnistysajoista ei löytynyt muuta poikkeavaa kuin nro seitsemän noin kymmenen sekuntia hitaampi käynnistysnopeus.

6.3 Tulokset

6.3.1 Käynnistysajat ja ajoajat

Laitteiden käynnistysnopeuksissa ei havaittu merkittäviä eroja lukuun ottamatta prototyyppilaitetta nro seitsemän. Muiden prototyyppien keskimääräinen käynnistysaika oli 27,9 sekuntia ja kaikki mitatut käynnistysnopeudet prototyyppilaitteille 1 – 6 sisältyivät suorituskykytestauksen mukaisesti laitteelle määriteltyyn marginaaliin, mikä oli tälle prototyyppilaitteelle kolme sekuntia. Laitteen nro seitsemän käynnistysnopeudet olivat tasaisesti noin yhdeksän sekuntia hitaampia verrattuna muihin prototyyppilaitteisiin lukuun ottamatta ensimmäistä käynnistysnopeutta. Ensimmäisen käynnistysajan selkeä kasvu saman laitteen muihin käynnistysnopeuksiin nähden selittyy sillä, että ensimmäisessä käynnistyksessä prototyyppilaite käy läpi ohjelmiston asennuksen jälkeiselle ensimmäiselle käynnistyskerralle ominaiset prosessit, kuten tiettyjen tietokantojen luomiset ja indeksoinnit. Tästä huolimatta kyseessä oli selkeästi rautapohjainen vika, sillä tämä toistui vain yhdellä laitteella, eikä myöskään ohjelmiston uudelleen asennus vaikuttanut tuloksiin. Kuviossa 9 on havainnoitu kaikkien prototyyppilaitteiden käynnistysnopeudet sekä niiden mediaani.

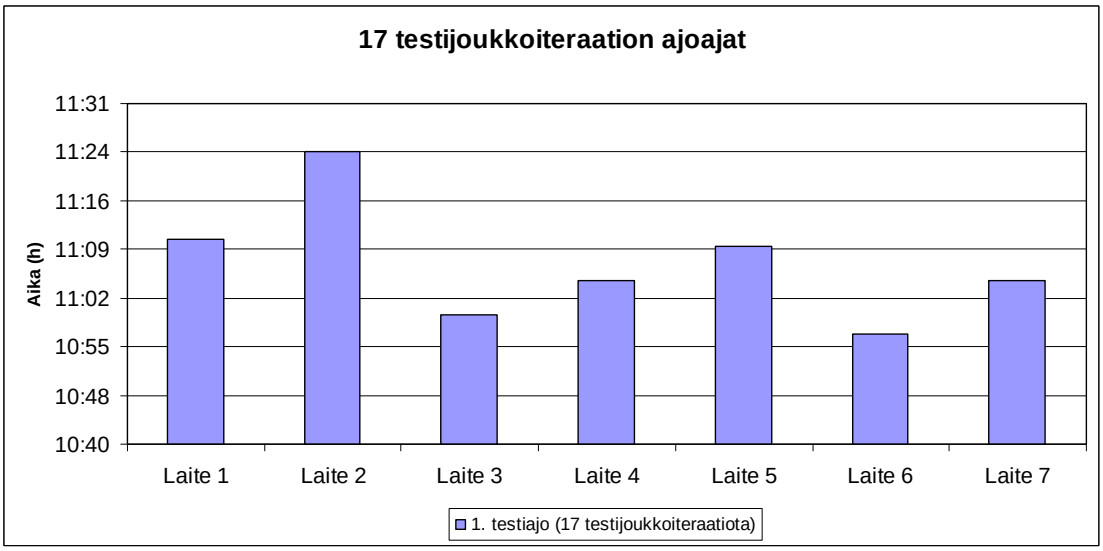


	Laite 1	Laite 2	Laite 3	Laite 4	Laite 5	Laite 6	Laite 7
Käynnistysnopeus heti ohjelmiston asennuksen jälkeen	27,6	26,8	29,1	27	26,9	27,1	41,3
Asetusten laiton jälkeinen käynnistysnopeus	26,9	27,8	27,9	27,2	27	27,3	35,9
Käynnistysnopeus 17 testijoukkoiteraation jälkeen	28	28,1	28	28,3	27,9	27,9	36,2
Käynnistysnopeus 60 testijoukkoiteraation jälkeen	28,1	28,8	28,7	28,7	28,6	28	36,5
Mediaani	27,8	28,0	28,4	27,8	27,5	27,6	36,4

KUVIO 9. Prototyyppilaitteiden käynnistysnopeudet

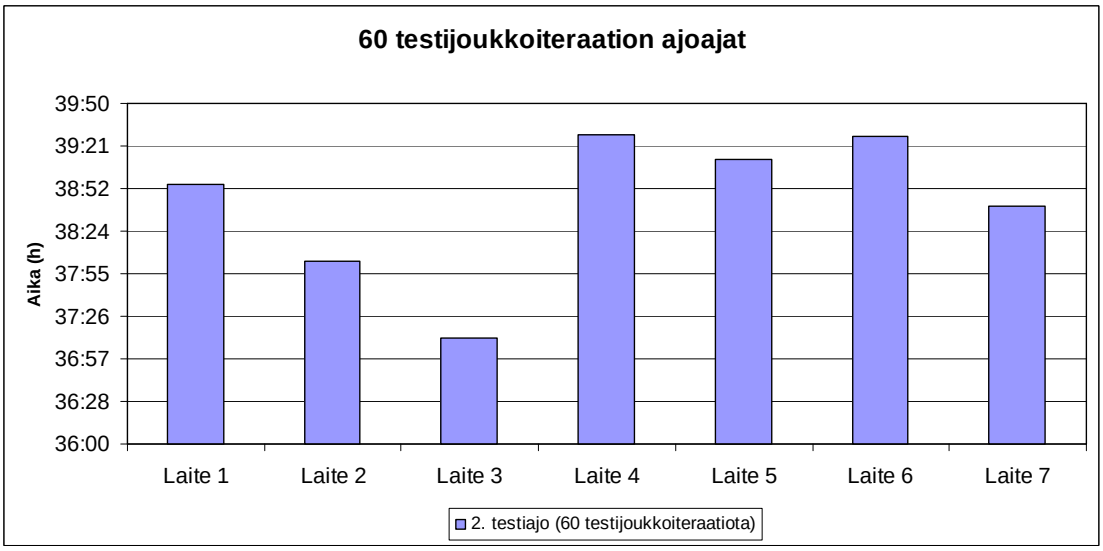
Kokonaisuudessa kaikki 17 testitapausjoukkoiteraation ajoajat kestivät alle 12 tuntia, jolloin tavoite lyhyistä testiajokestoista toteutui. Kovin selkeitä eroja ei ajokestojen välillä ollut. Suurin ero oli prototyyppilaitteen nro kuusi ja laitteen nro kaksi välillä ja sekin oli vain 27 minuuttia. Pienet ajokestojen erot selittyvät sillä, että ajokestoihin vaikuttaa jonkin verran verkon toimivuus, eli esimerkiksi kuinka pitkään menee aikaa siihen, että tekstiviesti saadaan lähetettyä tai kuinka pitkään kestää web-sivun latailu. Tähän voi vaikuttaa prototyyppilaitteen ohjelmisto ja rautatoteutus, mutta pitäisi olla selkeitä poikkeamia muihin testattaviin prototyyppilaitteisiin, jotta kyseessä olisi rautapohjainen vika. Myös automaatiolla oli vaikutusta ajokestoihin, sillä jokaisessa testiskriptissä tarkistettiin, että prototyyppilaite on suorittanut testitapauksen toiminnot oikein. Tämä tarkistus tapahtui vertaamalla etukäteen määriteltyä tekstiä tai kuvaa pro-

totyyppilaitteen näytön näkymään ja vertailun onnistumiseen kuluva aika vaihteli aina hieman. Suuria suoritusaikaeroja ei kuitenkaan automaatiosta johtuen voinut tulla, sillä kaikissa testiajoissa käytettiin samaa automaatioympäristöä, jolloin automaation aiheuttama viive oli samassa suhteessa kaikissa testatuissa prototyyppilaitteissa. Yksittäisen testitapausiteraation kohdalla edellä mainituista mahdollisista viiveistä tuli vain lyhyt lisä ajokestoon, mutta pienet lisät ajokestoon yhdistettynä 204 testitapausiteraation osalta vaikuttivat kokonaisajoaikaan useita kymmeniä minuutteja. Tästä huolimatta kyseessä ei voinut olla rautapohjainen vika, sillä ei tullut esiin selviä poikkeamia eri prototyyppilaitteiden välillä. Kuviossa 10 on havainnoitu 17 testitapausjoukon iteraatioiden kesto prototyyppilaitteittain.



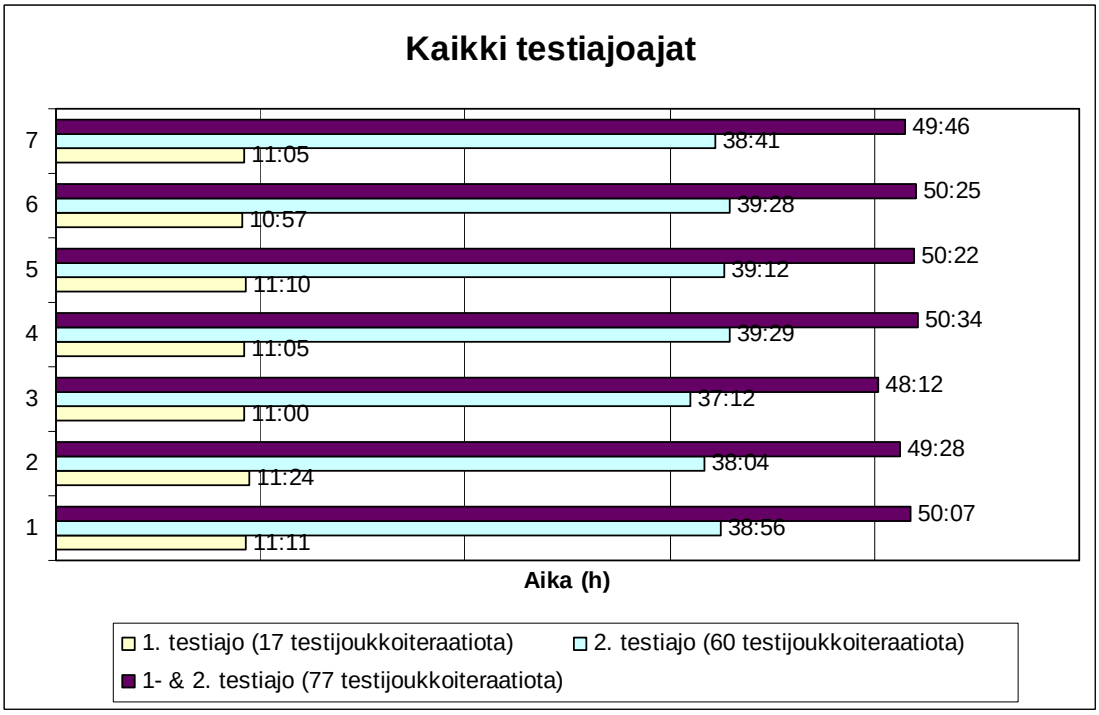
KUVIO 10. 17 testijoukkoiteraation ajoajat prototyyppilaitteittain

Kaikilla prototyyppilaitteilla pysyivät myös pidemmän referenssiajon testitapausjoukon 60 iteraation testiajokestot hyvin tasaisesti samassa suhteessa kuin testitapausjoukon 17 iteraation ajoajat. Iteraatiomäärien kasvusta johtuen pidemmässä ajossa erot vaan näkyivät selvemmin. Syyt ajokestojen eroihin olivat samat kuin mitä lyhyemmissäkin testitajoissa. Kuviossa 11 on havainnoitu testijoukon 60 iteraation ajoajat prototyyppilaitteittain.



KUVIO 11. 60 testijoukkoiteraation ajoajat prototyyppilaitteittain

Kuten kuviossa 12 on havainnollistettu, suuria eroja ei tullut prototyyppilaitteiden ajoaikoihin kummassakaan testiajossa. Kuvioon on myös laitettu molempiin testiajoihin yhteensä kulunut ajoaika prototyyppilaitteittain sen varmistamiseksi, ettei olisi ollut poikkeamia kokonaisajokestoissa laitteiden välillä.

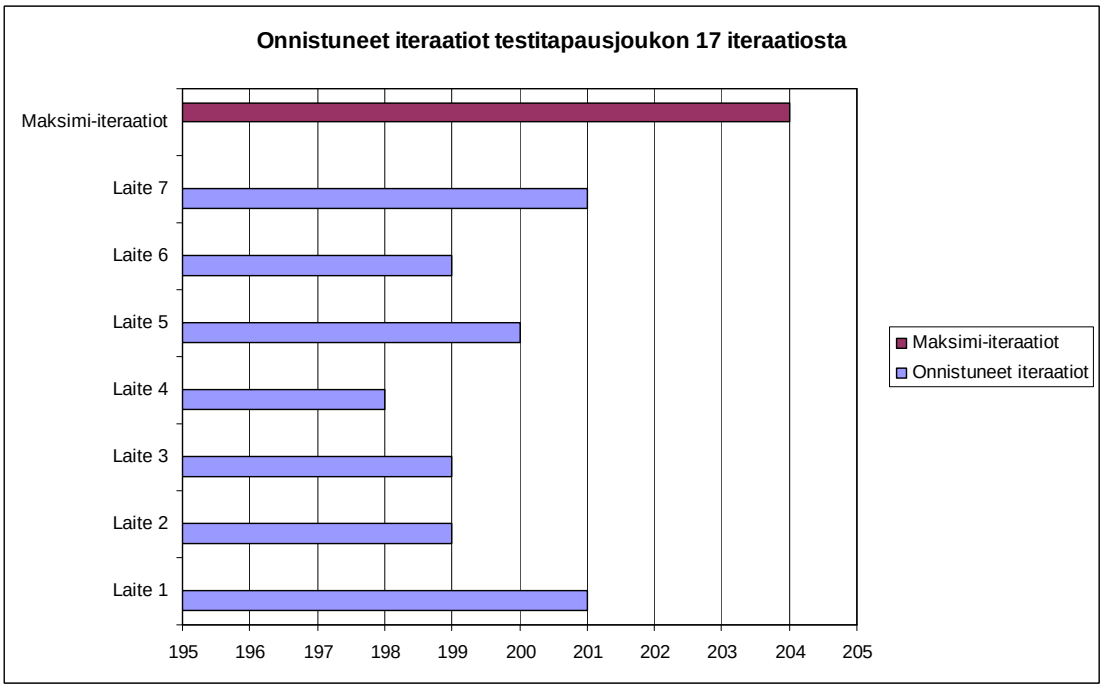


KUVIO 12. Kaikkien testiajojen kestot sekä ajoajat yhteensä prototyyppilaitteittain

6.3.2 Testitapausjoukkokohtaiset tulokset

Samoja skriptejä pystyttiin käyttämään kaikissa prototyyppilaitteiden testiajoissa, jolloin ajoihin vaadittava työmäärä pysyi pienenä. Testiajojen yhteydessä löydettiin yksi ohjelmistopohjainen virhetila, joka toistui Internet-selainta käytettäessä. Virhetila toistui kuitenkin kaikissa prototyyppilaitteiden testiajoissa, joten kyseessä ei ollut rautapohjainen vika. Muut epäonnistuneet iteraatiot johtuivat automaation suorasukaisuudesta. Esimerkiksi valitaan ”Sulje” valikko muistikortilta avatun musiikin toiston jälkeen, laite palaa muistikortille, mutta saattaa hidastua hetkellisesti prosessoimaan tietokantaa eikä välttämättä reagoi heti muistikorttinäkymän näytölle piirron kanssa samanaikaisesti tapahtuneeseen Takaisin-valikon painantaan. Normaalisti käyttäjä ei välttämättä edes ehtisi painamaan näppäintä yhtä nopeasti kuin automaatio, ja jos ehtisikin, niin painaisi hetken päästä uudestaan Takaisin valintaa. Automaatio taas olettaa ensimmäisen painalluksen epäonnistumisen jälkeen testitapauksen iteraation epäonnistuneen, sillä seuraavaa määriteltyä parametria ei löydy laitteen näytöltä edellisen käskyn epäonnistumisesta johtuen.

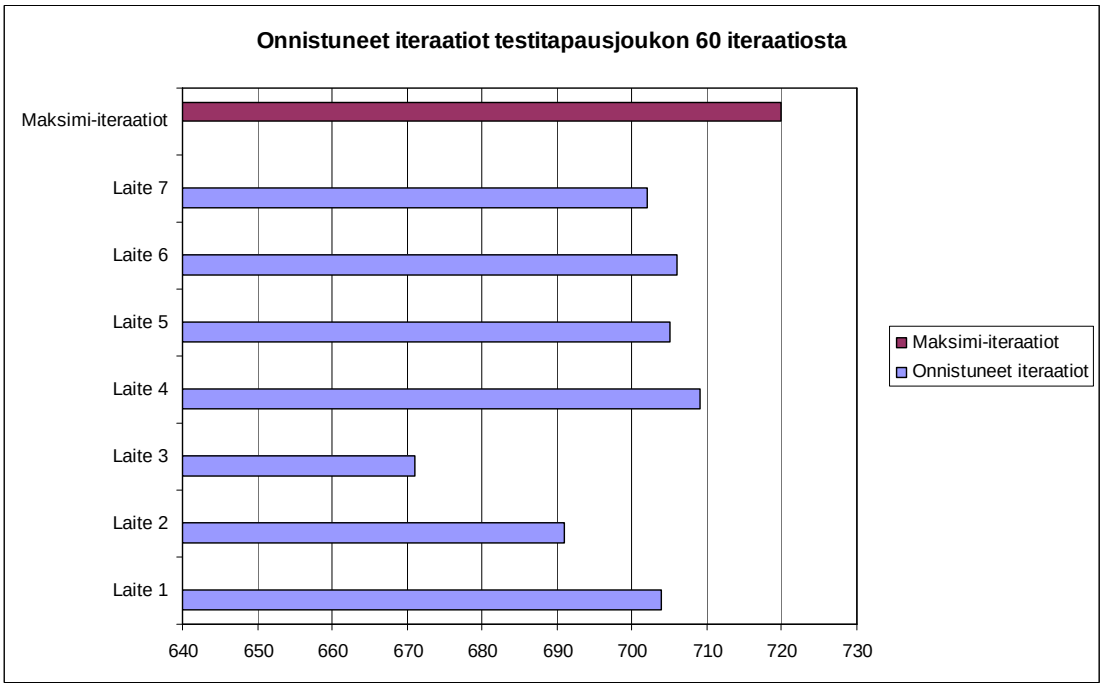
Yhdestä ohjelmistopohjaisesta toistuvasta virhetilasta ja automaatiosta johtuvista epäonnistuneista iteraatioista huolimatta testiajojen onnistuneiden iteraatioiden prosentit olivat huomattavasti korkeammalla verrattuna aikaisemmassa kehitysvaiheissa oleviin prototyyppimallien testiajoihin. Tämä johtui siitä, että sekä testauskäytössä olevat rautatoteutuksen että ohjelmiston versiot olivat kyseisen prototyyppimallin kehityskaaren loppuvaiheessa. 17 testitapausjoukkoiteraatioiden onnistumisprosentti oli kaikissa prototyyppilaitteissa vähintään 97%. Kuviossa 13 on havainnollistettu prototyyppilaitekohtaisesti onnistuneiden iteraatioiden määrä verrattuna 204 testitapausiteraation maksimimäärään.



KUVIO 13. Onnistuneet iteraatiot maksimi-iteraatiomäärän ollessa 204

Verratessa 17 testitapausjokkoiteraation tuloksia 60 testijoukkoiteraation tuloksiin todettiin, ettei ajoajan kasvattaminen vaikuttanut ollenkaan rautapohjaisten vikojen esille tulemiseen. Laitteen nro seitsemän käynnistysaika pysyi hitaampana samassa suhteessa muihin testattaviin laitteisiin sekä ennen referenssiajoa että sen jälkeen. Ainoastaan ohjelmistopohjaisia virhetiloja rupesi esiintymään enemmän testiajokeston kasvaessa.

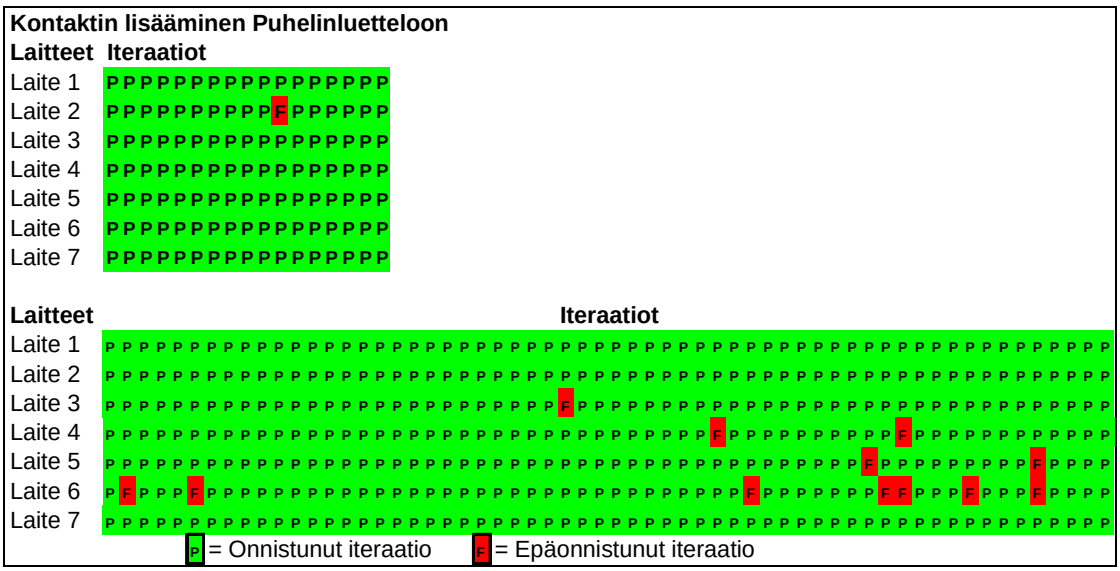
Myös 60 testitapausjoukkoiteraatioiden onnistuneisuusprosentit olivat hyvin korkealla. Alhaisin onnistumisprosentti, eli 93%, oli prototyyppilaitteella nro kolme. Tämä johtui enimmäkseen siitä, että toisen testikäytössä olleen websivun palvelimelle tuli huoltokatkos testiajon aikana, joten sivun tunnistus ei toiminut tekstin poiketessa valmiiksi määritellystä. Seuraavaksi alhaisin onnistumisprosentti, eli 96%, oli prototyyppilaitteella nro kaksi, jossa referenssiajon aikana laite hidastui Internet-selaimen käytön aikana ohjelmistopohjaisen virhetilan takia ja Siirry web-sivulle -toiminnon valinta automaattisesti epäonnistui usein. Muilla prototyyppilaitteilla onnistumisprosentti oli vähintään 97,5%. Kuviossa 13 on havainnollistettu prototyyppilaitekohtaisesti onnistuneiden iteraatioiden määrä verrattuna 720 testitapausiteraation maksimimäärään.



KUVIO 14. Onnistuneet iteraatiot maksimi-iteraatiomäärien ollessa 720

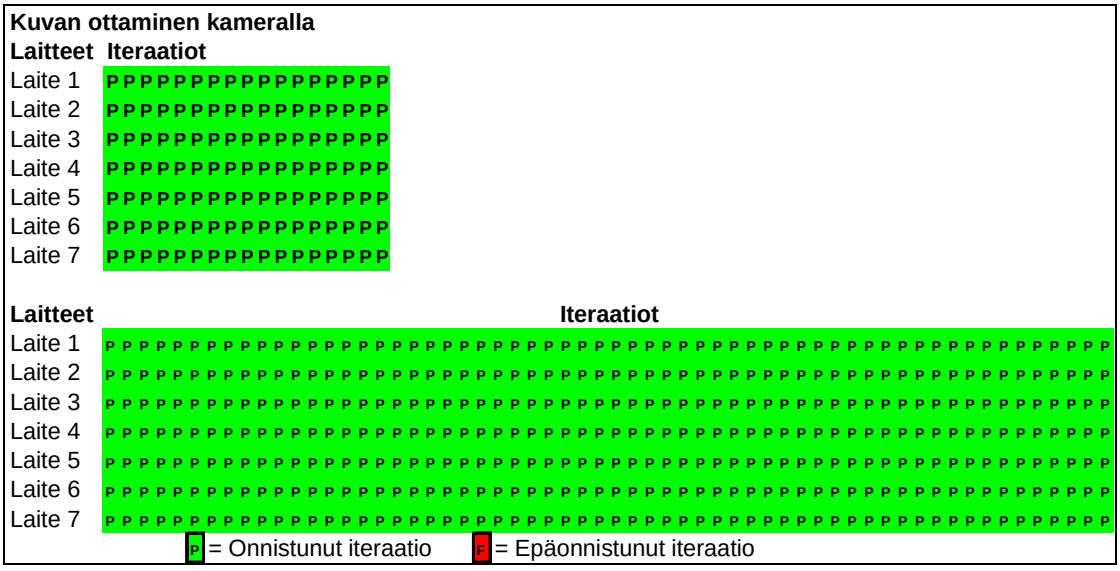
6.3.3 Testitapauskohtaiset tulokset

17 testitapausjoukon iteraation aikana testitapaus Kontaktin lisääminen Puhelinluetteloon epäonnistui vain kerran johtuen automaation epäonnistumisesta kontaktin numeron kirjoittamisessa kokonaan. 60 testitapausjoukkoiteraation aikana automaatiosta johtuvia epäonnistuneita iteraatioita tuli hieman enemmän johtuen prototyyppilaitteiden lievästä hidastumisesta jatkuvan käytön aikana. Kuviossa 15 havainnollistetaan Kontaktin lisääminen Puhelinluetteloon testitapauksen iteraatiot sekä 60 testijoukkoiteraation että 17 testijoukkoiteraation osalta.



KUVIO 15. Testitapaus ”Kontaktin lisääminen Puhelinluetteloon” iteraatiot

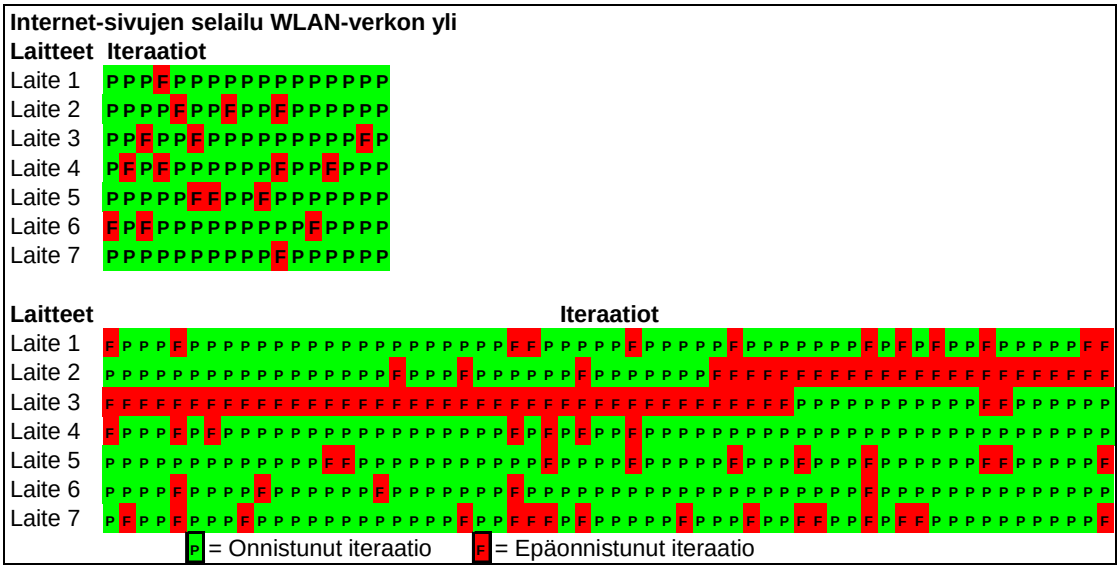
Kamerakomponentit toimivat kaikissa prototyyppilaitteissa moitteettomasti, eikä yhtään epäonnistunutta iteraatiota tapahtunut yhdelläkään laitteella kummassakaan testiajossa. Kuten kuviossa 16 näkyy, yksikään iteraatio ei epäonnistunut tämän testitapauksen ajoissa.



KUVIO 16. Testitapaus ”Kuvan ottaminen kameralla” iteraatiot

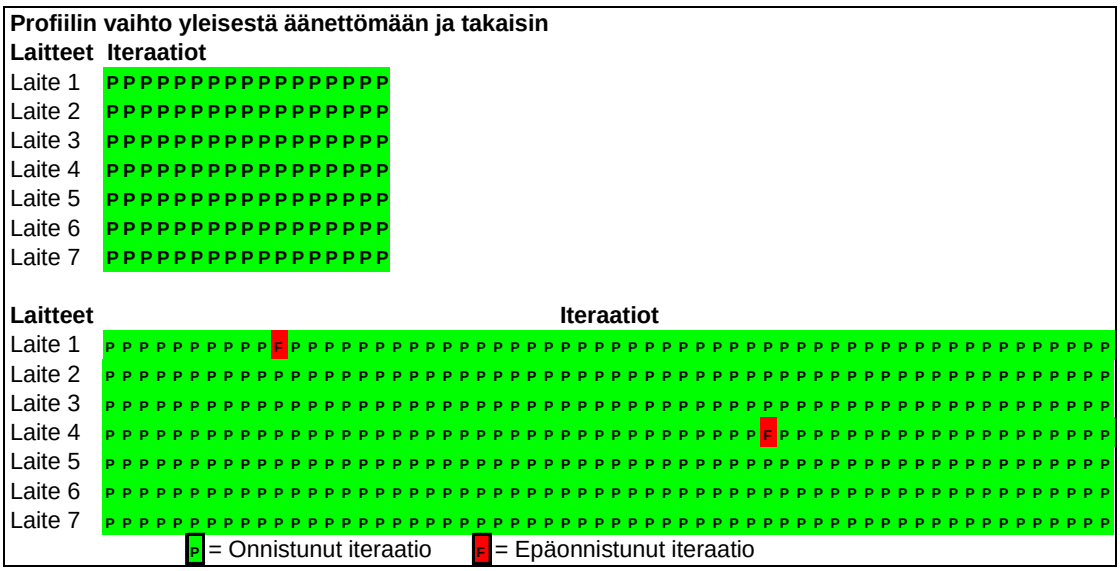
Testitapauksen ”Internet-sivujen selailu WLAN-verkon yli” yhteydessä tuli esille ohjelmistopohjainen virhe, jossa selaimen web-sivun lataus epäonnistuu. Tämä toistui jokaisella prototyyppilaitteella jo 17 testitapausjoukkoiteraation

aikana. 60 testitapausjokkoiteraation testiajon aikana prototyyppilaite nro 2 hidastui Internet-selaimen käytön aikana edellä mainitusta virhetilasta johtuen, jolloin ”Siirry” valinnan painaminen web-sivulle siirtymiseksi epäonnistui yhteensä 19 kertaa. Tämä automaatiosta johtuva virhe toistui laitteella nro kaksi useammin kuin muilla prototyyppilaitteella, mutta itse ohjelmistopohjainen virhe toistui myös laitteella nro kaksi keskimäärin yhtä monta kertaa kuin muillakin laitteilla. Laitteen nro kolme suuri määrä epäonnistuneita iteraatioita tässä testitapauksessa johtui enimmäkseen siitä, että tämän testiajon aikana testisivun palvelimella oli huoltokatkos. Ohjelmistopohjainen virhetila toistui vain seitsemän kertaa, mikä oli samassa suhteessa virhetilan esiintymisen määrään muissa prototyyppilaitteissa. Kuviossa 17 näkyy testitapauksen ”Internet-sivujen selailu WLAN-verkon yli” iteraatiot sekä 60 testijoukkoiteraation että 17 testijoukkoiteraation osalta.



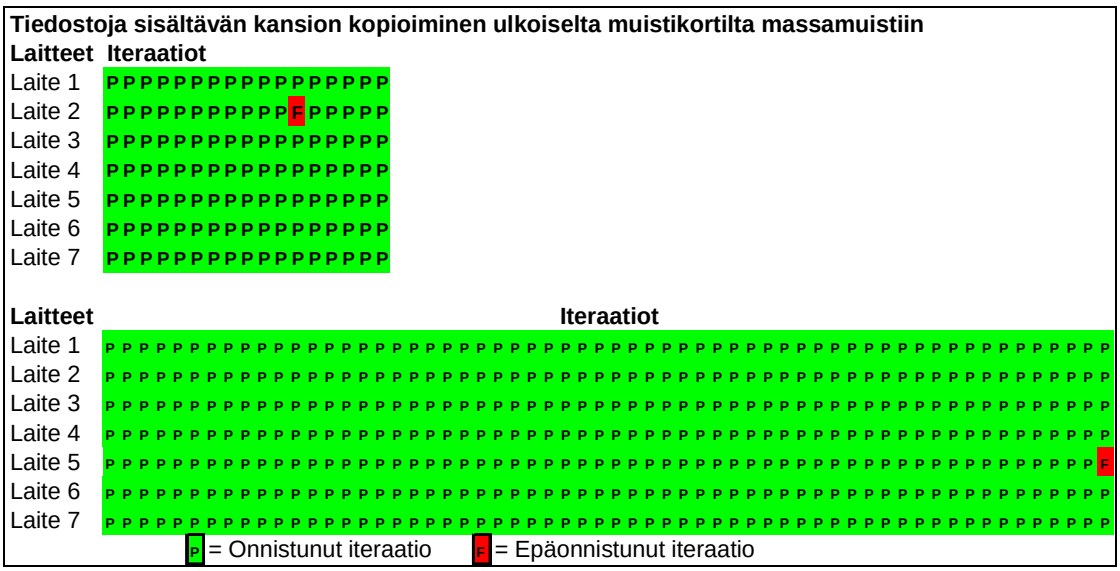
KUVIO 17. Testitapaus ”Internet-sivujen selailu WLAN-verkon yli” iteraatiot

Testitapaus ”Profiilin vaihto yleisestä äänettömään ja takaisin” onnistui kaikilla prototyyppilaitteilla hyvin. Kuten kuviosta 18 näkyy, 17 testitapausjoukkoiteraation aikana ei tapahtunut yhtään epäonnistunutta iteraatiota ja 60 testitapausjoukkoiteraation aikana epäonnistui vain yksi iteraatio kahdella prototyyppilaitteella automaatiosta johtuen.



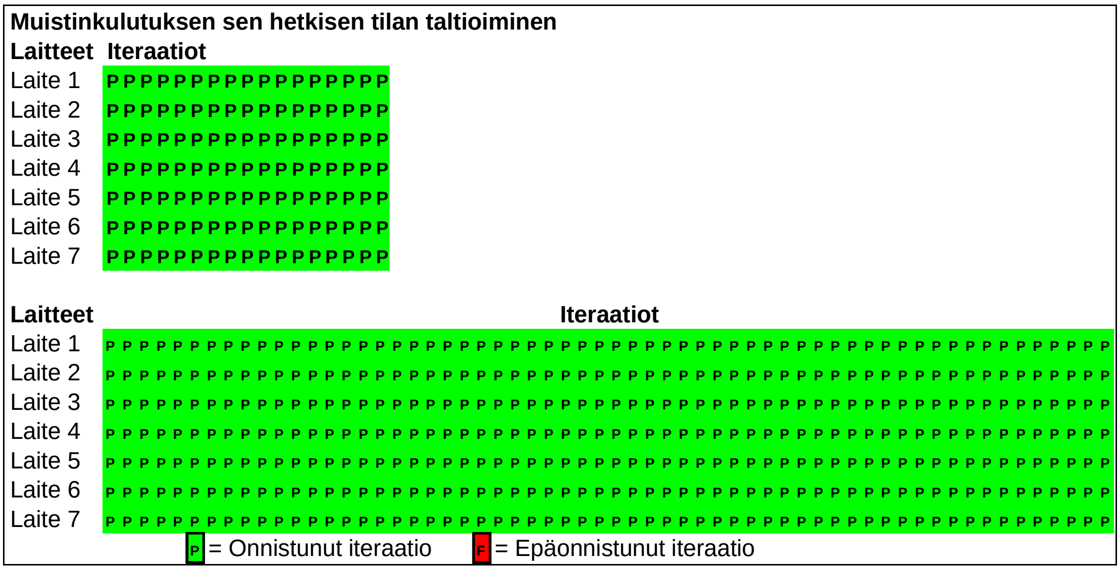
KUVIO 18. Testitapaus ”Profiilin vaihto yleisestä äänettömään ja takaisin” iteraatiot

Testitapaus ”Tiedostoja sisältävän kansion kopioiminen ulkoiselta muistikortilta massamuistiin” iteraatioiden onnistuneisuuden perusteella prototyyppilaitteiden datansiirron toiminnallisuus tietojärjestelmän sisällä toimi erinomaisesti. Kuviossa 19 on havainnollistettu, että 17 testitapausjoukkoiteraation aikana vain yksi iteraatio epäonnistui automaatiosta johtuen laitteella nro kaksi. Automaatio aiheutti myös 60 testitapausjoukkoiteraation yhden epäonnistuneen iteraation laitteella nro viisi.



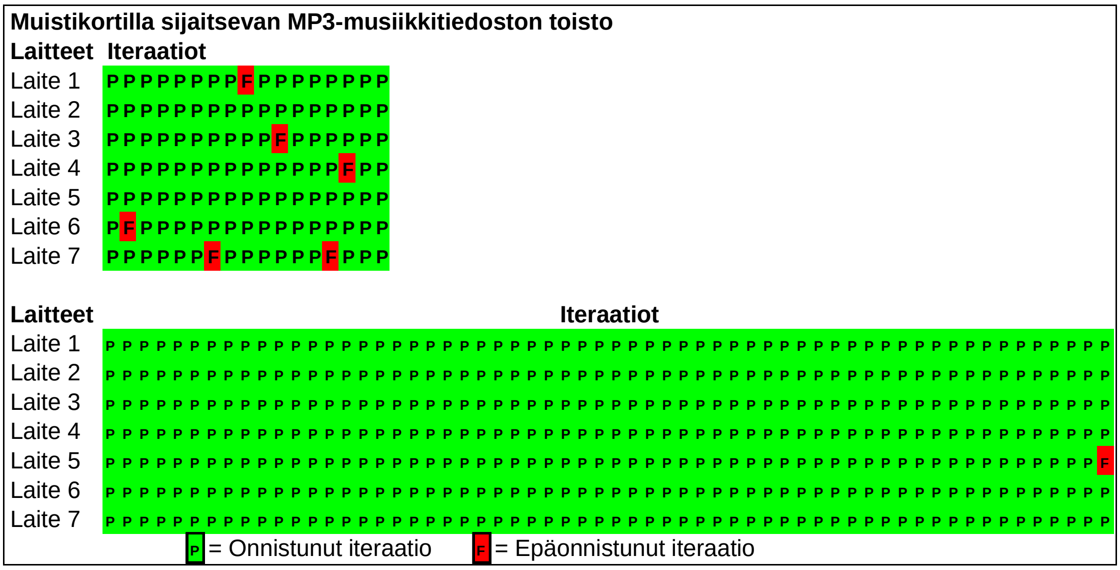
KUVIO 19. Testitapaus ”Tiedostoja sisältävän kansion kopioiminen ulkoiselta muistikortilta massamuistiin” iteraatiot

Testitapaus ”Muistinkulutuksen sen hetkisen tilan taltioiminen” jokainen iteraatio onnistui kaikilla prototyyppilaitteilla, mikä on havainnollistettu kuviossa 20. Muistinkulutuksessa ei esiintynyt mitään poikkeamia, eikä muistivuotoja havaittu yhdelläkään laitteella.



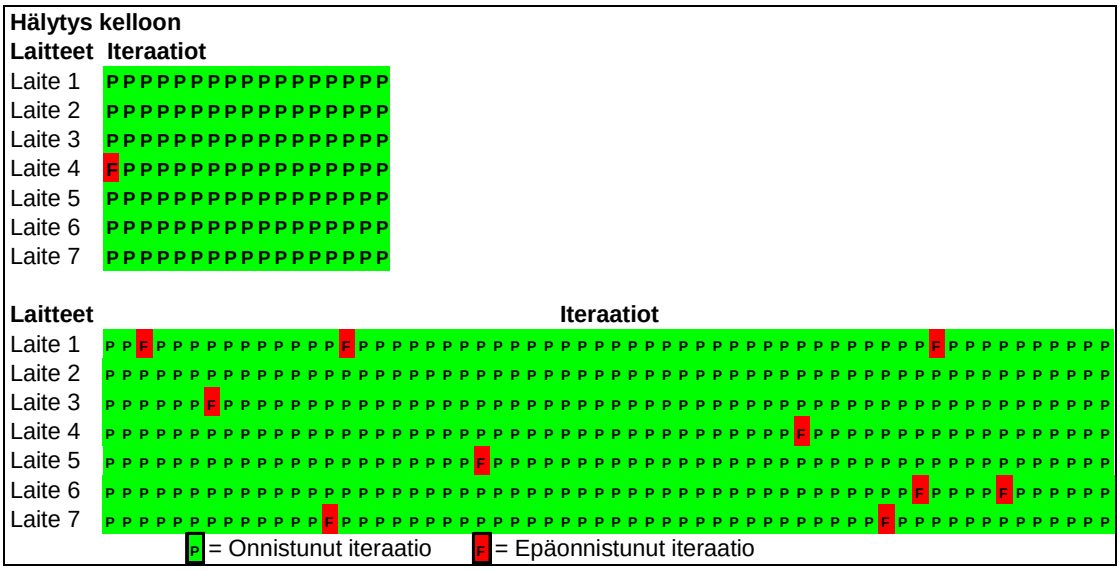
KUVIO 20. Testitapaus Muistinkulutuksen sen hetkisen tilan taltioiminen iteraatiot

Testitapaus ”Muistikortilla sijaitsevan MP3-musiikkitiedoston toisto” epäonnistui joitakin kertoja 17 testitapausjoukkoiteraation aikana johtuen siitä, että prototyyppilaitteet hidastuivat hieman hetkellisesti musiikintoisto-ohjelman sulkemisen jälkeen. Samoin tapahtui kerran laitteelle viisi 60 testitapausjoukkoiteraation aikana, kuten kuviosta 21 näkyy.



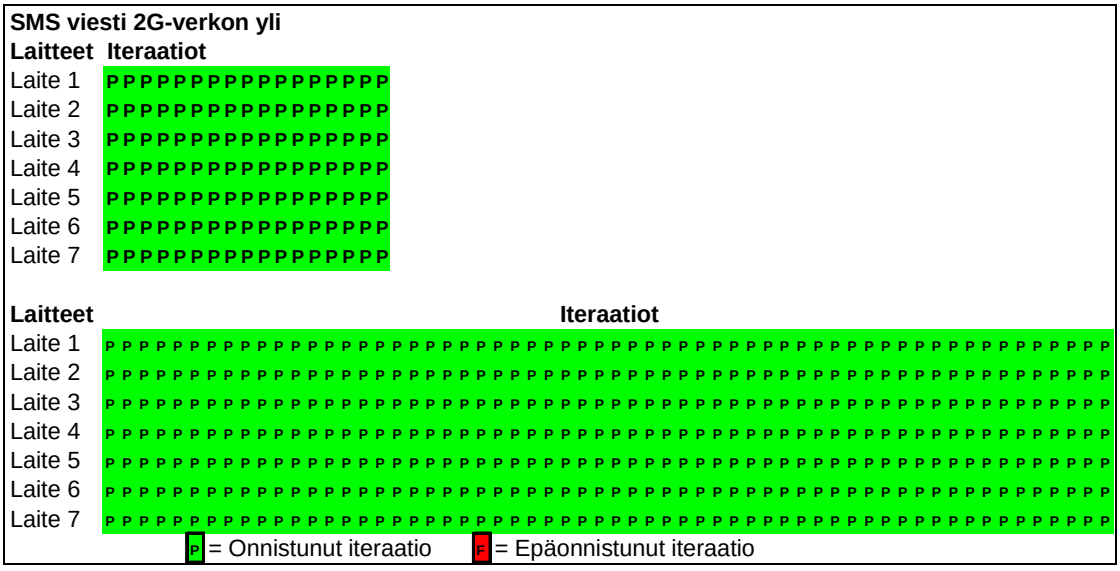
KUVIO 21. Testitapaus ”Muistikortilla sijaitsevan MP3-musiikkitiedoston toisto” iteraatiot

Testitapaus ”Hälytys kelloon” iteraatioista epäonnistui vain yksi automaatiosta johtuen laittella nro neljä 17 testitapausjoukkoiteraation aikana. 60 testitapausjoukkoiteraation aikana testitapauksen iteraatioita epäonnistui hieman enemmän, koska automaatiolla oli vaikeuksia kirjoittaa hälytyksen aika onnistuneesti. Epäonnistuneita iteraatioita tapahtui kuitenkin hyvin vähän, kuten kuviosta 22 näkyy.



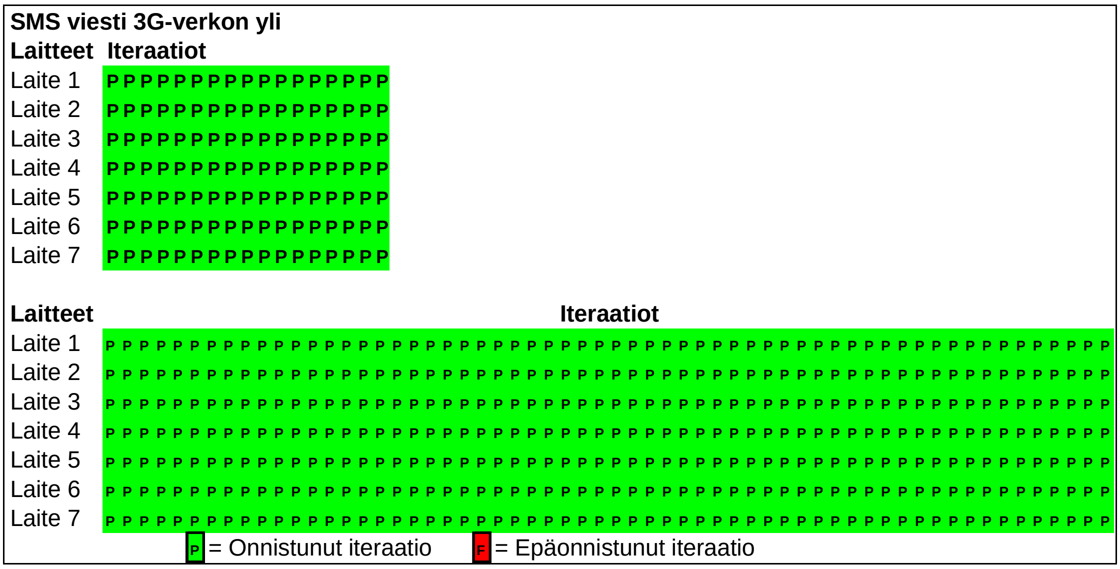
KUVIO 22. Testitapaus ”Hälytys kelloon” iteraatiot

Testitapaus ”SMS viesti 2G-verkon yli” jokainen iteraatio onnistui kaikilla prototyyppilaitteilla, mikä on havainnollistettu kuviossa 23.



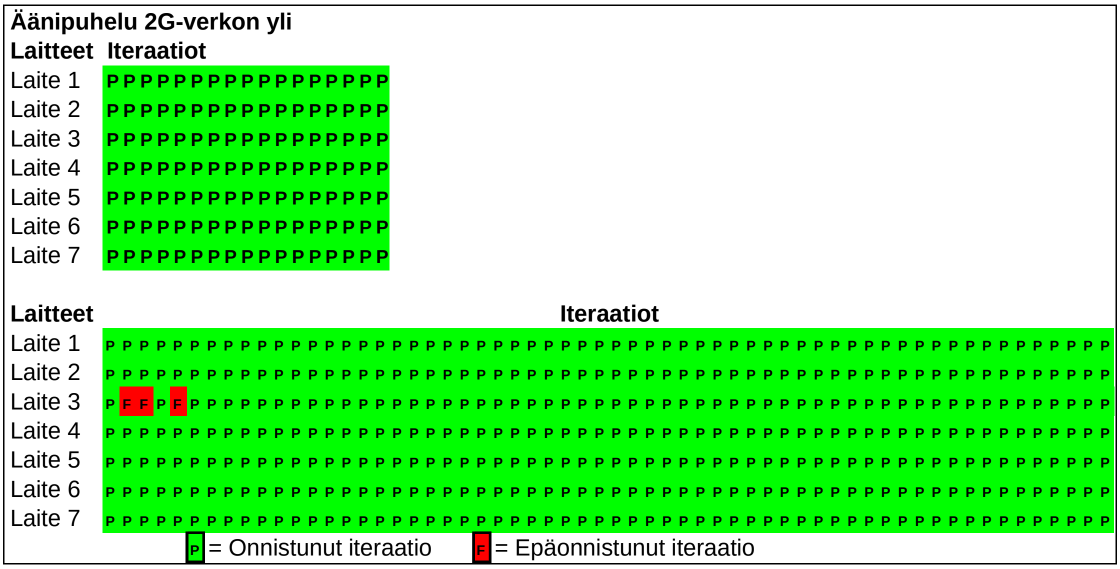
KUVIO 23. Testitapaus ”SMS viesti 2G-verkon yli” iteraatiot

Testitapaus ”SMS viesti 3G-verkon yli” jokainen iteraatio onnistui kaikilla prototyyppilaitteilla, mikä on havainnollistettu kuviossa 24.



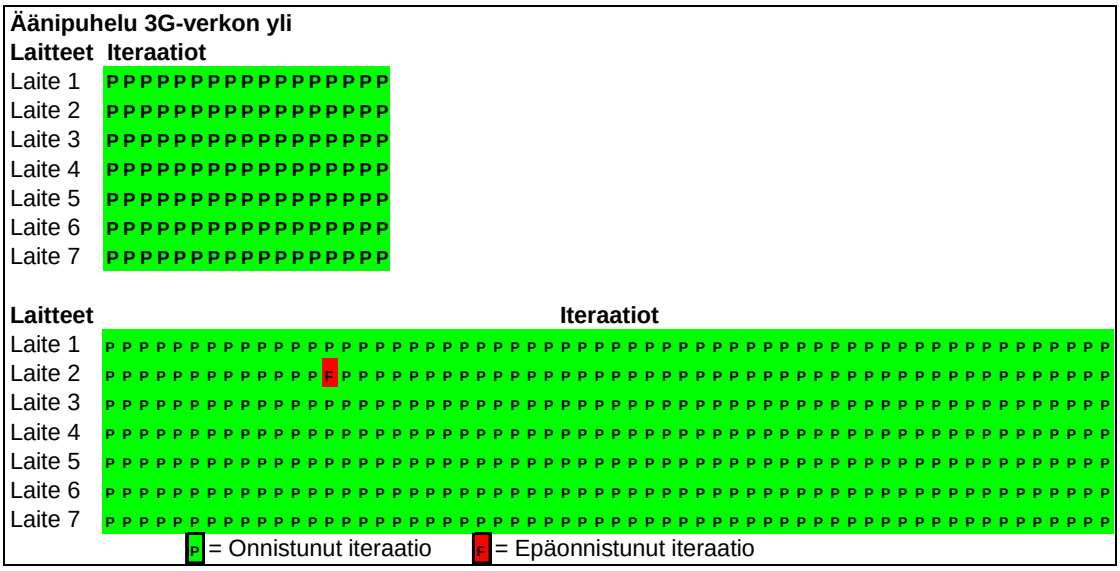
KUVIO 24. Testitapaus ”SMS viesti 3G-verkon yli” iteraatiot

Testitapaus ”Äänipuhelu 2G-verkon” yli 17 testitapausjoukkoiteraatioista onnistuivat kaikki ja 60 testitapausjoukkoiteraatioista epäonnistui vain kolme iteraatiota laitteella nro kolme, koska automaatio epäonnistui painamaan soitto näppäintä. Testitapauksen iteraatioiden tulokset on havainnollistettu kuviossa 25.



KUVIO 25. Testitapaus ”Äänipuhelu 2G-verkon yli” iteraatiot

Testitapaus ”Äänipuhelu 3G-verkon yli” 17 testitapausjoukkoiteraatioista onnistuivat kaikki ja 60 testitapausjoukkoiteraatioista epäonnistui vain yksi iteraatio laitteella nro kaksi, koska automaatio epäonnistui painamaan soitto näppäintä. Testitapauksen iteraatioiden tulokset on havainnollistettu kuviossa 26.



KUVIO 26. Testitapaus ”Äänipuhelu 3G-verkon yli” iteraatiot

7 OPINNÄYTETYÖN JÄLKITARKASTELU

7.1 Pohdinta

Prototyyppilaitteiden automatisoidun rautapohjaisen suorituskyvyn testauskonseptin käytännön toteutus osoitti, että prototyyppilaitteiden mahdolliset rautapohjaiset viat voidaan löytää konseptissa suunniteltuja kestävyystestaukseen ja suorituskykytestaukseen pohjautuvia mustalaatikkotestaustyyppisiä toimintatapoja käyttäen. Käytännön toteutuksessa löytyi vain yksi rautapohjainen vika, mikä johtui siitä, että testattavat prototyyppilaitteet olivat kehityselinkaarensa lopussa varsinkin rautatoteutuksen osalta, jolloin rautapohjaisia vikoja ei yleensä edes pahemmin esiinny. Konseptin virallisessa käyttöön otossa testattaviksi prototyyppilaitteiksi tulisi myös kehityskaarensa aikaisemmissa vaiheissa olevia laitteita, jolloin sekä rautapohjaisten että ohjelmistopohjaisten virhetilanteiden esiin tuleminen kasvaisi, mutta toteutuksen perusperiaate pysyisi silti samana. Konseptin toteutuksessa olisi voitu käyttää myös aikaisemmassa vaiheessa olevia prototyyppilaitteita, mutta siinä tapauksessa vaara testiajojen keskeytyksestä virallisen ohjelmistopohjaisen testauksen tieltä olisi kasvanut huomattavasti eikä konseptin tekemiseen suunnitellusta aikataulusta olisi ollut mahdollista pitää kiinni. Kehityskaaren loppuvaiheessa olevien prototyyppien käyttö käytännön toteutuksessa varmisti myös sen, että konsepti toimii kehityskaaren kaikissa vaiheissa. Konseptin toteutuksessa käytetty viimeisin ohjelmistoversio takasi myös sen, ettei paljon ohjelmistopohjaisia virhetilanteita tullut esiin. Konseptin virallisessa käytössä varsinkin aikaisessa kehitysvaiheessa olevia prototyyppilaitteita joudutaan testaamaan myös keskeneräisellä ohjelmistoversiolla, mutta vaikka ohjelmistopohjaisia virhetiloja esiintyisi tästä johtuen enemmän, niiden erottaminen rautapohjaisista vioista olisi helppoa, sillä ne toistuisivat kaikilla testattavilla laitteilla toisin kuin rautapohjaiset viat.

Testauskonseptissa tehdystä pidemmästä referenssiajosta huomattiin se, ettei ajokestolla ollut juurikaan vaikutusta rautapohjaisten vikojen löytymiseen. Ajoajan kasvaessa tuli vain enemmän ohjelmistopohjaisia virhetilanteita esille, eikä näiden löytäminen ollut konseptin tarkoitus. Ottaen huomioon, että rauta-

pohjaiset viat ovat prototyyppilaitteissa valmiina ennen testauksen aloittamista, niin pidempi kuin noin puolen vuorokauden testiajo yhdelle prototyyppilaitteelle ei vaikuttaisi rautapohjaisten vikojen esille tulemiseen. Toki hyvin pitkän ajan, esimerkiksi kuukausia kestävän testauksen myötä on mahdollista, että myös prototyyppilaitteiden rautatoteutus hajoaa esimerkiksi mekaniikan kulumisen myötä, mutta tämä tapahtuisi hyvin hitaasti, eikä konseptin tarkoitus ollut hajottaa laitteiden rautatoteutusta hyvin massiivisella ja pitkäkestoisella automatisoidulla testauksella.

Tuloksien läpikäymisessä huomattiin, että prototyyppilaitteella nro seitsemän oli poikkeuksellisen pitkä käynnistysnopeus. Tämän vaikutuksia esimerkiksi suorituskykytestauksen alialueisiin, kuten vasteaikojen mittaukseen, ei voida tietää ilman tarkempaa vertailua eri prototyyppilaitteiden välillä, mutta tässä tapauksessa voidaan kuitenkin päätellä, ettei laite nro seitsemän ollut sopiva suorituskykytestaukseen. Laitteen nro seitsemän testiajojen tuloksista huomattiin, ettei hidas käynnistysnopeus kuitenkaan vaikuttanut laitteen suorituskykyyn automatisoidun kestävyyskyvyn testitapauksien osalta, joten periaatteessa sitä olisi voitu käyttää myös viralliseen ohjelmistopohjaiseen kestävyystestaukseen sen jälkeen, kun olisi selvitetty, mikä aiheuttaa hitaan käynnistysajan. Ilman testauskonseptin käytännön toteutusta tätä ei kuitenkaan olisi voitu tietää.

Testauskonseptin toteutukseen olisi voitu lisätä kellon ajassa pysymisen seuranta. Prototyyppilaitteiden kellot synkronoidaan automaattisesti asetuksien laiton yhteydessä PC:n – jossa automaatioympäristö sijaitsee – kellon kanssa. Testauksen lopettamisen yhteydessä varmistettaisiin, että prototyyppilaitteen ja käytetyn PC:n kello olisivat edelleen samassa ajassa. Samoin testauskonseptin testitapaus ”Profiilin vaihto yleisestä äänettömään ja takaisin” olisi voitu vaihtaa testitapaukseen ” Profiilin vaihto yleisestä poiskytkettyyn (Offline) ja takaisin”. Prototyyppilaitteen verkon pois kytkeminen olisi vaikuttanut laajemmin laitteen toimintoihin kuin pelkästään äänettömälle laittaminen.

Konsepti olisi myös ollut periaatteessa mahdollista toteuttaa suoraan rautatoteutusta testaamalla ilman ohjelmistoversion mukaan ottamista. Mutta siihen ei toimeksiantajalla ole tällä hetkellä ole tarvittavia laitteita, joten rautapohjai-

sen suorituskyvyn testaus on siis toistaiseksi suoritettava ohjelmistoversiota käyttäen.

Yhteenvetona testauskonseptin päätavoitteista ja niiden toteutumisesta:

- Toimiva ja selkeä tapa löytää rautapohjaiset viat prototyyppilaitteista – rautapohjaiset viat löytyivät helposti sillä ne esiintyivät vain yksittäisellä laitteella, kun taas ohjelmistopohjaiset viat esiintyvät kaikilla laitteilla, sillä kaikissa on käytössä samat ohjelmistopohjaiset virheet sisältävä ohjelmistoversio.
- Konseptin käytännön toteutuksen oli oltava mahdollisimman nopeasti suoritettavissa – yhden laitteen testiajonkesto oli noin 12 tuntia, mikä riittää rautapohjaisten vikojen löytymiseen. Pidempi referenssiajo osoitti, että ajokeston kasvaessa myös todennäköisyys ohjelmistopohjaisten virhetilanteiden esiintymiseen kasvaa. Rautapohjaisia vikoja ei tullut kuitenkaan enempää esiin.
- Konseptin käytännön toteutuksen oli oltava mahdollisimman pienellä työmäärällä toteutettavissa – suurin osa konseptiin kuluvasta työmäärästä menee testitapauksien ja asetusskriptin skriptaamiseen, minkä jälkeen laitteiden testiajoon laittaminen tapahtuu nopeasti samoja skriptejä käyttämällä.

7.2 Työn onnistuminen

Testauskonseptin suunnittelu ja käytännön toteutus onnistuivat melko hyvin, sillä löydettiin käytettävissä olevin keinoin toimivin ja selkein tapa erottaa rautapohjaiset viat ohjelmistopohjaisista virhetilanteista. Pientä hienosäätöä vielä toki tarvittaisiin konseptin toimivuuden optimointiin, mutta saatujen tuloksien perusteella ollaan selkeästi oikeilla jäljillä. Konseptin käytännön toteutuksen suorittaminen onnistui myös hyvin lyhyessä ajassa, sillä laitekohtaisesti melkein kaksi vuorokautta kestänyt referenssiajo osoitti, ettei pidemmällä ajokestolla ole vaikutusta rautapohjaisten vikojen löytämiseen. Konseptin toteutukseen tarvittava työmäärä pysyi suhteellisen pienenä, koska jokaiselle proto-

tyyppilaitteen malliversiolle ei tarvitse tehdä kuin kerran yhdet skriptit, minkä jälkeen niitä voidaan käyttää jokaisen laitteen testiajossa.
Myös suhteellisen yksinkertaiset testitapaukset pitivät skriptaukseen kuluvan työmäärän pienenä, monimutkaisia testitapauksia taas ei ollut järkevää suunnitella, koska ohjelmistopohjaisten virhetilojen esiintymisen todennäköisyys kasvaa testitapauksien monimutkaistuessa.

Koska konseptista tuli vaatimuksien mukaisesti nopea ja vähällä työmäärällä suoritettavissa oleva kokonaisuus, on myös virallinen käyttöönotto mahdollista, sillä sen suorittaminen ei kuluta liikaa resursseja ohjelmistopohjaiselta testaukselta. Tämä parantaisi testaustulosten laatua ja toistettavuutta sekä varmistaisi sen, ettei rikkinäisiä prototyyppilaitteita ainakaan käytettäisi suorituskykytestaukseen. Viallisten prototyyppilaitteiden löytäminen ennen virallista testausta vähentäisi mahdollisuuksia löytää rikkinäisiä laitteita virallisen ohjelmistotestauksen yhteydessä, eikä näin ollen tarvitsisi kuluttaa työtunteja viallisella laitteella jo testattujen tuloksien uudelleen testaukseen. Konseptin virallinen käyttö antaisi myös osittaisen kuvan testattavan prototyyppimallin yleisestä kypsyydestä, sekä testauksessa käytettävän ohjelmistoversion yleisestä tasosta ennen virallisen ohjelmistotestauksen aloittamista.

LÄHTEET

Black, R. 2007. Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional. Indianapolis, Indiana: Wiley Publishing, Inc.

Digia Oyj. 2008. Digia Oyj:n viralliset verkkosivut. Viitattu 13.11.2008. http://www.digia.com/C2256FEF0043E9C1/0/405000098?opendocument&lang=fi.

Digia Oyj. 2009. Digia Oyj:n viralliset verkkosivut. Viitattu 30.3.2009. http://www.digia.com/C2256FEF0043E9C1/0/405000761?opendocument&lang=fi

Hambling, B., Morgan, P., Samaroo, A., Thompson, G. & Williams, P. 2007. Software Testing An ISEB Foundation. korj. p. Edinburg, Scotland: CAPDM Limited.

Li, K. & Wu, M. 2004. Effective Software Test Automation: Developing an Automated Software Testing Tool. Alameda: SYBEX Inc.

Pääkkönen, P. 2005. Toiminnallisen testauksen automatisointi. Opinnäytetyö. Jyväskylän ammattikorkeakoulu, Informaatioteknologian instituutti, tekniikka ja liikenne.