**Mari Kasanen**

# Automatisoitu GUI-testaus mobiiliapplikaatioissa

Tietotekniikan kandidaatintutkielma

2. toukokuuta 2017

Jyväskylän yliopisto

Tietotekniikka

**Tekijä:** Mari Kasanen

**Yhteystiedot:** r.mari.s.kasanen@student.jyu.fi

**Työn nimi:** Automatisoitu GUI-testaus mobiiliapplikaatioissa

**Title in English:** Automated GUI testing in mobile apps

**Työ:** Kandidaatintutkielma

**Sivumäärä:** 26+0

**Tiivistelmä:** Mobiiliapplikaatioiden määrä on kasvanut viime vuosina paljon, joten niiden laadunvalvonta on tärkeää. Tässä tutkielmassa etsitään automatisoidusta GUI-testauksesta piirteitä, jotka auttaisivat mobiilitestausta kohtaamaan alan kasvavat vaatimukset. Lupaavimpia ratkaisuja tarjoavat testityökalut, jotka yhdistävät erilaisia testaustapoja tai toimivat useilla alustoilla.

**Avainsanat:** Android, GUI-testaus, automatisointi, mobiilitestaus

**Abstract:** The number of mobile applications has grown much over the past years, making quality control for apps important. In this thesis traits that help mobile testing face the growing demands of the industry are looked for from automated GUI testing. The test tools that seem most promising combine different ways of testing or work on multiple platforms.

**Keywords:** Android, GUI testing, automated, mobile testing

# Taulukot

# Sisältö

# 1 Johdanto

Älylaitteiden ja erityisesti niille kehitettyjen mobiiliapplikaatioiden määrän kasvaessa nousee tärkeäksi tavoitteeksi kehittää uusia testityökaluja (Gao ym. 2014). Koska manuaalinen testaus on usein tehotonta (Azim ja Neamtiu 2013), parempaa testaustapaa etsitään automatisoidusta testauksesta.

Automatisoidussa graafisen käyttöliittymän (GUI, graphical user interface) testauksessa simuloidaan käyttäjätapahtumia, jonka jälkeen tarkistetaan oliko applikaation toiminta sellaista mitä odotettiin (Lin, Rojas ym. 2014). Tarkistus voidaan esimerkiksi tehdä testattavan laitteen kuvakaappauksia vertailemalla, kuten testityökalu SPAG:ssä (Lin, Chu ym. 2014), tai vertaamalla applikaation tilaa aikaisemmin GUI:sta tehdyn mallin tilaan, kuten MobiGUITARissa (Amalfitano ym. 2015) tai SwiftHandissä (Choi, Necula ja Sen 2013).

Tutkielman tavoitteena on esitellä erilaisia GUI-testausmenetelmiä, pohtia testauksen automatisointiin liittyviä kysymyksiä, analysoida kehitettyjä testaustapoja kriittisesti ja löytää niistä laadukkaita ja yleistettäviä piirteitä, jotka veisivät mobiilitestauksen alaa eteenpäin. Tutkielma toteutetaan kuvailevana kirjallisuuskatsauksena.

Mobiilitestauksen tutkimus keskittyy markkinoiden enemmistön käyttämään mobiilikäyttöjärjestelmään, joka on jo vuosia ollut Android. GUI:ta testaamalla voidaan löytää puutteita GUI:sta sekä muualta applikaatiosta (Wen ym. 2015), ja mobiiliapplikaatioiden monimutkaisuus piilee niiden graafisissa käyttöliittymissä (Choi, Necula ja Sen 2013). Siksi GUI-testaus on luonnollinen tapa mobiiliapplikaatioiden toimivuuden testaamiseen. Näistä syistä tutkielma keskittyy Android-käyttöjärjestelmissä tapahtuvaan GUI-testaukseen ja sen automatisoimiseen.

Luvussa kaksi esitellään GUI-testausta ja kerrotaan yleisellä tasolla, minkälaisia osia GUI-testaukseen kuuluu. Tässä luvussa listataan myös haasteita, jotka ovat mobiilitestaukselle ominaisia. Tämän jälkeen kolmannessa luvussa käsitellään GUI-testauksen automatisointia: mitkä osiot voidaan automatisoida ja miten. Neljännessä luvussa analysoidaan olemassaolevia GUI-testaustapoja. Lopussa tehdään yhteenveto ja listataan lähteet.

# 2 GUI-testaus

GUI-testauksessa ohjelmistotestausta tehdään pääasiassa graafisen käyttöliittymän kautta. Kuten muussakin ohjelmistotestauksessa, GUI-testauksessa tarkoituksena on joko löytää vikoja (bugs) tai todeta applikaation toimivan tarkoituksenmukaisesti. Testi voidaan tehdä manuaalisesti applikaation graafista käyttöliittymää käsittelemällä tai automatisoidusti, eli esimerkiksi simuloimalla käyttäjätapahtumia ja syöttämällä tapahtumat testattavalle laitteelle. Automatisoidusta GUI-testauksesta kerrotaan lisää luvussa 3.

GUI-testauksessa on mahdollista käyttää erilaisia algoritmeja eri tarpeisiin. Jos tavoitteena on applikaation systemaattinen läpikäyminen, voi hyvä algoritmi olla syvyyshaku, kun taas applikaation kaatumisen tutkimiseen voi riittää satunnaistestaus.

Luvussa 2.1 esitellään GUI-testauksen eri osia, ja luvussa 2.2 kerrotaan, minkälaisia haasteita nimenomaan mobiilitestaus kohtaa.

## 2.1 GUI-testauksen osat

GUI-testaus koostuu yleisesti ottaen kahdesta vaiheesta, joista ensimmäisessä tutkitaan ja joissain tapauksissa mallinnetaan testattavaa GUI:ta. Toisessa vaiheessa tarkastetaan, että applikaatio toimii odotetusti. Jotkin testityökalut tosin yhdistävät näitä vaiheita — esimerkiksi Amalfitano, Fasolino, Tramontana, Carmine ja Memon (2012) tekivät testityökalussaan näin. Tässä alaluvussa esitellään ensimmäiseen vaiheeseen liittyvää mallinnusta, toiseen vaiheeseen liittyvää todennusta ja viimeiseksi tallennus ja toisto -testaustapaa (record-and-replay). Tässä testaustavassa tallennus kuuluu GUI:n tutkimisvaiheseen ja toisto tarkastusvaiheeseen.

Mallinnuksessa luodaan tutkittavasta applikaatiosta malli, jota voidaan esittää esimerkiksi kaavioina tai äärellisinä automaatteina. Malliin sisältyy tieto erilaisista näkymistä (app screens) ja usein lista näkymässä sallituista eleistä (gestures) tai tapahtumista (events). Olennainen osa mallinnusta on se, että mallista tehdään abstrahoitu kuva applikaatiosta. Täten mallissa tärkeää ei ole GUI-elementtien sisältö, kuten väri tai teksti, vaan se voidaan jättää

huomiotta (Choi, Necula ja Sen 2013).

Todentamisessa verrataan testin tulosta aikaisemmin nauhoitettuun tai tiedossa olevaan tilaan, jossa applikaation pitäisi testin jälkeen olla. Tähän voidaan käyttää applikaatiosta tehtyä mallia. Toinen mahdollisuus todentamiseen on käyttää kuvia: tarkasteluvaiheessa otettuja kuvia verrataan testin aikana otettuihin kuviin. Näin voidaan kuvien perusteella tarkastaa, onko applikaatio päätynyt testissä oikeaan tilaan. Kuvina on mahdollista käyttää testattavan laitteen kuvakaappauksia (Lin, Chu ym. 2014) tai ulkoisen kameran ottamia kuvia (Lin, Rojas ym. 2014).

Tallennus ja toisto -testaustavan ensimmäisessä osassa, tallennuksessa, nauhoitetaan käyttäjää käyttämässä GUI:ta. Käyttäjän tuottamat tapahtumat nauhoitetaan matalan tason tapahtumavirrasta (low level event feed) (Gomez ym. 2013), ja tallennetaan lokitiedostoon. Tällöin loki sisältää tiedon siitä, mihin kohtaan laitteen näyttöä käyttäjä on koskenut milläkin hetkellä. Testaustavan toisessa osassa, toistossa, käytetään tallennusvaiheessa luotuja lokitiedostoja ja nimensä mukaisesti toistetaan nauhoitettua tapahtumajälkeä (event trace) osittain tai kokonaan. Tapahtumat syötetään testattavalle laitteelle lokin mukaisesti. Toistossa tarkoitus on saada applikaatio käyttäytymään samalla tavalla kuin tallennuksessa. Tämän testaustavan huonoin puoli on se, että tallennusvaihe täytyy tehdä uudestaan aina kun GUI:ta muutetaan (Linares-Vásquez 2015).

## 2.2 Mobiilitestauksen haasteet

Monet asiat tekevät mobiiliapplikaatioiden testauksesta haastavaa. Yksi näistä asioista on mobiiliapplikaatioiden luonne — niiden tulee toimia kaikkialla ja eri käyttöjärjestelmien, näytönkokojen, laskentatehojen ja akunkestojen kanssa (Gao ym. 2014). Mobiilitestauksessa tulee siis ottaa huomioon monia sellaisia asioita, joita tietokoneohjelmien testauksessa ei. Esimerkiksi osa applikaatioista reagoi laitteen kontekstin muutoksiin, kuten puhelun tuloon, laitteen heiluttamiseen ja sijainnin muutokseen. Tällaisia applikaatioita testatessa työkalu, joka kykenee simuloimaan kontekstinmuutoksia, pystyy testaamaan suuremman osan applikaatiosta kuin yksinkertaisempi työkalu (Amalfitano ym. 2013). Tästä huolimatta kontekstinmuutosten simulointi ei ole testityökaluissa yleistä.

Tietokoneisiin verrattuna mobiililaitteiden pieni laskentateho voi vaikeuttaa täysin natiivia testausta (kokonaan laitteessa tapahtuvaa testausta). Applikaation ja testityökalun suorittaminen yhtäaikaisesti laitteessa ei aina ole järkevä vaihtoehto, ja voi hidastaa laitetta niin ettei testi anna luotettavaa kuvaa applikaatiosta. Vain harvat testityökalut suoritetaan täysin natiivina. Aiheesta keskustellaan lisää luvussa 4.2.

Linares-Vásquez (2015) kritisoi applikaation systemaattista tutkimista siitä, että se ei edusta luonnollista käyttöä. Luonnollisella käytöllä viitataan tässä siihen, miten käyttäjät keskimäärin toimivat applikaatioita käyttäessä: usein käyttöaika applikaation avaamisesta sen sulkemiseen on lyhyt, eikä kaikkia toimintoja käytetä. Tulisiko siis testata kaikki koodi, vai tuleeko tyytyä testaamaan vain usein käytetyt osat applikaatiosta? Takala, Katara ja Harty (2011, s. 383) taas ehdottavat, että sellaisen testin läpäisemistä, joka käy läpi kaikki applikaation toiminnat, siirtymät, tilat ja näiden yhdistelmät, voitaisiin tulevaisuudessa pitää vaatimuksena ohjelmistoversion julkaisulle.

Gao ym. (2014) mukaan mobiilitestauksen automatisoinnissa on kaksi suurta ongelmaa: mobiilitestauksen infrastruktuuria ja skriptikieliä ei ole standardoitu, eikä ole yhtenäistä automatisointiratkaisua ja -infrastruktuuria joka toimisi useimpien mobiililaitteiden alustoilla ja selaimilla.

Nämä haasteet tuovat osaltaan motivaatiota testauksen automatisoinnille, sillä manuaalisessa testauksessa testaajan eli ihmisen tulisi muistaa ne kaikki ja osata ottaa ne huomioon testeissä. Nämä asiat täytyy ratkaista testityökalua kehitettäessä.

# 3 Automatisoitu GUI-testaus

Automatisoitu GUI-testaus pyrkii paitsi testaamaan applikaatiota GUI:n avulla, myös automatisoimaan osan testiprosessista tai koko testiprosessin. Manuaalinen testaus voi olla työlästä ja aikaavievää, puhumattakaan sen tehottomuudesta. Automatisoitu testaus houkuttaa parempien mahdollisuuksiensa ansiosta.

Azim ja Neamtiu (2013, s. 644-646) tekivät seitsemän käyttäjän tutkimuksen 25:llä applikaatiolla, jossa huomattiin normaalin käytön aikana löytyvän vain pienen osan applikaation näkymistä ja metodeista. Testikäyttäjät löysivät keskimäärin noin 30% applikaation näkymistä ja vain noin 6% lähdekoodin metodeista. Automaattiset testaustavat pyrkivät testaamaan normaalia käyttöä huomattavasti suuremman osan metodeista.

Luvussa 3.1 käsitellään mallinnuksen automatisointia ja luvussa 3.2 testitapausten luomista ja suorittamista automaattisesti. Viimeisenä luvussa 3.3 käsitellään siirtokopiointia.

## 3.1 Mallinnuksen automatisointi

Mallinnuksen automatisointiin on useita tapoja. Yksi näistä on GUI:n siirtokopiointi (GUI ripping). Siirtokopioinnin yhteydessä voidaan tehdä muutakin kuin mallintaa applikaatiota, joten aihetta käsitellään erikseen luvussa 3.3.

Muita tapoja mallintamiseen ovat esimerkiksi staattinen analyysi ja koneoppiminen (machine learning). Staattisella analyysillä voi selvittää näkymien välisiä suhteita, ja lisätä malliin niiden välille siirtymän, jos näkymästä pääsee toiseen (Azim ja Neamtiu 2013). Koneoppimista voidaan käyttää kun verrataan näkymää aiemmin löydettyihin näkymiin (Choi, Necula ja Sen 2013). Jos näkymä on tarpeeksi samankaltainen jonkin aiemmin löydetyn näkymän kanssa, ne yhdistetään mallissa. Muutoin malliin lisätään uusi näkymä.

Mallinnettaessa applikaatiota täytyy pyrkiä siihen, että malli sisältää koko GUI:n. Tästä huolehditaan usein siten, että GUI:n läpikäyminen hoidetaan systemaattisesti eli esimerkiksi aloittamalla applikaation aloitusnäkymästä ja tutkimalla kaikki sen siirtymät ja sitten tekemällä sama näkymille, joihin aloitusnäkymästä päästiin. Testityökalu voi käyttää erilaisia

algoritmeja päättääkseen, missä järjestyksessä siirtymät ja näkymät käydään läpi.

Testityökalusta riippuu myös, millaisessa muodossa malli esitetään. Joitakin esimerkkejä ovat Choi, Necula ja Sen (2013) käyttämä laajennettu deterministinen merkitty siirtymäjärjestelmä (extended deterministic labeled transition system), Azim ja Neamtiu (2013) käyttämä staattinen näkymien siirtymäkaavio (static activity transition graph) ja Amalfitano ym. (2015) käyttämä tilakone (state machine).

Kaikki eivät ole sitä mieltä, että mallinnusta tulisi automatisoida. Takala, Katara ja Harty (2011) väittävät, että manuaalinen mallinnus on tärkeä osa testausta, ja sen aikana voi löytää jo suuren osan applikaation vioista. Heidän mukaansa manuaalisen mallinnuksen monet hyödyt katoavat, kun prosessia yritetään automatisoida.

## 3.2 Testitapausten luomisen ja suorittamisen automatisointi

Tämän tutkielman puitteissa mielletään automaattisesti luoduiksi testitapauksiksi skriptitiedostojen lisäksi sellaiset mallin perusteella luodut testit, joihin testityökalun käyttäjä ei ole vaikuttanut manuaalisesti.

Skriptitiedostojen käyttäminen voi olla hyödyllistä esimerkiksi rinnakkaistestauksen toteuttamisessa. Wen ym. (2015) tekevät testitapauksistaan skriptitiedostoja. Näin testi on mahdollista suorittaa muualla kuin skriptin luoneessa koneessa.

Testitapauksia voi luoda applikaatiosta tehdyn mallin perusteella. Tällöin malliin sisältyviä tietoja käytetään hyväksi ja voidaan testityökalun algoritmin tai heuristiikkojen perusteella valita ele tai tapahtuma, jolla päästään näkymästä seuraavaan. Usein yksi testitapaus sisältää siirtymiä GUI:n aloitusruudusta johonkin ennaltamääriteltyyn näkymään, josta ei enää pääse eleillä uuteen näkymään. Kun tähän umpikujanäkymään on päästy, testitapaus on suoritettu ja voidaan siirtyä seuraavaan tapaukseen.

Vaihtoehtoisesti kerralla voidaan pyrkiä käymään suurempi osa GUI:sta läpi (Choi, Necula ja Sen 2013). Kun ollaan päädytty umpikujanäkymään, voidaan palata näkymissä taaksepäin ja jatkaa GUI:n tutkimista muualta. Näkymissä palaaminen on yksinkertaista, kun simuloidaan laitteen takaisin- (back button) tai kotinäppäimen (home button) painamista. Tällöin

yksittäinen testitapaus voi olla huomattavasti suurempi.

Testitapausten luomiseen on myös mahdollista käyttää valmista JUnit-työkalua. Androidin SDK:hon (software development kit) sisältyvä JUnit tekee lähdekoodin perusteella jokaiselle applikaation näkymälle oman testitapauksen. Tätä työkalua käyttävät testaamisessa apunaan mm. Hu ja Neamtiu (2011) ja Amalfitano, Fasolino, Tramontana, Carmine ja Imparato (2012).

Kun testitapaukset on luotu, niiden määrä voi olla korkea erityisesti jos tarkoituksena on testata kaikki applikaatiosta löytyvät metodit ja/tai applikaatio on suuri. Toisaalta yhden testitapauksen käsittelyssä voi kestää kauan, jos kerralla yritetään tutkia suuri osa applikaatiosta. Molemmissa tapauksissa testitapausten suoritus kannattaa automatisoida, sillä manuaalinen puuttuminen niiden suorittamiseen vain hidastaisi testausprosessia.

## 3.3 Siirtokopiointi

GUI:n siirtokopiointi (GUI ripping) on lähes kokonaan tai kokonaan automatisoitu prosessi, jonka aikana applikaatiota suoritetaan jäsennellysti. Siirtokopioinnin tarkoitus on takaisinmallintaa (reverse engineer) GUI iteratiivisesti. Tätä menetelmää on aiemmin käytetty onnistuneesti tietokone- ja webapplikaatioiden testaukseen (Amalfitano, Fasolino, Tramontana, Carmine ja Imparato 2012).

Siirtokopioinnissa suoritettavan takaisinmallinnuksen idea Amalfitano ym. (2015) mukaan on seuraavanlainen: applikaatio avataan aloitusnäkymään ja testityökalu hankkii itselleen listan kaikista tapahtumista, joita tässä näkymässä voi tehdä. Tämä lista lisätään suurempaan tehtävälistaan. Siirtokopiointityökalu (ripper) suorittaa yksitellen tehtävälistan tapahtumia ja sitten poistaa ne listasta. Tapahtuman viedessä uuteen näkymään hankitaan lista uuden näkymän mahdollisista tapahtumista ja lisätään se tehtävälistaan. Näin jatketaan, kunnes tehtävälista on tyhjä. Tällöin kaikki näkymät ja tapahtumat, joihin siirtokopiointityökalu pääsee, on käyty läpi.

Jotkin siirtokopiointityökalut tekevät yhtä aikaa muutakin kuin suorittavat applikaatiota. Esimerkiksi Amalfitano, Fasolino, Tramontana, Carmine ja Memon (2012) kehittämä testityö-

kalu luo ja samalla suorittaa testitapauksia havaiten ajonaikaiset kaatumiset. On myös tavallista, että siirtokopioinnin ohessa testityökalu pitää yllä jonkinlaista mallia GUI:sta. Malli voi olla esimerkiksi äärellinen automaatti (finite state machine), tapahtumavirtakaavio (EFG, event-flow graph) tai GUI-puu (GUI tree) (Amalfitano, Fasolino, Tramontana, Carmine ja Imparato 2012). Osa työkaluista palauttaa tekemänsä mallin ja osa vain käyttää sitä testauksen ajan.

Löytyy myös testityökaluja, jotka käyttävät GUI:n siirtokopiointia hyväksi osana omaa testaustapaansa. Tällaisia ovat esimerkiksi BugRocket (Ma ym. 2016), A$^3$E (Azim ja Neamtiu 2013) ja PATS (Wen ym. 2015). BugRocketin lisä normaaliin siirtokopiointiin on se, että se osaa käyttää käyttäjäkohtaista dataa. PATS taas luo väliaikaisia tapahtumaketjuja (short-term testing event sequences), joita orjapalvelimet lähettävät isäntäpalvelimelle koottavaksi suuremmiksi kokonaisuuksiksi. A$^3$E ei lisää siirtokopiointiin erikoisuuksia, mutta käyttää sitä osana testaustaan.

# 4 Erilaisia testauskehyksiä

Tässä luvussa esitellään ja vertaillaan eri näkökulmista testityökaluja ja -ympäristöjä. Tarkasteltavia testityökaluja on seitsemäntoista ja ne on koottu taulukkoon 1. Taulukossa kerrotaan testityökalulle annetun nimen lisäksi mistä lähteestä työkalu on löytynyt. Luvussa viitataan usein testityökaluihin pelkillä nimillä. Testityökalujen esittelyissä keskitytään hyviin tai ominaisuuksiltaan muista poikkeaviin ratkaisuihin, eikä kaikkia työkaluja käsitellä yksityiskohtaisesti.

Taulukossa ”tapa”-sarake kertoo, käyttääkö testityökalu jotakin luvussa 4.1 käsiteltävistä kolmesta testaustavasta: malliin perustuva testaus (malli), tallennus ja toisto (T&T) tai siirtokopiointi (siirto). Jos tämä sarake on tyhjä, tarkoittaa se sitä, että testityökalu ei täysin noudata näitä testaustapoja, mutta mahdollisesti omaa silti piirteitä jostakin niistä.

Luvussa 4.1 esitellään testityökaluja eri testaustapojen avulla. Tämän jälkeen luvussa 4.2 vertaillaan emulointia ja natiivitestausta. Testausta nopeuttavia menetelmiä on koottu lukuun 4.3. Luvussa 4.4 keskustellaan lähdekoodin käytöstä GUI-testauksessa ja siitä, voiko automatisoidusti testata kaiken tärkeän.

## 4.1 Testaustapoja ja testityökaluja

Tässä alaluvussa tarkastellaan ensin lähemmin kolmea testaustapaa ja niitä käyttäviä testityökaluja. Luvussa 4.1.4 keskitytään testaustapoja yhdisteleviin työkaluihin ja luvussa 4.1.5 käsitellään useilla alustoilla testaamista.

### 4.1.1 Malliin perustuva testaus

Malliin perustuvassa testauksessa (model-based testing) GUI:sta tehtyä mallia hyödynnetään applikaation testauksessa. Mallinnuksesta kerrottiin erikseen luvussa 2.1. Koska malli on abstraktimpi kuin oikea GUI, ei pienien muutoksien takia tarvitse tehdä mallia uudestaan. Tämä tekee malliin perustuvasta testauksesta kestävän ratkaisun, vaikka suuret muutokset vaativat mallin uudelleen tekemisen. Takala, Katara ja Harty (2011) väittävätkin, että mallia

Taulukko 1. Testityökalut lähteen mukaan aakkostettuna. Nimi-sarakkeessa ”-” tarkoittaa sitä, että testityökalulle ei ole annettu nimeä.

| # | Nimi | Tapa | Lähde |
|---|---|---|---|
| 1 | Extended Ripper | siirto | Amalfitano ym. 2013 |
| 2 | GUI Ripper | siirto | Amalfitano, Fasolino, Tramontana, Carmine ja Imparato 2012 |
| 3 | AndroidRipper | siirto | Amalfitano, Fasolino, Tramontana, Carmine ja Memon 2012 |
| 4 | MobiGUITAR | siirto | Amalfitano ym. 2015 |
| 5 | $A^3E$ | siirto, T&T, malli | Azim ja Neamtiu 2013 |
| 6 | SwiftHand | malli | Choi, Necula ja Sen 2013 |
| 7 | RERAN | T&T | Gomez ym. 2013 |
| 8 | Mobilette | | Grønli ja Ghinea 2016 |
| 9 | - | | Hu ja Neamtiu 2011 |
| 10 | UGA | T&T, malli | Li ym. 2014 |
| 11 | SPAG | T&T | Lin, Chu ym. 2014 |
| 12 | SPAG-C | T&T | Lin, Rojas ym. 2014 |
| 13 | T+ | T&T, malli | Linares-Vásquez 2015 |
| 14 | BugRocket | siirto | Ma ym. 2016 |
| 15 | MobTAF | | Nagowah ja Sowamber 2012 |
| 16 | - | malli | Takala, Katara ja Harty 2011 |
| 17 | PATS | siirto | Wen ym. 2015 |

on helpompi ylläpitää kuin suurta määrää testiskriptejä.

Malliin perustuvaa testausta käyttävät SwiftHand (Choi, Necula ja Sen 2013) ja nimetön testityökalu, jonka ovat luoneet Takala, Katara ja Harty (2011). SwiftHand käyttää koneoppimista mallin tekemiseen ja tarkkailee applikaatiota testien ajon aikana. Näin SwiftHand kykenee hallitsemaan applikaation uudelleenkäynnistyksien määrää. Aiheesta ja SwiftHandistä keskustellaan lisää luvussa 4.3. Nimetön testityökalu taas luottaa TEMA-työkaluilla tehtyyn manuaaliseen mallinnukseen. Mallin valmistuttua se suorittaa testejä, jotka kattavat kaikki mallin toiminnat. Jos nämä testit epäonnistuvat, voi mallissa olla virheitä. Mallin eheydestä varmistuttuaan testityökalu suorittaa satunnaistestejä, joiden aikana siirtymät valitaan mielivaltaisesti.

Siirtokopiointityökalut tekevät myös applikaatiosta mallin. Muiden erojen takia siirtokopiointityökaluja käsitellään erikseen luvussa 4.1.3.

### 4.1.2 Tallennus ja toisto

Tallennus ja toisto -testaustavassa nauhoitetaan applikaation manuaalista käyttöä. Testaustavan vaiheita esiteltiin luvussa 2.1. Nauhoitettuja tapahtumajälkiä voidaan toistovaiheessa toistaa esimerkiksi nopeutettuna. GUI:n muuttuessa tallennusvaihe täytyy tehdä uudelleen. Kyseessä ei siis ole kestävä testaustapa, jos testattava applikaatio on kehityksensä alkuvaiheissa tai GUI muuttuu muista syistä.

Tätä testaustapaa käyttäviä testityökaluja ovat RERAN, SPAG ja SPAG-C. RERAN (Gomez ym. 2013) teki uraauurtavaa työtä käyttäessään monimutkaisten eleiden (kuten swipe) simuloimiseen tallennusvaiheen lokitiedostoja. Tämä mahdollisti testeissä pääsyn sellaisiin applikaation näkymiin, joihin oli aiemmin vaikeaa tai mahdotonta päästä. RERAN:iä käyttävät monet sen jälkeen kehitetyt testityökalut, kuten $A^3E$ ja UGA. Lisäksi löytyy testikehyksiä, kuten T+, jotka käyttävät osittain samankaltaisia tekniikoita.

SPAG (Lin, Chu ym. 2014) keskittyy toistovaiheen todennukseen, johon se käyttää testattavan laitteen kuvakaappauksia. Kuvakaappaukset lähetetään testiä valvovalle tietokoneelle verrattavaksi tallennusvaiheen vastaaviin kuvakaappauksiin. SPAG:iä käyttävä SPAG-C (Lin, Rojas ym. 2014) käyttää todennukseen ulkoisen kameran ottamia kuvia. Todentamisen tark-

kuutta parantamalla SPAG ja SPAG-C pyrkivät testityökalun luotettavuuden parantamiseen.

### 4.1.3 Siirtokopiointityökalut

Siirtokopioinnissa suoritetaan applikaatio jäsennellysti ja yleensä samalla rakennetaan GUI:sta malli. Siirtokopioinnin tekniikoista keskusteltiin tarkemmin luvussa 3.3. Siirtokopiointityökaluja ovat GUI Ripper, AndroidRipper, Extended Ripper ja MobiGUITAR. Työkalut on lueteltu niiden julkaisujärjestyksessä.

GUI Ripper (Amalfitano, Fasolino, Tramontana, Carmine ja Imparato 2012) ja AndroidRipper (Amalfitano, Fasolino, Tramontana, Carmine ja Memon 2012) ovat samankaltaisia: siirtokopiointiprosessi on samanlainen, molempien parametrejä (kuten aikaviive suoritettujen tapahtumien välillä) voi muuttaa ja molemmat luovat malliksi GUI-puun. Merkittävin ero on se, että toisin kuin AndroidRipper, GUI Ripper palauttaa luomansa mallin. Molempien testityökalujen siirtokopiointiprosessin kesto on pitkä. GUI Ripperillä kesti applikaatiosta riippuen 2–3 tuntia ja AndroidRipperillä 4,5–5 tuntia.

Extended Ripper (Amalfitano ym. 2013) kykenee simuloimaan älylaitteen kontekstinmuutoksia (kuten laitteen heilutus ja sijainnin muutos). Extended Ripperiä voidaan ajatella AndroidRipperin laajennuksena, joka siirtokopioinnin lisäksi simuloi kontekstinmuutoksia saavuttaakseen suuremman osan applikaation metodeista. Testeissä AndroidRipperiä verrattiin Extended Ripperiin viiden applikaation avulla. Jokaisen applikaation kohdalla Extended Ripper saavutti saman tai korkeamman koodirivi- ja metodimäärän. Kontekstinmuutokset laukaisevat applikaatioissa tapahtumankäsittelijöitä (event handlers), jos niitä on applikaatioon toteutettu. Extended Ripper tarjoaa mahdollisuuden testata myös tapahtumankäsittelijöiden aikaansaamia GUI:n muutoksia.

MobiGUITARin (Amalfitano ym. 2015) tarkoitus on tarjota täysin automatisoitua testausta, joka toimii mobiilialustojen kiristyneiden turvallisuusmenettelyjen kanssa. MobiGUITAR tekee applikaation mallista tilakoneen, sillä kehittäjien aiemmin käyttämät EFG:t eivät reagoineet tilanmuutoksiin eivätkä sovi modernien mobiiliapplikaatioiden mallintamiseen. Tämä testityökalu palauttaa mm. kaatumisraportin, tilakonemallin ja tapahtumaketjut, jotka aiheuttivat applikaation kaatumisen. MobiGUITARia testattiin neljällä applikaatiolla, joista se

löysi yhteensä kymmenen vikaa. Verrattuna kahteen muuhun testityökaluun (Monkey ja Dynodroid), todettiin MobiGUITARin löytävän enemmän vikoja.

Yllämainittujen lisäksi A$^3$E, BugRocket ja PATS käyttävät siirtokopiointia osana testaustapojaan. Luvussa 3.3 kerrottiin, mitä nämä testityökalut yhdistävät siirtokopiointiin. A$^3$E esitellään luvussa 4.1.4 ja PATS sekä BugRocket luvussa 4.3.

### 4.1.4 Testaustapoja yhdistelevät työkalut

Monet testityökalut keskittyvät tietyn ongelman ratkaisemiseen. Tällöin kehitetty ratkaisu voi toisenlaisissa tilanteissa suoriutua huonosti. Testaustapoja yhdistelemällä luotu testityökalu voi täten selvitä useammista ongelmista ja olla monikäyttöisempi. Esimerkiksi T+ (Linares-Vásquez 2015) on testauskehys, joka yhdistää mallien käyttöä ja tallennus ja toisto -tapaa. T+ tallentaa RERAN:in kaltaisesti applikaation luonnollista käyttöä ja analysoi syntyneitä lokitiedostoja luodakseen automaattisesti testitapausehdokkaita. Ehdokkaita voidaan luoda myös T+:lle syötettyjen mallien avulla. Muita testaustapoja yhdisteleviä työkaluja ovat esimerkiksi A$^3$E ja UGA.

A$^3$E (Azim ja Neamtiu 2013) yhdistää malliin perustuvaa testausta, siirtokopiointia ja osittain tallennus ja toisto -testaustapaa. Kaksi erilaista systemaattisen tutkimisen tapaa, kohdennettu tutkiminen (targeted exploration) ja syvyys ensin -tutkiminen (depth-first exploration), tekevät A$^3$E:stä monipuolisen. Kohdennetussa tutkimisessa tarkoituksena on saavuttaa nopeasti kaikki näkymät ja syvyys ensin -tutkimisessa systemaattisesti tutkia applikaation tiloja. Molemmat tutkimistavat vaativat GUI-elementtien poimimista (extract) ja suorittamista (exercise). Tähän käytetään siirtokopiointia. A$^3$E tekee applikaatiosta staattisen näkymien siirtymäkaavion, jota se käyttää kohdennetun tutkimisen lähtökohtana. Syvyys ensin -tutkiminen voidaan suorittaa ilman siirtymäkaaviota. Tutkiminen tallennetaan käyttäen RERAN:iä, tarkoituksena mahdollistaa virheenkorjaus (debugging) ja tapahtumajälkien toistaminen myöhemmin esimerkiksi applikaation kaatumisen toistamiseksi. Applikaation tutkimisen helpottamiseksi Azim ja Neamtiu (2013) tekivät elekirjaston, joka sisältää eri suuntaisia swipe-eleitä ja vierityksen (scrolling). A$^3$E tukee myös ulkoisten sensoreiden, kuten mikrofonin ja GPS:n, tapahtumia.

A$^3$E:tä testattiin 25:llä applikaatiolla verraten näkymien ja metodien saavuttamista normaaliin manuaaliseen käyttöön. Kohdennettu tutkiminen saavutti keskimäärin 34,03% enemmän näkymiä ja 23,07% enemmän metodeja, kun taas syvyys ensin -tutkiminen saavutti 29,31% enemmän näkymiä ja 30% enemmän metodeja. Siirtymäkaavion muodostamisessa kesti keskimäärin 74 sekuntia ja kohdennetussa tutkimisessa 87 minuuttia. Syvyys ensin -tutkimisessa kesti keskimäärin 104 minuuttia.

Toinen testaustapoja yhdistelevä testityökalu UGA (Li ym. 2014) käyttää useita aiemmin julkaistuja testityökaluja sisältäen tallennus ja toisto -testaustavan osia, satunnaistestausta ja malliin perustuvaa systemaattista testausta. Tarkasteltavista testityökaluista UGA käyttää RERAN:iä, A$^3$E:tä ja nimetöntä työkalua (Hu ja Neamtiu 2011). UGA etsii tallennusvaiheen lokista pysähdyspaikkoja. Pysähdyspaikoiksi valitaan sellaiset tapahtumat, joita ennen käyttäjä on ollut useita sekunteja tekemättä mitään ja tapahtumat, jotka johtavat uuteen näkymään. Toistovaiheessa pysähdyspaikan kohdattuaan UGA vaihtaa automaattiseen testaukseen applikaation systemaattisen tutkimisen mahdollistamiseksi. Systemaattiseen tutkimiseen käytettiin syvyys ensin -tutkimista.

Testeissä verrattiin normaalia satunnaistestausta UGA:n ja satunnaistestauksen yhdistelmään seitsemällä applikaatiolla, ja huomattiin jälkimmäisen löytävän metodeja keskimäärin 32,5% ensimmäistä enemmän. Systemaattista läpikäymistä verrattiin UGA:n ja systemaattisen läpikäymisen yhdistelmään — jälkimmäinen löysi keskimäärin 30,8% enemmän metodeja.

### 4.1.5 Useilla alustoilla testaaminen

Nykyaikana kehittäjien täytyy luoda mobiiliapplikaatiostaan eri versioita, jotka toimivat eri alustoilla. Eri alustoja ovat esimerkiksi kaikki Android- ja iOS-versiot. Lukuisat päivitykset älylaitteisiin ja alustoihin tekevät mobiilitestausympäristöjen luomisen jokaiselle alustalle hankalaksi (Gao ym. 2014).

Gao ym. (2014) mukaan yhtenäisyyden puute testauksen automatisoinnin infrastruktuureissa hankaloittaa eri alustoilla testaamista. Eri testityökalujen integraatiota ja yhteentoimivuutta edistäviä standardoituja ratkaisuja skriptien tekemiselle ei ole. Heidän mukaansa ideaalinen ja uudelleenkäytettävä testiympäristö olisi helposti liitettävissä eri alustoihin, tukisi syste-

maattista applikaation asennusta ja suorittamista eri alustoilla ja omaisi laajat vaihtoehdot mobiiliverkkojen kokoonpanoille (configurations).

Testityökalu Mobilette (Grønli ja Ghinea 2016) testaa rinnakkain samaa applikaatiota Androidille ja iOS:lle. Mobilette koostuu neljästä osasta: asiakas, palvelin, Robotium-ajuri ja Frank-ajuri. Asiakas sisältää Mobiletten käyttöliittymän. Palvelin kommunikoi testattavan applikaation kanssa ajureiden välityksellä ja pitää yllä listaa yhdistetyistä laitteista. Lista sisältää laitteelle annetun ID-numeron sekä metatietoja, kuten laitteen käyttöjärjestelmän, IP-osoitteen ja näytönkoon. Ajurit muuntavat palvelimen käskyt laitteissa toimiville samannimisille testikehyksille sopiviksi (Androidilla Robotium, iOS:llä Frank). Näin käskyt voidaan antaa yhtäaikaa molemmille alustoille. Mobilette-prototyyppi tukee vain elementtien koskettamista ja tekstin asettamista ja hakemista.

Eri alustoilla toimivat testityökalut voisivat tulevaisuudessa helpottaa applikaatioiden kehitystä tekemällä testaamisesta yksinkertaisempaa. Testaus voi olla helpompaa toteuttaa, jos samaa testityökalua voi käyttää kaikilla alustoilla, joille applikaatiota kehitetään. Kasvava alustojen määrä kannustaa myös testityökalujen kehittäjiä tekemään useilla alustoilla toimivia työkaluja.

## 4.2 Emulointi vai natiivitesti

On paljon mielipide-eroja siitä, tulisiko applikaatioiden testaus suorittaa emulaattorilla vai natiivisti. Yksinkertaistettuna vaihtoehtoja on kolme: testityökalu ja applikaatio suoritetaan älylaitteessa (täysin natiivi), testityökalu suoritetaan tietokoneella/palvelimella ja applikaatio älylaitteessa (natiivi) tai testityökalu suoritetaan tietokoneella/palvelimella ja applikaatio tietokoneella emulaattorissa (emulointi). Keskimmäisessä tapauksessa voidaan myös miettiä, tuleeko älylaitteen olla kiinni tietokoneessa vai voiko testauksen hoitaa langattoman yhteyden välityksellä.

Emuloinnin hyvä puoli löytyy pienistä kustannuksista: emulaattoreita käyttämällä ei tarvitse hankkia älylaitteita testausta varten. Emulaattorin käytöllä on myös rajoituksia: emulaattori perustuu yleensä tiettyyn laitteeseen tai alustaan ja rajoituksiksi muodostuvat tämän laitteen ominaisuudet eikä suurin osa emulaattoreista kykene tunnistamaan kaikkia monimutkaisem-

pia eleitä (Gao ym. 2014). Tarkasteltavista testityökaluista kuusi toimii vain emulaattorin kanssa: GUI Ripper, AndroidRipper, Extended Ripper, MobiGUITAR, PATS ja nimetön testityökalu (Takala, Katara ja Harty 2011). Emulaattorin lisäksi oikeiden älylaitteiden kanssa toimii kaksi testityökalua: SwiftHand ja Mobilette. Toisen nimettömän testityökalun (Hu ja Neamtiu 2011) artikkelista ei löytynyt mainintaa emulaattorin tai natiivitestauksen käytöstä.

Täysin natiiviin testaamiseen liittyy paljon mahdollisia ongelmia. Tarkasteltavista testityökaluista ainoa täysin natiivi on MobTAF (Nagowah ja Sowamber 2012). Nagowah ja Sowamber (2012) esittivät, että testauksen voisi suorittaa tositilanteissa, kun testitapaukset ja testeissä tuotettu data tallennetaan puhelimeen. Yksi mahdollinen ongelma tässä on, että laitteen laskentateho tai muistikapasiteetti on liian pieni applikaation ja testityökalun yhtäaikaiseen suorittamiseen. MobTAF:in prototyyppi testattiin hyvin yksinkertaisella applikaatiolla, eikä laitteen hidastumista tai muistin loppumista esiintynyt testeissä. Toinen mahdollinen ongelma täysin natiivissa testauksessa on testiympäristön hallinnan puute. Esimerkiksi julkisen internetyhteyden nopeudesta ja luotettavuudesta ei ole takeita ja puhelun tulo kesken testin voi keskeyttää applikaation toiminnan ja näin pilata testin.

Aidoimman kuvan applikaation käytöstä saa natiivitestauksella. Tällöin testityökalun suoritus ei häiritse applikaation suoritusta, ja applikaatio suoritetaan tarkoitetussa ympäristössään. Kun testityökalu suoritetaan tietokoneessa, voi tietokone hoitaa suurta laskentatehoa tarvitsevat operaatiot ja lokitiedostojen tallentamisen. Oikeiden älylaitteiden käyttäminen testauksessa vaatii mukautumista laitteiden nopeaan kehitykseen (Gao ym. 2014). Natiivitestausta käyttää kahdeksan testityökalua: $A^3E$, BugRocket, MobTAF, RERAN, SPAG, SPAG-C, T+ ja UGA.

Tuleeko natiivitestauksessa tietokoneen olla testattavaan laitteeseen yhteydessä johdon vai langattoman yhteyden välityksellä? Johdon käyttö tekee yhteydestä varman ja nopean, kunhan johto ei ole viallinen. Toisaalta se saattaa estää laitteen liikuttamista. Jos langaton yhteys voidaan toteuttaa luotettavasti, voisi tuloksena olla testaustapa, joka mahdollistaa laitteen käytön luonnollisella tavalla ja antaa aidon kuvan applikaatiosta.

## 4.3 Testauksen nopeuttaminen

Yksi suuri menetelmä testauksen nopeuttamiseen on rinnakkaistestaus. Tällöin testitapauksia suoritetaan yhtäaikaisesti eri laitteissa tai emulaattoreissa. Useimmiten yksittäisten testien tulokset kootaan yhteen kun kaikki testitapaukset on suoritettu ja näin saadaan kokonaiskuva testeistä.

PATS (Wen ym. 2015) ja BugRocket (Ma ym. 2016) käyttävät isäntä-orja -mallia (master-slave model). PATS:in prototyyppi käytti isäntäpalvelinta ja kahta orjapalvelinta isännän toimiessa testikoordinaattorina. Näin PATS sai lyhennettyä testaukseen käytettävää aikaa 18–35% verrattuna testeihin, joissa käytetään yhtä isäntäpalvelinta kaikkien testien suorittamiseen. BugRocketissa orjapalvelimet muodostavat yhteyden testattaviin laitteisiin ja keräävät tietoa testien aikana. Tiedot välitetään isäntäpalvelimelle, joka analysoi tiedot ja tekee niistä yhteenvedon.

Rinnakkaistestaus ei ole kuitenkaan ainoa tapa nopeuttaa testausta. SwiftHand keskittyy testauksen nopeuttamisessa siihen, että applikaatiota uudelleenkäynnistetään mahdollisimman harvoin. Choi, Necula ja Sen (2013) mukaan jokaisen automaattisen tutkimisalgoritmin täytyy ajoittain uudelleenkäynnistää applikaatio päästäkseen tutkimaan aiemmin saavuttamattomia näkymiä. Ainoa luotettava tapa uudelleenkäynnistykseen on poistaa applikaatio ja asentaa se uudelleen. Tällaiseen uudelleenkäynnistykseen kuluu 30 sekuntia.

SwiftHand yrittää jokaisessa näkymässä pidentää testin suoritusta simuloimalla näkymässä sallittuja käyttäjätapahtumia (mukaanlukien takaisin- ja kotinäppäinten painaminen). Tämä perustuu olettamukseen, että on nopeampaa tutkia muita siirtymiä kuin uudelleenkäynnistää applikaatio. SwiftHandin menetelmää verrattiin satunnaistestaukseen ja koneoppimiseen perustuvaan testaukseen (Angluinin L*-algoritmiin). Testeissä SwiftHand saavutti suuremman osan applikaatioiden koodeista kuin nämä tavat ja oli niitä nopeampi. SwiftHand käytti uudelleenkäynnistyksiin aikaa 2,1–17,6% testauksen kestosta. Satunnaistestauksessa vastaavat luvut olivat 18,2–49,2% ja koneoppimistestauksessa 41,6–81,6%.

## 4.4 Huomioita lähdekoodin käytöstä ja testien kattavuudesta

GUI-testaus tapahtuu pääasiassa GUI:ta käsittelemällä, eikä lähdekoodin käyttö yleensä ole tarpeen. GUI-testaus lasketaankin black box -testaukseen kuuluvaksi. Lähdekoodia voidaan silti käyttää esimerkiksi JUnit-työkalulla testitapauksia tehtäessä tai metodien määrän laskemiseen. Metodien määrä on tärkeä tietää, jos halutaan demonstroida testityökalujen tehokkuutta metodien löytämisellä.

Jos lähdekoodia halutaan testauksessa käyttää, se on mahdollista erityisesti jos testaus tapahtuu applikaation kehittämisen aikana kehittäjien toimesta, tai lähdekoodi on muutoin vapaasti saatavilla. Jos taas testaajana toimii jokin kolmas osapuoli, jolle ei lähdekoodia haluta luovuttaa, on hyvä olla myös testaustapoja, jotka eivät tarvitse testaukseen applikaation lähdekoodia.

Monet pitkälle automatisoidut testityökalut, kuten MobiGUITAR, keskittyvät vikojen löytämisessä applikaation kaatumisen aiheuttaviin vikoihin. Nämä viat vaativat korjaamista, mutta eivät ole ainoita vikoja, jotka tekevät applikaatioista toimimattomia. Sellaiset viat, jotka vaativat testaajalta applikaation kontekstin tuntemusta, voivat olla automaattisilla testityökaluilla hyvin vaikeita tai mahdottomia havaita. Esimerkiksi tilanteessa, jossa applikaation näkymässä on tekstiä, testityökalu voi tunnistaa näkymässä olevan tekstilaatikon, ja että tekstilaatikko sisältää tekstiä. Voiko testityökalu kuitenkaan päätellä, onko teksti kontekstiin kuuluvaa, vai täytyykö ihmisen aina huomata tällaiset viat?

Voitaneen todeta, että automatisoitu GUI-testaus ei yksin kykene sellaiseen tietoisuuteen, jota edellämainitun kaltaiset viat vaativat tullakseen havaituiksi. Siksi hyviin tuloksiin päädytään sellaisilla testityökaluilla, joita ihminen voi ohjata. Manuaalisellakin testauksella voi löytää paljon vikoja, mutta ajan säästämiseksi automatisoitu testaus on hyvä vaihtoehto. Tekoälyn kehittyessä voisi olla mahdollista kehittää täysin automaattisia testityökaluja, jotka voivat löytää vikoja, joiden löytäminen vaatii testaajan tietoisuutta. Tätä voisi tutkia tulevaisuudessa.

# 5 Yhteenveto

Suuri osa olemassaolevista lähestymistavoista keskittyy yhteen ongelman osa-alueeseen, kuten siihen, miten nopeuttaa testausta (Wen ym. 2015) tai suorittaa testaus oikeassa älypuhelimessa emulaattorin sijaan (Nagowah ja Sowamber 2012). Tällöin esitettävät ratkaisut ovat hyvin tilannekohtaisia, eivätkä yksinään ratkaise suurta ongelmaa, jonka kasvava mobiilikulttuuri ohjelmistotestaukselle asettaa.

Parempia lopputuloksia saadaan testityökaluilla, jotka ratkaisevat useita ongelmia. Esimerkiksi Mobilette (Grønli ja Ghinea 2016) testaa yhtäaikaisesti samaa applikaatiota kahdella eri alustalla ja tallennus ja toisto -testityökalu RERAN (Gomez ym. 2013) käyttää toistovaiheessa lokitiedostoon nauhoitettuja monimutkaisia eleitä (kuten swipe) päästäkseen tutkimaan suuremman osan GUI:ta.

Malliin perustuva testaus kestää parhaiten GUI:n muutoksia. Tallennus ja toisto testaa applikaatiota oikeissa älylaitteissa. Siirtokopioinnilla voi suorittaa applikaation jäsennellysti ja automatisoidusti, mutta vikojen löytäminen keskittyy kaatumisen aiheuttaviin vikoihin. Näitä kolmea tapaa yhdistävä $A^3E$ (Azim ja Neamtiu 2013) sekä mallinnusta ja tallennus ja toisto -tapaa yhdistävä UGA (Li ym. 2014) tarjoavat ketterämpiä ratkaisuja, jotka selviytyvät monenlaisista ongelmista. $A^3E$ käyttää kahta erilaista systemaattista tutkimistapaa ja elekirjastoa applikaation läpikäymiseen. UGA yhdistää tallennus ja toisto -testaustapaan systemaattista tutkimista, jolloin monimutkaiset eleet löytyvät lokitiedostosta ja auttavat applikaation tutkimisessa. Molemmat testityökalut saavuttivat testeissään paljon suuremman osan applikaatiosta kuin tavat joihin niitä verrattiin.

Tulevaisuudessa voisi tutkia sellaisten testiympäristöjen kehittämistä, jotka toimivat useiden laitteiden ja alustojen kanssa. Testauksen automatisoinnin tutkimisessa voisi keskittyä siihen, onko kaikkien osien automatisoiminen kannattavaa — onko esimerkiksi manuaalisesta mallintamisesta niin paljon hyötyjä, että mallinnuksen automatisoiminen tekee testauksesta huonompaa? Vaihtoehtoisesti voisi tutkia, onko tekoälyn käyttäminen testityökaluissa hyödyllistä ja kasvattaisiko se automaattisten testityökalujen mahdollisuuksia löytää vikoja.

# Lähteet


Amalfitano, D., A. R. Fasolino, P. Tramontana ja N. Amatucci. 2013. “Considering Context Events in Event-Based Testing of Mobile Applications”. Teoksessa *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops,* 126–133. ID: 1. doi:10.1109/ICSTW.2013.22.

Amalfitano, D., A. R. Fasolino, P. Tramontana, S. De Carmine ja G. Imparato. 2012. “A toolset for GUI testing of Android applications”. Teoksessa *2012 28th IEEE International Conference on Software Maintenance (ICSM),* 650–653. ID: 1. doi:10.1109/ICSM.2012.6405345.

Amalfitano, D., A. R. Fasolino, P. Tramontana, S. De Carmine ja A. M. Memon. 2012. “Using GUI ripping for automated testing of Android applications”. Teoksessa *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering,* 258–261. ID: 1. doi:10.1145/2351676.2351717.

Amalfitano, D., A. R. Fasolino, P. Tramontana, B. D. Ta ja A. M. Memon. 2015. “MobiGUITAR: Automated Model-Based Testing of Mobile Apps”. ID: 1, *IEEE Software* 32 (5): 53–59. ISSN: 0740-7459. doi:10.1109/MS.2014.55.

Azim, Tanzirul, ja Iulian Neamtiu. 2013. “Targeted and Depth-first Exploration for Systematic Testing of Android Apps”. PT: J; NR: 41; TC: 16; J9: ACM SIGPLAN NOTICES; PG: 20; GA: 261XF; UT: WOS:000327697300036, *Acm Sigplan Notices* 48 (10): 641–660. ISSN: 0362-1340. doi:10.1145/2509136.2509549.

Choi, Wontae, George Necula ja Koushik Sen. 2013. “Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning”. PT: J; NR: 44; TC: 16; J9: ACM SIGPLAN NOTICES; PG: 17; GA: 261XF; UT: WOS:000327697300035, *Acm Sigplan Notices* 48 (10): 623–639. ISSN: 0362-1340. doi:10.1145/2509136.2509552.

Gao, J., X. Bai, W. T. Tsai ja T. Uehara. 2014. “Mobile Application Testing: A Tutorial”. ID: 1, *Computer* 47 (2): 46–55. ISSN: 0018-9162. doi:10.1109/MC.2013.445.

Gomez, L., I. Neamtiu, T. Azim ja T. Millstein. 2013. “RERAN: Timing- and touch-sensitive record and replay for Android”. Teoksessa *2013 35th International Conference on Software Engineering (ICSE),* 72–81. ID: 1. doi:10.1109/ICSE.2013.6606553.

Grønli, T. M., ja G. Ghinea. 2016. “Meeting Quality Standards for Mobile Application Development in Businesses: A Framework for Cross-Platform Testing”. Teoksessa *2016 49th Hawaii International Conference on System Sciences (HICSS),* 5711–5720. ID: 1. doi:10.1109/HICSS.2016.706.

Hu, Cuixiong, ja Iulian Neamtiu. 2011. “Automating GUI Testing for Android Applications”. Teoksessa *Proceedings of the 6th International Workshop on Automation of Software Test,* 77–83. AST ’11. Waikiki, Honolulu, HI, USA: ACM. ISBN: 978-1-4503-0592-1. doi:10.1145/1982595.1982612. http://doi.acm.org/10.1145/1982595.1982612.

Li, X., Y. Jiang, Y. Liu, C. Xu, X. Ma ja J. Lu. 2014. “User Guided Automation for Testing Mobile Apps”. Teoksessa *2014 21st Asia-Pacific Software Engineering Conference,* 1:27–34. ID: 1. doi:10.1109/APSEC.2014.13.

Lin, Ying-Dar, E. T. H. Chu, S. C. Yu ja Y. C. Lai. 2014. “Improving the Accuracy of Automated GUI Testing for Embedded Systems”. ID: 1, *IEEE Software* 31 (1): 39–45. ISSN: 0740-7459. doi:10.1109/MS.2013.100.

Lin, Ying-Dar, Jose F. Rojas, Edward T. H. Chu ja Yuan-Cheng Lai. 2014. “On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices”. PT: J; NR: 29; TC: 2; J9: IEEE T SOFTWARE ENG; PG: 14; GA: AR9ML; UT: WOS:000343899100002, *IEEE Transactions on Software Engineering* 40 (10): 957–970. ISSN: 0098-5589. doi:10.1109/TSE.2014.2331982.

Linares-Vásquez, M. 2015. “Enabling Testing of Android Apps”. Teoksessa *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,* 2:763–765. ID: 1. doi:10.1109/ICSE.2015.242.

Ma, X., N. Wang, P. Xie, J. Zhou, X. Zhang ja C. Fang. 2016. “An Automated Testing Platform for Mobile Applications”. Teoksessa *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C),* 159–162. ID: 1. doi:10.1109/QRS-C.2016.25.

Nagowah, L., ja G. Sowamber. 2012. “A novel approach of automation testing on mobile devices”. Teoksessa *2012 International Conference on Computer and Information Science (ICCIS),* 2:924–930. ID: 1. doi:10.1109/ICCISci.2012.6297158.

Takala, T., M. Katara ja J. Harty. 2011. “Experiences of System-Level Model-Based GUI Testing of an Android Application”. Teoksessa *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation,* 377–386. ID: 1. doi:10.1109/ICST.2011.11.

Wen, H. L., C. H. Lin, T. H. Hsieh ja C. Z. Yang. 2015. “PATS: A Parallel GUI Testing Framework for Android Applications”. Teoksessa *2015 IEEE 39th Annual Computer Software and Applications Conference,* 2:210–215. ID: 1. doi:10.1109/COMPSAC.2015.80.