KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutus

Juuso Tuunanen

REACT-KOMPONENTTIEN YKSIKKÖ- JA INTEGRAATIOTESTAUS

Opinnäytetyö
Toukokuu 2020

Karelia
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
Toukokuu 2020
Tietojenkäsittelyn koulutus

Tikkarinne 9
80200 JOENSUU
+358 13 260 600 (vaihde)

Tekijä
Juuso Tuunanen

Nimeke
React-komponenttien yksikkö- ja integraatiotestaus

Toimeksiantaja
Nolwenture Oy

Tiivistelmä

Tämän opinnäytetyön tavoitteena oli tutkia React JavaScript -kirjastolla toteutetun websovelluksen yksikkö- ja integraatiotestaamista. Ohjelmiston testaaminen ei pelkästään tuo laadullista parannusta koodiin, mutta tuo sitä enemmän säästöjä yritykselle mitä aiemmin testit luodaan ohjelmistokehitysprosessissa. Ohjelmistotestaus ei ole helppoa, vaan yksinkertaisenkin ohjelman testaaminen kaikilla mahdollisilla arvoilla on todella epäkäytännöllistä.

Aihetta lähestyttiin tutkimalla ohjelmistotestausta yleisellä tasolla, täysin koodikieleen tai -arkkitehtuuriin katsomatta, etenemällä kohti käytännön toteutusta tutustumalla React-komponenttien rakenteeseen sekä yksinkertaisten komponenttien luontiin. Lopuksi tutkittiin, mitä Reactin virallisen dokumentaation ja yleisten käytäntöjen puitteissa olevia vaihtoehtoja oli tarjolla React-komponenttien testaukseen. Löydetyt tulokset sekä testauksen teoria yhdistettiin lopuksi yksinkertaisen React-sovelluksen käytännön testaamiseen.

Tämän opinnäytetyön perusteella voidaan todeta, että React-komponenteille on mahdollista kirjoittaa yksikkö- sekä integraatiotestejä. Virallinen dokumentaatio ja muu testaukseen liittyvä kirjallisuus on laajaa. Kuitenkaan tuloksista ei voida vetää johtopäätöstä, että kaikki React-komponentit ovat testattavissa. Erityisesti kolmannen osapuolen kirjaston käyttäminen React-komponenteissa voi johtaa tilanteeseen, jossa niitä ei voi testata lainkaan. Tästä syystä React-komponenttien testaamisesta riittää muillekin vielä paljon tutkittavaa.

Kieli
suomi

Sivuja 76
Liitteet 6
Liitesivumäärä 12

Asiasanat
react, komponentti, web, sovellus, testaus

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
May 2020
Degree Programme in Business
Information Technology

Tikkarinne 9
80200 JOENSUU
FINLAND
+ 358 13 260 600 (switchboard)

Author
Juuso Tuunanen

Title
React component unit and integration testing

Commissioned by
Nolwenture Oy

Abstract

The purpose of this thesis was to research unit and integration testing for web application using React JavaScript library. Software testing does not only improve the quality of code but brings enormous savings for the company, the earlier testing is included in the software development process. Software testing is not easy and testing a simple application with all possible inputs can be impractical.

This subject was approached by researching software testing on a general level, disregarding any code language or architecture, and by exploring the structure of React components and creation of simple components in a practical manner. Finally, it was researched what options there were within official React documentation and within generally widespread practices. Discovered results and software testing theory were finally combined in practical testing of a simple React application.

Based on the results of this thesis, it can be stated that it is possible to create unit and integration tests for React components. Official React documentation and other software testing-related literature is widespread. Though, from the results, it cannot be stated that all React components are testable. Especially using third party libraries with React components can lead to a situation where components cannot be tested at all. For this reason, there still remains a lot of research for others to continue.

Language
Finnish

Pages 76
Appendices 6
Pages of Appendices 12

Keywords
react, component, web, application, testing

Sisältö

Liitteet

Liite 1 City-komponentti
Liite 2 City-komponentin testi
Liite 3 Search-komponentti
Liite 4 Search-komponentin testi
Liite 5 App-komponentti
Liite 6 App-komponentin testi

Lyhenteet

DOM Document Object Model (suom. dokumenttioliomalli) kuvaa puurakenteella esimerkiksi HTML, XHTML sekä XML-dokumenttien rakennetta. Rakenteen olioita voidaan muuttaa ja tarkastella JavaScriptiä käyttäen. (Wikipedia 2019.)

CD Continuous Delivery (suom. jatkuva toimitus) on automatisoitu ohjelmiston julkaisuprosessi, joka usein lisätään jatkuvan integraation (CI) jatkeeksi (Anastasov 2019).

CI Continuous Integration (suom. jatkuva integraatio) on ohjelmistokehittäjien käytänne, jossa koodi koostetaan keskeiselle palvelimelle (Anastasov 2019).

CI/CD CI/CD on ohjelmiston toimitusprosessi, joka automaattisesti suorittaa esimerkiksi ohjelmiston julkaisun tuotantoympäristöön tai automatisoidut testit. Tarkoituksena on poistaa inhimilliset virheet, nopeuttaa tuotteen iteraatioita sekä tarjota standardisoitu palaute tuotteen kehitystyöstä. (Anastasov 2019.)

1 Johdanto

Testaus on lähes poikkeuksetta kallein toimenpide tai työvaihe ohjelmistoprojektissa. Maailmanlaajuisesti arviolta 25–65 % projektin kokonaiskustannuksista kuluu testaukseen ja Suomessa vastaava luku on keskimäärin 27 %. Siitä huolimatta, että testaus on kallista ja aikaa vievää, testauksesta on automatisoitu keskimäärin vain 10 %. Toisin sanoen valtaosa testauksesta on manuaalista. (Kasurinen 2013, 12.) Hintaa osittain selittää se, että yleisin testaustapa on ad hoc eli niin kutsuttu satunnaistestaus, jota ei suunnitella lainkaan etukäteen (Kasurinen 2013, 16).

Testaamista ei tulisi jättää pois paremman tuoton saamiseksi. Huolellisesti testattu ohjelmistoprojekti on taloudellisesti kannattavampi kuin huonosti testattu. Tämän lisäksi virheentäyteinen ohjelmisto voi aiheuttaa paljon huonoa mainetta ja täten karkottaa uudet asiakkaat yritykseltä. (Kasurinen 2013, 12.)

Ohjelmiston testausta ei tule myöskään jättää viime tippaan. Mitä myöhemmin virhe löydetään ohjelmistosta, sitä kalliimmaksi sen korjaaminen tulee. Mikäli ohjelmistovirhe huomataan suunnitteluvaiheessa sen korjaaminen maksaa 1–2 % ja tuotekehityksessä löydetty virhe 10 % siitä mitä se maksaisi julkaisun jälkeen. (Kasurinen 2013, 18.)

Web-ohjelmistokehitys voidaan jakaa karkeasti kahteen eri osa-alueeseen, jotka ovat palvelinpuolen ja käyttöliittymän kehitys. Palvelinpuolella hallinnoidaan tiedon tallentamista, kirjautumisen toteuttamista, maksuja ja niin edelleen. Webohjelmissa käyttöliittymä toteutetaan selaimeen käyttäen HTML, CSS ja JavaScript kieliä. Tässä opinnäytetyössä keskitytään React JavaScript -kirjastolla toteutetun käyttöliittymän testaukseen.

Reactin luoma ”Create React App” on valmis paketti, joka luo tarvittavan vähimmäisasennuksen React ympäristön käyttämiseen. Tämän lisäksi React-dokumentaatiossa annetaan selkeät ohjeet millä työkaluilla testaamisen voi aloittaa (React 2019g). Tämän opinnäytetyön tarkoituksena on selvittää, kuinka React-komponentteja voi testata käytännössä, aloittamalla Reactin tarjoamasta lähtökohdasta.

Tässä opinnäytetyössä ei ole tarkoitus tutkia, kuinka esimerkiksi toimeksiantajani käyttämä Redux-kirjasto otetaan huomioon testeissä. Redux on tilanhallintaan suunniteltu JavaScript-kirjasto, jota käytetään kaikissa toimeksiantajani projekteissa, joissa Reactia käytetään käyttöliittymän toteutuksessa. Tämä ei silti tarkoita sitä, että kaikissa React-sovelluksissa tai edes React-komponenteissa käytettäisiin Reduxin tarjoamaa tilanhallintaa. Siispä Redux rajataan tämän opinnäytetyön aiheen ulkopuolelle.

Theseuksesta löytyy muutama React-komponenttien testaukseen liittyvä opinnäytetyö vuodesta 2017 alkaen, käyttämällä hakutermejä ”react testausta” ja ”react test” sekä tarkennetulla rajauksella ”nimeke sisältää react”. Löytyneistä opinnäytetöistä kaksi olivat lähellä tämän opinnäytetyön aihetta. Käsittelen tämän opinnäytetyön tulosten ja Peltolan (2020) sekä Morozin (2019) opinnäytetöiden tulosten välisiä eroja luvussa 5.4.

React-komponenttien testattavuutta lähdetään selvittämään seuraavien tutkimuskysymysten kautta: Miksi webohjelmistoa tulisi testata? Mitä testausmenetelmiä on olemassa koodikieleen ja -arkkitehtuuriin katsomatta? Mitä tulisi ottaa huomioon React-komponenttien testauksessa? Näihin kysymyksiin vastataan käsittelemällä aihetta ensin yleisellä tasolla edeten kohti käytännön toteutusta. Luvussa kaksi käsitellään yleisiä, web-kehitykseen ja Reactiin riippumattomia, ohjelmistotestauksen käytänteitä ja testausmenetelmiä. Luvussa kolme käsitellään, kuinka Reactilla luodaan komponentteja. Luvussa neljä tarkastellaan, mitä vaihtoehtoja on tarjolla React-komponenttien testaukseen sekä käsitellään käytännönlähteisesti, kuinka luvussa kaksi esille tuodut yleiset periaatteet voidaan yhdistää React-sovelluksen testaamiseen. Luvussa viisi käsitellään, mitä päätelmiä

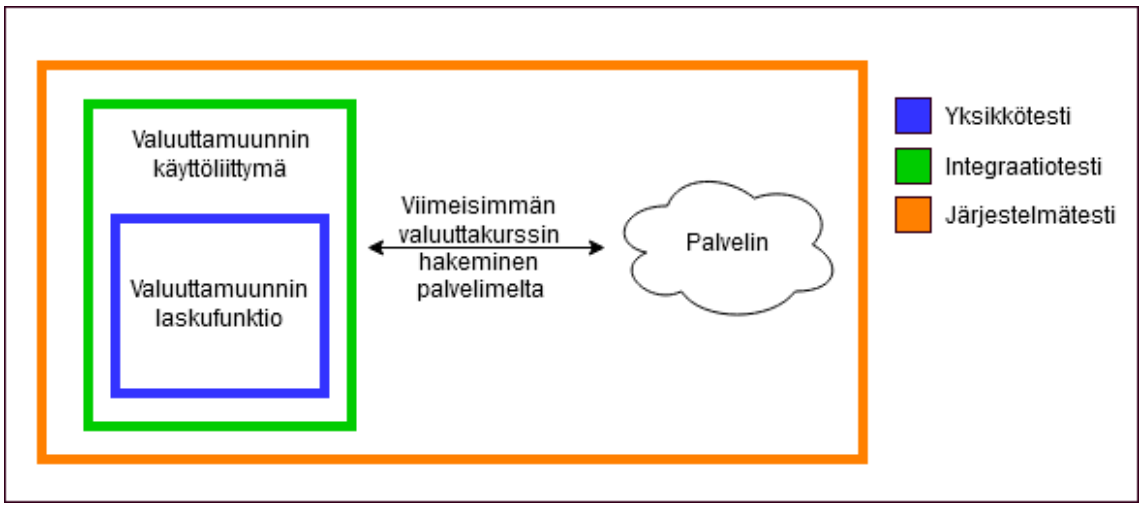
voidaan johtaa aiemmissa luvuissa esille tuoduista asioista sekä kuinka tämän opinnäytetyön tutkimusprosessi eteni.

2 Ohjelmistotestaus

Ohjelmistotestaus on suunniteltu prosessi, jonka tarkoituksena on varmistaa, että ohjelmistokoodi tekee mitä se on tarkoitettu tekemään sekä ettei se tee mitä sitä ei ole tarkoitettu tekemään. Ohjelmiston tulisi olla ennakoitavaa ja johdonmukaista, eikä sen tulisi aiheuttaa käyttäjälle yllätyksiä. (Myers 2011, 2.) Ohjelmistotestaus ei kuitenkaan ole helppoa. Yksinkertaisenkin ohjelman täysin kattava testaus on hyvin vaikeaa, ellei jopa mahdotonta. (Myers 2011, 3–4.) Täydellisessä maailmassa ohjelmiston jokainen versio haluttaisiin testata, mutta useimmiten se ei ole mahdollista. Yksinkertaisellakin ohjelmalla voi olla satoja tai tuhansia erilaisia syötteiden ja tulosteiden yhdistelmiä. Kaikkien yhdistelmien kattava testaus olisi epäkäytännöllistä, koska se veisi aivan liian pitkään ja vaatisi liikaa ihmisiä ollakseen taloudellisesti kannattavaa. (Myers 2011, 5.)

2.1 Testaustasot

Testaus voidaan jakaa neljään eri tasoon: yksikkö-, integraatio-, järjestelmä- ja hyväksymistestaukseen. Testaustaso nimitys tulee siitä, että testit kohdistuvat ohjelmiston eri osa-alueisiin. (Kasurinen 2013, 51.)



Kuvio 1. Valuuttamuunnin-komponentti testaustasoittain.

Yksikkötestit kohdistuvat yhteen komponenttiin. Kuviossa 1 esitetyssä valuuttamuunnin-komponentissa yksikkötesti luotaisiin testaamaan laskufunktion toiminnallisuutta. Integraatiotestit kohdistuvat useamman komponentin väliseen toimintaan. Kuviossa 1 integraatiotesti luotaisiin testaamaan käyttöliittymän ja laskufunktion välistä toiminnallisuutta. Järjestelmätestauksessa testataan koko ohjelmistoa kehitysympäristössä. Kuviossa 1 järjestelmätesti luotaisiin testaamaan käyttöliittymän ja palvelimen välistä toiminnallisuutta. Hyväksymistestaus eroaa järjestelmätestauksesta vain siten, että ohjelmistoa testaan lopullisessa kohdeympäristössä. (Kasurinen 2013, 51.)

Tämän opinnäytetyön testit tulevat olemaan vain yksikkö- ja integraatiotestejä, sillä tutkimuksen kohteena on käyttöliittymäkirjaston testaaminen, joka välittömästi sulkee pois järjestelmä- tai hyväksymistestauksen. Tästä huolimatta järjestelmä- sekä hyväksymistestausta avataan lyhyesti paremman kokonaiskuvan hahmottamiseksi.

2.1.1 Yksikkötesti

Yksikkötestillä testataan yksittäistä ohjelmiston osaa itsenäisesti, kuten esimerkiksi komponenttia tai funktiota (Kasurinen 2013, 51). Useimmiten testaaminen suoritetaan lasilaatikko testaustavalla (STF 2019a). Lasilaatikkotestauksessa testaaja kykenee näkemään komponentin sisäisen toteutuksen. Toisin sanoen

yksikkötesti vaatii yksityiskohtaista ymmärrystä koodista. Tämän takia on useimmiten tehokkaampaa, kun ohjelmistokehittäjä tekee yksikkötestin eikä ulkopuolinen testaaja. (Dustin, Rashka & Paul 2008, 351.) Kuitenkin parhaan tuloksen takaamiseksi ulkopuolinen testaaja kykenee tekemään yksikkötestin objektiivisemmin kuin ohjelmistokehittäjä, sillä ihmiset ovat usein sokeita omille virheilleen (Dustin, Rashka & Paul 2008, 352).

Yksikkötesti tekee ohjelmistokehityksestä nopeampaa, sillä hyvin testattu osa ohjelmistoa on monikäyttöisempi ja testausta ei tarvitse suorittaa esimerkiksi käyttöliittymän kautta. Laadukkaat yksikkötestit myös tekevät ohjelmistokehityksestä halvempaa, sillä mitä aiemmin virheet löytyvät ohjelmistokehityksessä, sitä halvemmaksi niiden korjaaminen tulee. (STF 2019a.)

Käytännön esimerkki yksikkötestistä on polkupyörän renkaan testaaminen. Rengasta tulee testata yksittäin ja täysin irrallaan polkupyörästä. Kun renkaaseen pumpataan ilmaa, sen tulisi säilyttää ilmanpaine sisällään. Mikäli näin ei tapahdu, testi on onnistuneesti paikallistanut renkaan vian, joka voidaan korjata. Samalla tavoin yksikkötestit toimivat osana ohjelmistokokonaisuutta; yksittäin ja eristyksistä kokonaisuudesta. Samalla tavoin polkupyörävalmistaja kykenee estämään viallisen polkupyörän myynnin, mikä aiheuttaisi lisäkustannuksia valmistajalle.

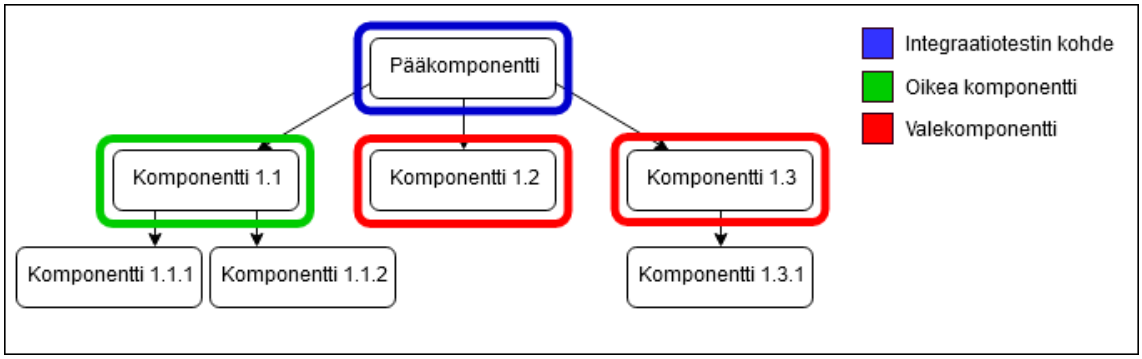
2.1.2 Integraatiotesti

Integraatiotestillä testataan useampaa ohjelmiston osaa yhtaikaa, jonka tarkoituksena on löytää mahdollisia yhteensopivuusvirheitä. Testiä suorittaessa kaikki testiin osallistuvat komponentit eivät välttämättä ole käytettävissä, jolloin niiden toimintaa jäljitellään luomalla valekomponentti. (STF 2019a.) Valekomponentilla matkitaan alkuperäisen komponentin toimintaa testien aikana, jotta testi voidaan kohdentaa tietyn toiminnallisuuden testaamiseen. (Elliott 2017).

Integraatiotestauksessa komponentit kytketään ja testaan vaiheittain yhdessä (Dustin, Rashka & Paul 2008, 354). Integraatiotestauksen tavoitteena on yhdis-

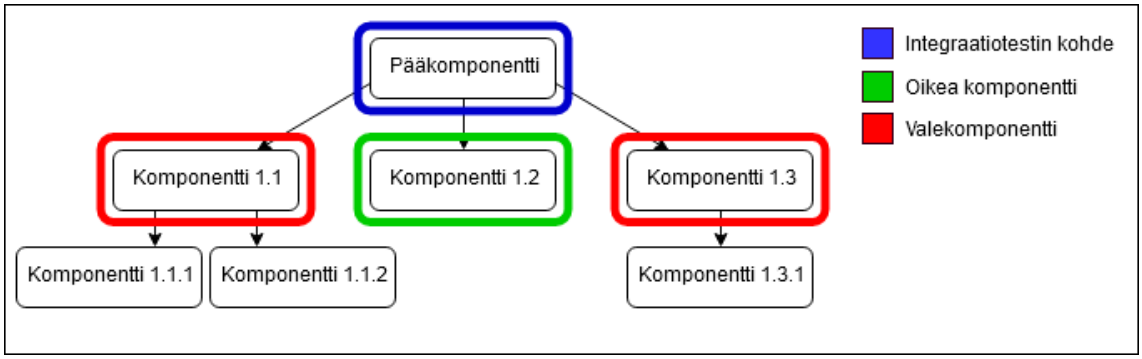
tää todistetusti toimivaan kokonaisuuteen yksi osa lisää ja tarkistaa kokonaisuuden toimivuus. Tähän on olemassa seuraavia lähestymistapoja: alkuräjähdys, ylhäältä alaspäin, alhaalta ylöspäin sekä voileipä. (Kasurinen 2013, 54–55.) Koska voileipä on yhdistelmä ylhäältä alaspäin ja alhaalta ylöspäin lähestymistapoja, sitä ei käsitellä erikseen.

Alkuräjähdyksessä kaikki komponentit kytketään yhteen testiä varten. Se muistuttaakin paljon savutestausta, jonka tarkoituksena on nähdä, käynnistyykö testattava kokonaisuus aiheuttamatta virheilmoituksia. Tämä lähestymistapa soveltuu parhaiten tilanteisiin, jossa vain pieneen osaan komponenteissa on tehty muutoksia. (Kasurinen 2013, 55.)



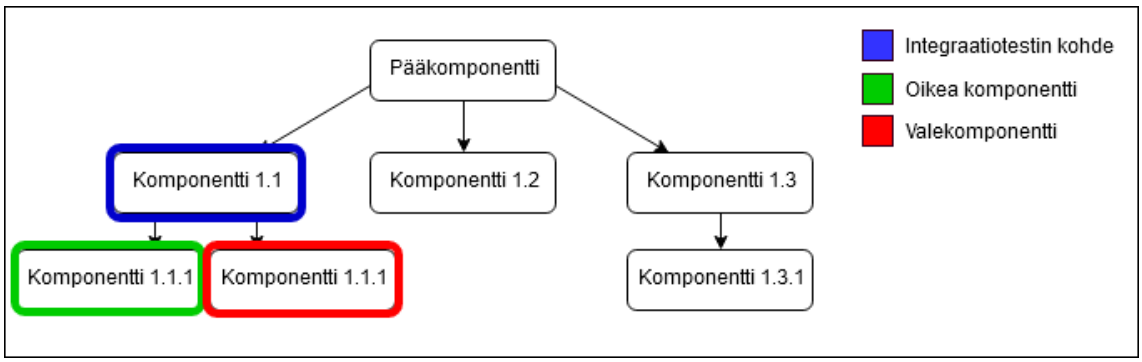
Kuvio 2. Esimerkki integraatiotestin komponenttihierarkiasta.

Ylhäältä alaspäin lähestymistavassa testi aloitetaan kuviossa 2 näkyvästä pääkomponentista, joka kutsuu alempia komponentteja 1.1–1.3, joille luodaan integraatiotestejä varten valekomponentit. Valekomponentit matkivat oikean komponentin toimintaa. Pääkomponentin testaamiseksi tarvitaan kolme integraatiotestiä. Ensimmäisessä pääkomponentin integraatiotestissä komponentti 1.1 on aito komponentti ja 1.2 sekä 1.3 ovat valekomponentteja. (Jorgensen 2008, 206–207.)



Kuvio 3. Pääkomponentille suoritettava integraatiotesti, jossa komponentti 1.2 on aito komponentti ja komponentit 1.1 sekä 1.3. ovat valekomponentteja.

Kuviossa 3 näkyvässä pääkomponentin toisessa integraatiotestissä, komponentti 1.2 on aito komponentti ja 1.1 sekä 1.3 ovat valekomponentteja. Kolmannessa testissä komponentit 1.1–1.2 ovat valekomponentteja ja 1.3 on aito komponentti. Näin pääkomponentti testataan kokonaan kolmella integraatiotestillä. (Jorgensen 2008, 206–207.)



Kuvio 4. Pääkomponentin testauksen jälkeen, komponentti 1.1 testataan.

Kun pääkomponentti on testattu, testataan komponentit 1.1–1.3 samaa toimintatapaa noudattaen. Kuviossa 4 testataan komponentti 1.1, jonka alikomponenteista 1.1.1 on aito komponentti ja 1.1.2 on valekomponentti. Kun komponentti 1.1 on testattu kokonaan, siirrytään testaamaan muita tason 1.X komponentteja. Tätä toimintatapaa jatketaan niin kauan, kunnes kaikki komponentit ovat testattu. (Jorgensen 2008, 206–207.)

Alhaalta ylöspäin lähestymistapa on peilikuva ylhäältä alaspäin lähestymistavasta. Sen sijaan, että alikomponenteista luotaisiin valekomponentteja, pääkomponentti korvataan valekomponentilla. Tämä lähestymistapa vähentää huomattavasti luotavien valekomponenttien määrää, mutta ylemmän tason valekomponentit ovat huomattavasti monimutkaisempia toteuttaa. (Jorgensen 2008, 206–207.)

Käytännön esimerkin kautta integraatiotestiä voidaan kuvata useamman eri polkupyörän osan yhteen sovittamista. Polkupyörän runkoon kiinnitetään vaiheittain keskiö, polkimet, rattaat, takarengas ja ketjut. Kun keskiö ja polkimet ovat kiinnitetty runkoon, testaan pyörivätkö polkimet eturattaan kanssa. Tämän jälkeen voidaan lisätä takarengas rattaineen ja testata pyörivätkö ne paikoillaan. Viimeisenä nämä kaksi kokonaisuutta yhdistetään toisiinsa ketjulla ja testataan, pyöriikö takarengas polkimia pyörittämällä. Mikäli kaikki toimii oikein, voidaan osien lisäämistä jatkaa, kunnes pyörä on täysin kasattu ja valmis järjestelmätestiin.

2.1.3 Järjestelmätesti

Nimensä mukaisesti järjestelmätesti suoritetaan koko ohjelmistolle, eikä testissä käytetä lainkaan valekomponentteja. Testin tavoitteena on varmistaa, toteuttaako ohjelmisto kaikki sille määritellyt tavoitteet ja toimiiko järjestelmä kokonaisuutena. (Kasurinen 2013, 56–57.) Järjestelmätestin tavoitteena ei siis ole löytää vikoja, vaan todentaa ohjelmiston toimivuus. Tämän vuoksi järjestelmätestit useimmiten tehdään funktionaalisina testeinä, mikä puolestaan johtaa intuitiivisempiin testeihin. (Jorgensen 2008, 229.) Funktionaalisessa testissä komponenttia tai funktiota testataan antamalle sille syötteitä ja tarkastelemalla, mitä tulosteita se antaa. (STF 2019j)

Käytännön esimerkki järjestelmätestistä on suorittaa koeajo polkupyörällä tehtaalla. Testaaja kokeilee, pitävätkö renkaat ilmanpaineensa, toimivatko jarrut ja kykeneekö polkupyörää ohjaamaan liikkeessä. Mikäli kaikki toimii, voidaan polkupyörä toimittaa jälleenmyyjälle asiakkaan ostettavaksi ja testattavaksi.

2.1.4 Hyväksyntätesti

Hyväksyntätestillä testataan ohjelmiston valmiutta toimitettavaksi loppukäyttäjälle. Ohjelmistokehityksen ulkopuoliset henkilöt suorittavat testit. Tarkoituksena on suorittaa niin sanottu koekäyttö, ennen ohjelmiston levittämistä laajemman yleisön saataville. Esimerkiksi yritysasiakas, joka on tilannut ohjelmistokehityksen toiselta yritykseltä, testaa ohjelmistoa ennen ohjelmiston myymistä kuluttajille. (STF 2019d). Hyväksyntätestissä järjestelmää käytetään kohdeympäristössä varmistaen, että ohjelmisto on riittävän laadukas ja täyttää vaatimusmäärittelyssä esitetyt vaatimukset. Sen onnistuessa ohjelmisto siirtyy virallisesti asiakkaan omaisuudeksi, useimmiten lopettaen kehitystyötä tehneen yrityksen velvollisuuden ylläpitää ohjelmiston kehitystä. (Kasurinen 2013, 57).

Polkupyöräesimerkkiä käyttäen asiakas suorittaa hyväksyntätestin ostaessaan polkupyörää. Mikäli koeajo sujuu onnistuneesti, raha vaihtaa omistajaa ja polkupyörä siirtyy asiakkaan omaisuudeksi.

2.2 Testausmenetelmät

Kehitysvaiheessa testaus suoritetaan seuraavassa järjestyksessä: aloitetaan tekemällä yksikkötestit komponenteille, jatketaan testaamalla komponentteja yhdessä integraatiotesteissä, testataan ohjelmistoa kokonaisuutena järjestelmätesteissä ja kaikkien näiden testien jälkeen ohjelmisto annetaan virallisesti tarkastettavaksi ja hyväksyttäväksi hyväksymistesteihin. Kuitenkaan eri testaustasoilla suoritettujen testien testausmenetelmiä ei ole hakattu kiveen vaan niitä voi suorittaa eri tavoin. (Kasurinen 2013, 64.)

On olemassa seuraavia testausmenetelmiä: mustalaatikko-, lasilaatikko-, harmaalaatikko-, ad hoc- sekä ketterätestaus (eng. *agile testing*) (STF 2019m). Tässä opinnäytetyössä ei keskitytä harmaalaatikkotestaukseen, sillä se on sekoitus musta- ja lasilaatikkotestausta (STF 2019m), jotka tarkemmin tulevissa aliluvuissa. Ketterään testaukseen tässä opinnäytetyössä ei myöskään syvennytä,

sillä se on vahvasti kytköksissä koko projektia koskevaan ketterään kehitysmenetelmään (eng. *agile software development*) (STF 2019m) ja täten menee opinnäytetyön laajuuden ulkopuolelle.

2.2.1 Musta laatikko

Musta laatikko -testaus on perinteisin testausmenetelmistä. Sen ideana on antaa komponentille syötteitä ja tarkkailla, miten komponentti reagoi syötteisiin. Ilmaisu musta laatikko tulee siitä, ettei komponentin sisäistä toimintaa tarkastella lainkaan, vaan testien tarkoitus on testata komponentin toiminnallisuutta. Vaikkakin tätä testausmenetelmää voi käyttää millä tahansa testaustasoilla (Kasurinen 2013, 65), sitä useimmiten käytetään integraatio-, järjestelmä- ja hyväksymistesteihin (STF 2019e).

Koska testattava ohjelmisto voi olla hyvinkin monimutkainen ja komponentin testaaminen kaikilla mahdollisilla arvoilla on todella epäkäytännöllistä (Myers 2011, 5), musta laatikko -testauksen tavoitteena on löytää mahdollisimman paljon virheitä rajallisella testimäärällä (Myers 2011, 10). Tämän vuoksi musta laatikko -testaukseen käytetään seuraavia tekniikoita: samanarvoisuuden ositus (eng. *equivalence partitioning*), raja-arvoanalyysi (eng. *boundary value analysis*) sekä syy-seuraus-kuvaaja (STF 2019e). Syy-seuraus-kuvaaja menee opinnäytetyön laajuuden ulkopuolelle, mutta samanarvoisuuden ositusta ja raja-arvoanalyysia on syytä käsitellä tarkemmin, sillä ne ovat keskeisessä osassa React-komponenttien testauksessa.

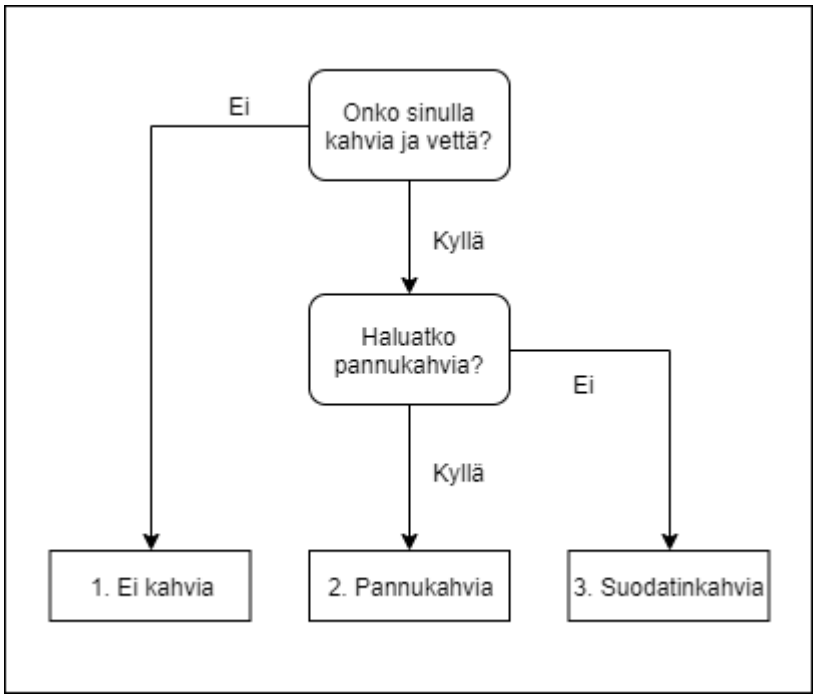
Samanarvoisuuden osituksen tarkoituksena on vähentää testattavien syötteiden kokonaislukumäärää jakamalle ne samanarvoisuusluokkiin (Haikala & Mikkonen 2011, 209). Esimerkiksi jos hyväksytty syöte komponentissa on kokonaisluku 1–999 välillä, niin testattavia samanarvoisuusluokkia olisi kolme: pätevä kokonaisluku väliltä 1–999; virheellinen kokonaisluku, joka olisi pienempi kuin 1 sekä virheellinen kokonaisluku, joka olisi suurempi kuin 999 (Myers 2011, 51). Tällä tavoin voidaan olettaa, että mikäli komponentti antaa virheen esimerkiksi kokonaisluvulla 28, komponentti antaisi saman virheen muillakin sen

samanarvoisuusluokan arvoilla eli pätevillä kokonaisluvuilla, jotka ovat välillä 1–999. Näin testien kokonaislukumäärää saadaan vähennettyä tuhannesta testistä yhteen.

Raja-arvoanalyysillä tarkoitetaan samanarvoisuusluokkien rajoilla olevia tapauksia (Haikala & Mikkonen 2011, 209). Testit, jotka hyödyntävät raja-arvoanalyysiä, tuottavat paremmin tulosta kuin testit, jotka eivät hyödynnä sitä, koska raja-arvoanalyysi kohdistaa testit hyväksyttävien arvojen rajalle sekä raja-arvoanalyysissä otetaan huomioon myös ohjelman tulosteet. Raja-arvoanalyysissä käytetään seuraavia tapoja löytämään testitapauksia. Mikäli ohjelma vastaanottaa lukuja välillä 1–2, tulisi ohjelmaa testata luvuilla 0, 1, 2 ja 3. Toisaalta ohjelman vastaanottaessa syötteitä väliltä 1–10, tulisi ohjelmalle antaa 0, 1, 10, 11 syötettä. Mikäli syötteet vaikuttavat tulosteeseen, esimerkiksi tuotetilauksen saadessa alennusprosentin tietyn summan ylittäessä, ohjelmalle tulee syöttää arvoja, joilla saadaan suurin alennus ja ei mitään alennusta. Ohjelmasta tulisi myös tutkia mahdollisuus saada virheellisiä alennusprosentteja. Mikäli tulosteeksi tulee rajattu määrä tuloksia syötteiden perusteella, esimerkiksi korkeintaan neljä hakutermiin liittyvää hakutulosta, tulisi syötteeksi antaa arvoja, joilla saadaan nolla, yksi ja neljä hakutulosta. Ohjelmasta tulisi myös tutkia mahdollisuus saada virheellisiä tuloksia, kuten enemmän kuin neljä hakutulosta. Mikäli syöte tai tuloste on järjestetty lista, esimerkiksi aakkosjärjestyksessä, testit tulisi kohdistaa ensimmäiseen ja viimeiseen arvoon. (Myers 2011, 55–56.)

2.2.2 Lasilaatikkotestaus

Lasilaatikkotestauksessa testaaja pystyy näkemään komponentin sisälle ja jäljittämään kooditasolla mistä virhe aiheutui. Tämä tekee lasilaatikkotestauksesta syvällisempää ja tarkempaa kuin musta laatikko -testaus. Tämä tosin edellyttää, että testaaja ymmärtää komponentin toimintaa sekä ohjelmointityötä. Lasilaatikkotestauksen mittareina käytetään muun muassa suoritettujen ohjelmapolkujen, käytettyjen syötteiden ja kokeiltujen komentojen määrää suhteutettuna niiden kokonaismäärään. (Kasurinen 2013, 67–68.) Pääasiassa lasilaatikkotestausta käytetään yksikkötesteissä (STF 2019f).



Kuvio 5. Esimerkki komponentin logiikasta.

Lasilaatikkotestauksessa ohjelmistokehittäjä kykenee näkemään koodista kuvion 5 logiikan ja täten kykenee testaamaan kaikki kolme eri suoritettua ohjelmapolkua syötteillä ei, kyllä ja ei sekä kyllä ja kyllä. Näin testeillä saadaan katettua kaikki kuviossa 5 esitetyn komponentin ohjelmointipolkujen kattavuus edellä mainituilla testeillä.

2.2.3 Ad hoc

Ad hoc -testaus on sattumanvaraista ja täysin suunnittelematonta. Testauksen tarkoituksena on löytää ohjelmistovirheitä, jotka eivät välttämättä tulisi muutoin löydetyiksi suunnitelmallisten tai prosessimaisen testauksen kautta. Testauksen haasteena on testien uudelleen tuottaminen, mutta tuloksena voi löytyä hyvinkin mielenkiintoisia ohjelmistovirheitä. Useimmiten ad hoc -testejä suoritetaan hyväksyntätestien aikana. (STF 2019g.)

Mikäli ohjelmiston käyttäjä avaisi sovelluksen, klikkailisi eri painikkeita sekä kokeilisi antaa ohjelmistolle muitakin satunnaisia syötteitä, tätä kutsuttaisiin ad hoc

-testaamiseksi. Testauksessa ei ole siis mitään päämäärää, vaan tarkoitus on umpimähkään antaa erilaisia syötteitä ohjelmistolle ja välttää niin sanottujen oletettujen prosessien, kuten sähköpostin lähettämisen seuraamista.

2.3 Testauslajit

Testauslajeja on rajoittamaton määrä ja vain niiden määritelmä erottelee ne toisistaan (STF 2019h). Tässä luvussa käsitellään savu-, funktionaalista-, käytettävyys sekä regressiotestausta, sillä ne liittyvät läheisesti React-komponenttien automaatiotestaukseen.

Muita huomionarvioisia testauslajeja ovat turvallisuus-, suorituskyky- ja vaatimustenmukaisuustestit (STF 2019h), harvemmin sovelletaan käyttöliittymän testaukseen. Tämän vuoksi edellä mainitut testauslajit ovat tämän opinnäytetyön laajuuden ulkopuolella, joten niitä ei käsitellä lainkaan.

2.3.1 Savutesti

Savutestin tarkoituksena on testata, että kaikki komponentin perusasiat toimivat. Testin tuloksen perusteella päätetään, onko testattava ohjelmisto tarpeeksi vakaa jatkotesteille. (STF 2019i.) Yksinkertainen järjestelmätason savutesti on esimerkiksi mennä hakukoneen etusivulle ja tarkkailla tuleeko käyttöliittymä kokonaan näkyviin. Tämän testin tarkoituksena on siis testata, onko testattava ohjelmisto valmis käytettäväksi.

Savutestissä ohjelmistoa ei testata perusteellisesti, vaan pyritään tuomaan esille ohjelmistokehityksen alkuvaiheilla integraatio ongelmat sekä muut pääasialliset ongelmat. Onkin suositeltavaa automatisoida testaus, mikäli testejä suoritetaan usein. Ohjelmiston kehittyessä, savutestejä tulee tehdä kattavammaksi. Savutestausta käytetään tavallisesti osana integraatio-, järjestelmä- ja hyväksyntätestejä. (STF 2019i)

2.3.2 Funktionaalinen testi

Funktionaalisessa testissä ohjelmistoa testataan sen vaatimusten ja määritelmien perusteella. Komponenttia tai funktiota testataan antamalle sille syötteitä ja tarkastelemalla mitä tulosteita se antaa. Funktionaalisissa testeissä ollaan kiinnostuneita testien lopputuloksesta eikä komponentin tai funktion toteutustavasta. Funktionaalista testausta käytetään tavallisesti järjestelmä- ja hyväksyntätesteissä. (STF 2019j.)

Esimerkki funktionaalisesta testistä on kirjautua sisälle sivustolle syöttämällä oikeat kirjautumistiedot ja testata pääseekö niillä sisälle ohjelmistoon. Testissä varmennetaan sisäänkirjautumisen toimivuus, eikä miten sisäänkirjautuminen käytännössä katsoen toteutetaan käyttöliittymän ja palvelimen välillä.

2.3.3 Käytettävyystesti

Käytettävyystestillä tarkoitetaan käyttäjän suorittamaa testiä, jolla pyritään saamaan selville ohjelmiston helppokäyttöisyys. Toisin sanoen, vaikka ohjelmisto olisi rakennettu toimimaan täysin virheettömästi, loppukäyttäjän suorittamat toiminnot voivat olla liian vaikeasti tavoitettavissa. Esimerkiksi usein käytetty tulostustoiminto voi olla jopa kymmenen klikkauksen päässä, kun se voitaisiin siirtää vain parin klikkauksen päähän. Käytettävyystestejä ajetaan normaalisti järjestelmä- ja hyväksyntätesteissä. (STF 2019k.)

Käytettävyyttä voidaan mitata käyttämällä muun muassa katseenseurantamonitoria, toiminnannauhoitussovellusta, taustavalvontaohjelmia, haastatteluja sekä reaktioiden kuvaamista. Näiden työkalujen tarkoituksena on saada käyttäjältä syötteitä sekä palautetta järjestelmän toimivuudesta ja intuitiivisuudesta. Esimerkiksi erään pelin ominaisuudet ja kenttäsuunnitelmat menivät uusiksi, kun testikäyttäjät ihastuivat bugin aiheuttamaan pelihahmon poikkeavaan liikkumistapaan. (Kasurinen 2013, 70–71.)

2.3.4 Regressiotesti

Regressiotestillä tarkoitetaan muutettujen komponenttien uudelleen testaamista aiemmin kirjoitetuilla testeillä. Regressiotesti pohjautuu ajatukselle, että useimmiten virheet sijoittuvat uuteen koodiin tai uutta koodia käyttävään toiminnallisuuteen. Tämän vuoksi koko komponentti tai ohjelmisto tulisi testata uudelleen. (Kasurinen 2013, 68–69.) Regressiotesti voidaan suorittaa missä tahansa ohjelmiston kehitysvaiheessa, mutta se on merkityksellisin järjestelmätestauksessa (STF 2019l).

Esimerkki regressiotestistä on suorittaa uudestaan aiemmin kirjoitetut testit komponentille. Esimerkiksi jossain sisäänkirjautumiskomponentissa olisi aiemmin voitu löytää vika, ettei siihen voinut kirjoittaa kuin fi-päätteisiä sähköpostiosoitteita. Viimeisimmässä virheraportissa olisi todettu, että sähköpostiosoitteet eivät myöskään voineet sisältää numeroita. Virheen korjaukseen jälkeen kaikki aiemmin kirjoitetut testit suoritetaan uudestaan, mukaan lukien testi, jossa testataan muitakin kuin fi-päätteisiä sähköpostiosoitteita. Tällä tavoin varmistetaan, ettei uuden virheen korjaaminen tuo takaisin aiemmin löydettyä virhettä.

2.4 Manuaalinen testaus

Luvussa 2 esiteltyjä testaustasoja, -menetelmiä ja -lajeja voidaan suorittaa automaattisesti tai manuaalisesti. Kasurisen (2013, 77–78) mukaan automaatiotestaus on tarkoitettu vähentämään manuaalisen regressiotestauksen määrää, koska alustavan investoinnin jälkeen automaatiotestaus on halvempaa ja helpompaa kuin jatkuvien manuaalisien testien suorittaminen. Testien automatisointia tulisi harkita, mikäli testit toistetaan vähintään 4–20 kertaa projektin aikana. Tämä ei kuitenkaan poista manuaalisten testien tarvetta. Automaatiotesteillä varmistetaan nykyisen ohjelmiston toimivuus ja manuaalitestauksella etsitään uusia tapoja rikkoa ohjelmistoa.

Manuaalisesti suoritetut testaustavat ovat niin tehokkaita löytämään virheitä, että niitä tulisi käyttää jokaisessa ohjelmistoprojektissa (Myers 2011, 19). Nämä testaustavat edesauttavat kahdella tavalla: mitä aiemmin virheet löydetään, sitä halvemmaksi niiden korjaaminen tulee ja sitä suuremmalla todennäköisyydellä ne korjataan oikein; sekä ohjelmoijat ovat taipuvaisia tekemään enemmän virheitä korjatessaan automaatiotestien löytämiä virheitä kuin manuaalitestauksessa löytyneitä virheitä. (Myers 2011, 20.) Tässä luvussa käsitellään koodikatselmointia, sillä se on yleisesti käytössä oleva manuaalinen testitapa.

Koodikatselmoinnilla tarkastetaan, onko koodi kirjoitettu yleisesti hyväksyttyjen ohjelmointikäytäntöjen perusteella oikein. Päämääränä on madaltaa ohjelmistokehitykseen liittyviä riskejä sekä varmistaa, että kaikki projektiin osallistuvat osapuolet tietävät mitä tekevät sekä seuraavat ohjeita toimien sovitulla tavalla. (Kasurinen 2013, 90.)

Koodikatselmoinnissa ryhmä ihmisiä visuaalisesti tarkastelee tai lukee koodia. Musta laatikko -testaukseen verrattuna, tämä menetelmä auttaa paikallistamaan vian tarkasti, odottamattoman virheilmoituksen sijaan. Automaatiotesteissä usein paljastuu yksittäisiä virheen oireita, joita korjataan yksitellen. Koodikatselmointi useimmiten paljastaa ryppään ongelmia, jotka kyetään korjaamaan kerralla. (Myers 2011, 20–21.)

Tämän menetelmän avulla ohjelmistosta löytyy 30–70 % logiikka-, suunnitelma- ja koodivirheistä. Prosenttiluvulla ei tarkoiteta kaikkia mahdollisia löytyviä virheitä, vaan testausprosessin aikana löytyviä virheitä. Kuitenkaan nämä menetelmät eivät ole tehokkaita havaitsemaan korkeamman tason suunnitteluvirheitä, joita on tehty muun muassa vaatimusmäärittelyjä laatiessa. (Myers 2011, 21)

2.5 Testauksen periaatteista ja käytännöistä

Testaukseen liittyy vahvaa ideologiaa. Luvuissa 2.1–2.4 käsitellään yksityiskohtaisemmin eri käytännön menetelmiä ja tekniikoita, mutta ennen niiden käyttöä on tärkeää ymmärtää mitkä periaatteet ohjaavat käytännön testausta.

Kuten musiikinteoria auttaa säveltäjää löytämään harmonian ja melodian musiikissa, nämä periaatteet auttavat testaajaa varmistamaan ohjelmiston laadun säilymisen ja kohentumisen. Periaatteita ei olekaan pakko seurata orjallisesti, vaan pikemminkin antaa niiden ohjata testausprosessia eteenpäin.

Tulevissa luvuissa käsitellään React-komponenttien testauksen näkökulmasta tärkeimmät periaatteet ja käytänteet, eikä kaiken kattavaa listausta. Koska periaatteet ovat hyvinkin ideologisia, ristiriitoja löytynee. Tulevissa luvuissa tutkimustyön aikana kohdattuihin ristiriitoihin otetaan kantaa.

2.5.1 Virheiden löytämien

Testauksen tarkoitus ei ole osoittaa, että ohjelmisto toimii, vaan pikemminkin lisätä ohjelmiston arvoa löytämällä virheitä. Ohjelmiston arvo kasvaa, kun löydettyjä virheitä poistetaan ja täten lisää ohjelmiston luotettavuutta. (Myers 2011, 6.)

Testausta onkin tärkeää lähestyä oikeasta näkökulmasta. Koska ihmiset ovat tavoitteellisia, on tärkeää antaa testaajalle oikea tavoite. Mikäli tavoite on osoittaa, ettei ohjelmistossa löydy virheitä, testaajalla on taipumus valita testitapauksia, jotka epätodennäköisemmin aiheuttavat ohjelmistossa virheitä. Onkin tärkeää asettaa tavoitteeksi löytää virheitä ohjelmistossa, joka ohjaa testitapauksia löytämään enemmän virheitä ohjelmistosta. (Myers 2011, 6.)

Ohjelmistotestaus onkin tuhoava prosessi, jossa pyritään löytämään virheet ohjelmistosta. Onnistunut testi aiheuttaa ohjelmistovirheen. Tietenkin, testauksen päämääränä on tuoda varmuutta, että ohjelmisto toimii suunnitellusti, mutta tähän päämäärään päästään parhaiten uutteralla virheiden etsinnällä. (Myers 2011, 8.)

2.5.2 Komponentin toteutuksen yksityiskohtien testauksesta

Komponentin testeissä ei tulisi testata minkä nimisiä funktioita komponentti kutsuu tai miten se hallitsee sisäistä tilaansa, vaan keskittyä testaamaan kuinka

komponentti käyttäytyy, kun sille antaa erilaisia syötteitä. Toisin sanoen, testeissä tulisi suosia pääasiassa funktionaalisia testejä eikä lasilaatikkotestausta. Funktionaalisen testin etuna on sen riippumattomuus komponentin sisäisestä toteutuksesta, joka mahdollistaa testien kirjoittamisen saman aikaisesti komponentin kanssa (Jorgensen 2008, 8).

```
1   const laskin = {
2     tulos: 0,
3     summa: function (luku1, luku2) {
4       this.tulos = luku1 + luku2;
5       return this.tulos;
6     },
7   };
8
9   const yhteenlasku = laskin.summa(2, 1);
10  console.log(laskin.tulos); // Tulostaa arvoksi 3
11  console.log(yhteenlasku); // Tulostaa arvoksi 3
```

Kuvio 6. Esimerkki summa olion toteutuksesta JavaScriptillä.

Kuviossa 6 oliolla laskin on kaksi sisäistä muuttujaa: summa ja tulos. Funktio summa tallentaa rivillä 4 yhteenlaskun tuloksen sisäiseen tilamuuttujaan tulos ja palauttaa sen arvon rivillä 5.

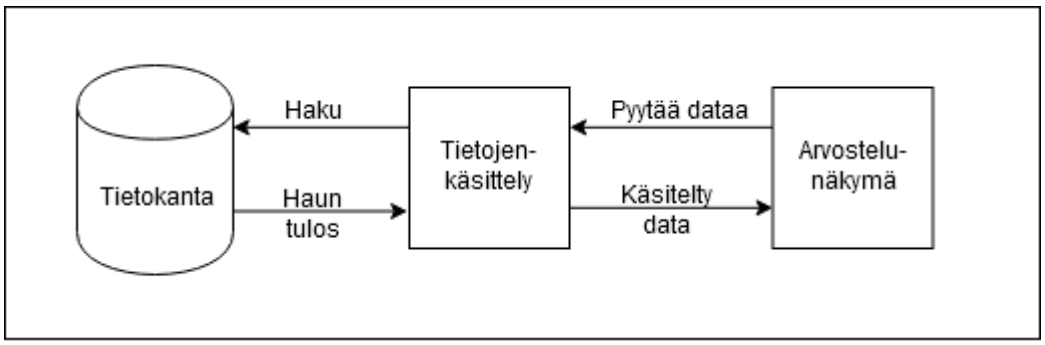
Oliolle laskin voisi luoda testin, joka tarkistaa sisäisen tilan muuttumisen ja yhteenlaskutoimituksen tuloksen sekä vertaisi näitä lukuja keskenään. Mikäli funktion summa toimintaa muutettaisiin siten, että se palauttaisi suoraan lukujen summan, tallentamatta sitä sisäiseen tilamuuttujaan tulos, aiemmin kirjoitetut testit antaisivat virheellisen ilmoituksen vaikkakin funktion summa toiminnallisuus säilyisi ennallaan. Tämän vuoksi onkin tärkeää luoda testit pääsääntöisesti testaamaan tulosteita komponentille annettujen syötteiden perusteella. React-komponenttien kanssa tilanne voi olla huomattavasti monimutkaisempi kuin kuviossa 6 esitetty esimerkki. Luvussa 4.1.2 on esitetty käytännön esimerkki React-komponentille luodusta testistä, joka kajoaa sisäiseen toteutukseen.

Luomalla funktionaalisia testejä, testitapauksista tulee vankempia, sillä komponentin sisäinen toteutus muuttuu useammin kuin komponentin rajapinta. Toisin

sanoen, lasilaatikkotestit voivat antaa aiheettomia virheilmoituksia, vaikka komponentin rajapinta ei olisi muuttunut lainkaan. Tämä johtaa hukkaan heitettyyn työhön, kun komponentin testejä joutuu kirjoittamaan uudelleen. (Elliott 2017.) On myös muistettava, että testaus on lähes poikkeuksetta kallein toimenpide tai työvaihe ohjelmistoprojektissa (Kasurinen 2013, 12). Hukkaan heitetyn työn välttämiseksi sekä projektin taloudellisen kannattavuuden säilyttämiseksi, komponentin sisäisen toteutuksen testaamista tulisi välttää.

2.5.3 Valekomponenttien käyttö

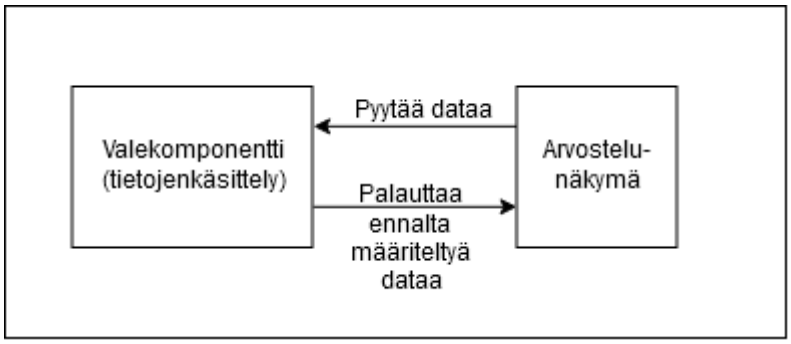
Valekomponentilla matkitaan aidon komponentin toimintaa testien aikana. Niitä käytetään useimmiten integraatiotesteissä korvaamaan yhtä tai useampaa testiin osallistuvaa komponenttia, jotta testi voidaan kohdentaa tietyn toiminnallisuuden testaamiseen. (Elliott 2017.)



Kuvio 7. Yksinkertaistettu komponenttirakenne arvostelunäkymän toiminnasta, mikä voisi olla käytössä esimerkiksi elokuva-arvostelusivustolla.

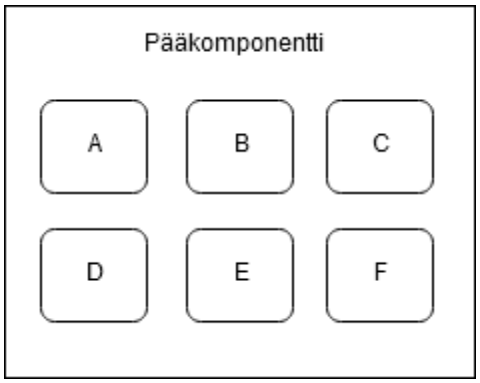
Kuvion 7 esimerkissä arvostelunäkymäkomponentti kutsuu tietojenkäsittelykomponenttia, joka puolestaan tekee tietokantakutsun. Tämän jälkeen tietokanta palauttaa hakukriteereihin sopivan tiedon tietojenkäsittelykomponentille käsittelyyn, jotta tieto saadaan rakenteellisesti sopimaan paremmin arvostelunäkymän tarpeisiin. Kun tieto on sopivassa muodossa, se palautetaan arvostelunäkymäkomponentille.

Mikäli arvostelunäkymäkomponenttia testataan ilman valekomponenttia, testi voi palauttaa virheellisen ilmoituksen epäonnistumisesta, kun tietokanta on suljettuna tai tietojenkäsittelykomponentissa tapahtuu jokin käsittelyvirhe. Toisin sanoen, valekomponentteja tarvitaan arvostelunäkymäkomponentin toiminnallisuuden testaamiseen muusta ohjelmistosta irrallaan. Useamman komponentin testejä kutsutaan integraatiotesteiksi. Mikäli kaikkien komponenttien yhteistoimintaa testattaisiin käyttämättä valekomponentteja, sitä kutsuttaisiin järjestelmä- tai hyväksyntätestiksi.



Kuvio 8. Tiedon käsittely -komponentti on korvattu valekomponentilla.

Kuvion 8 esimerkissä tietojenkäsittelykomponentti on korvattu valekomponentilla. Valekomponentti palauttaa jokaisessa testissä ennalta määriteltyä tietoa, joka soveltuu arvostelunäkymäkomponentille. Tällä tavoin kyetään testaamaan näyttääkö arvostelunäkymäkomponentti saamansa tiedon oikein. Testeissä paljastuvat virheet osoittavat nyt vain arvostelunäkymäkomponentin toiminnallisuuden puutteita.

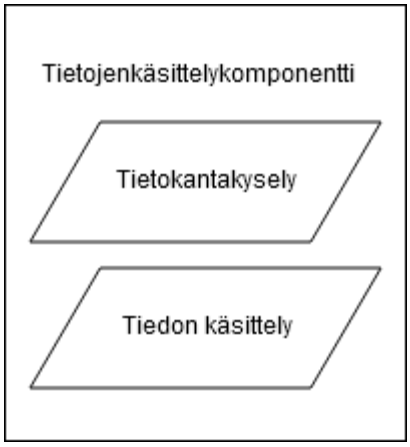


Kuvio 9. Esimerkki komponentin sisältämistä alikomponenteista.

Kuitenkin liiallista valekomponenttien käyttöä tulisi välttää. Mikäli alikomponentit A-F korvattaisiin kuvion 9 esimerkissä valekomponenteilla, testi ei kertoisi meille mitään hyödyllistä tietoa pääkomponentin toiminnallisuudesta.

Kuvion 7 esimerkissä käytettiin arvostelunäkymän ja tietokannan välissä yhtä komponenttia käsittelemään tiedonhakua ja tiedon käsittelyä. Jotta komponentin tietojenkäsittely toiminnallisuutta kyettäisiin testaamaan ilman tietokantakyselyn mahdollisesti aiheuttamia ongelmia, tietokantakyselylle tulisi luoda valekomponentti.

Jokainen kerta, kun testauksessa joudutaan käyttämään valekomponenttia, se paljastaa mahdollisuuden erotella toiminnallisuutta omiin komponentteihinsa. Valekomponenttien käyttöä voidaankin kutsua niin sanotuksia lemuksi koodissa. Tällä ei tarkoiteta, että koodi olisi ehdottoman väärin kirjoitettu, vaan pikemminkin mahdollisuudesta kohentaa koodin laatua. Testien tulisikin yksinkertaistaa koodia ja tehdä siitä helpommin ylläpidettävää. (Elliott 2017.)



Kuvio 10. Katsaus kuviossa 7 tietojenkäsittelykomponentin sisälle, joka koostuu karkeasti kahdesta eri toiminnallisuudesta.

Kuviossa 10 kuvatussa tietojenkäsittelykomponentista voidaan selkeästi erotella sen sisältämät kaksi toiminnallisuutta omiin komponentteihin. Kun toiminnallisuudet ovat eroteltu, tiedon käsittelyä voidaan testata funktionaalisilla testeillä, jossa komponentille annetaan syötteitä ja sen antamia tulosteita tarkastellaan. Valekomponentille ei ole siis tarvetta. Tämän lisäksi, mikäli tiedonhaun ja käsittelyn

prosessiin tulisi enemmän välivaiheita tai muita suuria muutoksia, kaikkia testejä ei tarvitsisi kirjoittaa uudestaan, sillä prosessia on jaoteltu pienempiin kokonaisuuksiin.

Pohjimmiltaan ohjelmistokehitys on ison ongelman pilkkomista pieniin itsenäisiin komponentteihin. Valekomponentteja tarvitaankin tilanteissa, jossa nämä itsenäiset komponentit tarvitsevat muita komponentteja toimiakseen. Toisin sanoen, valekomponentteja tarvitaan, kun itsenäiset komponentit eivät olekaan niin itsenäisiä kuin voisivat olla. (Elliott 2017)

3 React

React on Facebookin sekä vapaaehtoisten ohjelmoijien ylläpitämä toteava, tehokas ja joustava JavaScript-kirjasto käyttöliittymien rakentamiseen. React mahdollistaa monimutkaisten käyttöliittymien koostamisen pienistä ja eristetyistä koodinpätkistä, joita kutsutaan komponenteiksi. (React 2019a.) Facebook on amerikkalainen mainosrahoitteinen yritys, jonka tunnetuin tuote on Facebook-niminen yhteisöpalvelu.

3.1 Reactin toiminta

React on jatkuvan kehityksen kohteena ja kirjastoon lisätään vuosittain uusia ominaisuuksia. Vuoden 2019 aikana Reactin versioon lisättiin merkittävä uusi ominaisuus nimeltä Hooks (Abramov 2019), joka suoraviivaistaa Reactin käyttämistä. Tässä opinnäytetyössä käsitellään Reactia ennen versiota 16.8. avaten sen peruskäsitteitä näyttäen yksinkertaisia esimerkkejä Reactille kirjoitetusta koodista.

React yhdistää koodissa JavaScriptiä ja HTML:ää luoden oman syntaksijatkeen JavaScriptille, jota kutsutaan nimellä JSX. Tarkoituksena on yhdistää logiikkaa

käyttöliittymään, eikä näennäisesti erottaa niitä kahteen eri tiedostoon. (React 2019b.)

```
1   const element = <h1>Hello, world!</h1>;
```

Kuvio 11. Esimerkki JSX, jossa JavaScript muuttujalle annetaan arvoksi HTML elementti h1.

React-sovelmat sijoitetaan yhden HTML div-elementin sisälle, jonka prosessointia React ylläpitää (React 2019c). Lähtötilanteessa HTML-tiedosto sisältää esimerkiksi seuraavan div-elementin:

```
1   <div id="root"></div>
```

Kuvio 12. Elementti HTML-tiedostossa.

Lisätään Reactilla h1 HTML-elementin div-elementin sisälle:

```
1   const element = <h1>Hello, world</h1>;
2   ReactDOM.render(element, document.getElementById('root'));
```

Kuvio 13. Reactin prosessoima koodi sijoitetaan koodiesimerkissä 2 esitetyn div-elementin sisälle.

Lopputuloksena saadaan seuraava HTML tuloste:

```
1   <div id="root">
2     <h1>Hello, world</h1>
3   </div>;
```

Kuvio 14. Reactilla luotu yksinkertainen HTML tuloste.

Luodut React elementit voivat sisältää dynaamista sisältöä, kuten JavaScript muuttujia:

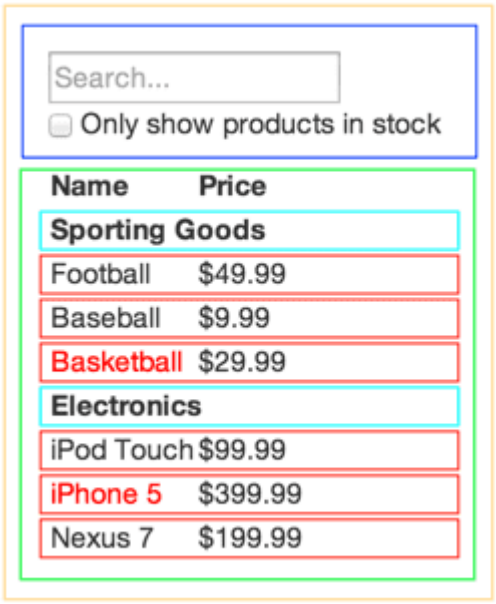
```
1   const country = "Finland";
2   const element = <h1>Hello, {country}</h1>;
```

Kuvio 15. React elementti sisältäen h1 HTML-elementin, joka saa dynaamisesti JavaScript muuttujasta osan tekstistään.

ReactDOM.render-funktio vertailee tämänhetkistä komponenttia edelliseen ja päivittää DOMia vain muutosten ilmaantuessa (React 2019c). Tämän toimintaperiaatteen päälle on mahdollista kirjoittaa monimutkaisia ohjelmia. Tällä tavoin kirjoittamalla koodi ei juurikaan eroaisi normaalista HTML-koodista, joten Reactissa logiikkaa sisältävä koodi paloitellaan komponenttien sisälle.

3.2 React-komponentit

Komponentit mahdollistavat käyttöliittymän jakamista itsenäisiin, uudelleen käytettäviin paloihin, ja auttavat käsittämään jokaisen palan kokonaisuutena. Käsitteellisesti komponentit toimivat kuten JavaScriptin funktiot. Niille voi syöttää arvoja, jotka palauttavat React elementtejä. (React 2019d.) Komponentin voi määritellä käyttäen ”yhden vastuun käytäntöä”, eli komponentti ihanteellisesti tekee vain yhden asian kerrallaan. Mikäli komponentin toiminta kasvaa, se tulisi jakaa useampaan osaan (React 2019e). Esimerkiksi kuviossa 16 punaisella reunuksella ympäröidyt HTML-elementit voidaan toteuttaa yhdellä React-komponentilla, sillä ne noudattavat yhdenmukaista toimintaa; komponentti näyttää nimen ja hinnan.



Kuvio 16. Esimerkki kokonaisuuden jakamista komponentteihin (React 2019e).

Reactin käyttö on muutoin joustavaa, mutta sen käyttöä ohjaa yksi tiukka sääntö: Kaikkien React-komponenttien tulee toimia puhtaiden funktioiden tavoin ja kunnioittaa saamiaan syötteitä. React-komponentti ei saa koskaan muokata saamiaan syötteitä (React 2019d).

Puhtaalla funktiolla tarkoitetaan funktiota, joka ei muuta sen saamia syötteitä ja se palauttaa aina saman tuloksen samoilla arvoilla. (React 2019d).

```
1   function summa(a, b) {
2     return a + b;
3   }
```

Kuvio 17. Puhdas funktio.

```
1   function nosto(tili, summa) {
2     tili.saldo = tili.saldo - summa;
3   }
```

Kuvio 18. Epäpuhdas funktio. Tämänkaltaisten funktioiden käyttäminen Reactissa on kielletty.

React-komponentit voi jakaa kahteen eri kategoriaan: funktionaalisiin ja luokkakomponentteihin. Funktionaaliset komponentit eivät sisällä sisäistä tilaa, kuten luokkakomponentit. (React 2019d) Tila on tarpeellinen, kun komponentista halutaan tehdä interaktiivinen. Staattisissa komponenteissa tulisi aina käyttää funktionaalisia komponentteja. (React 2019e)

3.2.1 Funktionaalinen komponentti

”Helpoin tapa määritellä funktionaalinen komponentti Reactilla, on kirjoittaa JavaScript-funktio” (React 2019d):

```
1   function Welcome(props) {
2     return <h1>Hello, {props.name}</h1>;
3   }
```

Kuvio 19. Funktionaalinen React-komponentti

Kuviossa 19 funktio toimii React-komponenttina, sillä se ottaa yhden olion nimeltä props syötteeksi ja palauttaa React elementin. Props on lyhenne englannin kielen sanasta properties, suomennettuna ominaisuudet. Näitä kutsutaan funktionaalisiksi komponenteiksi juuri siitä syystä, että nämä ovat kirjaimellisesti JavaScript-funktioita (React 2019d).

3.2.2 Luokkakomponentti

Luokkakomponentin voi kirjoittaa seuraavalla tavalla:

```
1   class Welcome extends React.Component {
2     render() {
3       return <h1>Hello, {this.props.name}</h1>;
4     }
5   }
```

Kuvio 20. React-luokkakomponentti

Reactin näkökulmasta kuviossa 20 esitetty luokkakomponentti on identtinen kuviossa 19 esitetyn funktionaalisen komponentin kanssa. Luokkakomponenteilla pystyy hallitsemaan myös komponentin tilaa, joita funktionaalisella komponentilla ei voi tehdä (React 2019d). Luvussa 3.3 käsitellään tarkemmin komponentin sisäistä tilanhallintaa.

3.2.3 Komponenttien käyttö komponentissa

Komponentit voivat viitata toisiin komponentteihin palauttaessaan React-elementtejä. Tämä mahdollistaa monimutkaisten asioiden abstrahointia yksinkertaisiin komponentti kutsuihin. React-sovelluksissa painike, lomake tai valintaikkuna ovat usein esitetty komponentteina. (React 2019d.)

```
1   function Welcome(props) {
2     return <h1>Hello, {props.name}</h1>;
3   }
4
5   function App() {
6     return (
7       <div>
8         <Welcome name="Sara" />
9         <Welcome name="Cahal" />
10        <Welcome name="Edite" />
11      </div>
12    );
13  }
14
15  ReactDOM.render(<App />, document.getElementById("root"));
```

Kuvio 21. Esimerkki React-komponenttien uudelleen käytössä toisen komponentin sisällä.

Kuviossa 21 riveillä 5–13 luodaan App-nimisen funktionaalisen komponentin, joka palauttaa kolme funktionaalista Welcome-komponenttia. Welcome-komponentti on määritelty riveillä 1–3. Rivillä 8 komponentille Welcome annetaan syötteen name arvoksi Sara. Rivillä 2 Welcome-komponentin määrittelyssä asetetaan

saadun name syötteen arvo h1 HTML-elementin sisälle. Sama prosessi toistetaan riveillä 9 ja 10. Rivillä 15 App-komponentti annetaan Reactille prosessoitavaksi.

```
1   <div>
2     <h1>Hello, Sara</h1>
3     <h1>Hello, Cahal</h1>
4     <h1>Hello, Edite</h1>
5   </div>;
```

Kuvio 22. HTML-tuloste koodiesimerkin 11 React-koodista.

3.3 React-komponenttien tilanhallinta

Tilaa käytetään komponenteissa, kun sen täytyy ylläpitää tietoa komponentin sisällä tapahtuvien päivitysten välillä (React 2019e). React tarjoaa useita eri sisäänrakennettuja tilanhallinta funktioita. Tilanhallinta funktiot jaetaan kolmeen eri kategoriaan: rakennus, päivitys ja purku. (React 2019f.)

```
1   export function LightBulb(props) {
2     let lightStatus;
3     if (props.powerOn === true) {
4       lightStatus = "Valo päällä!";
5     } else {
6       lightStatus = "Valo pois päältä!";
7     }
8     return <button>{lightStatus}</button>;
9   }
```

Kuvio 23. React-komponentti ilman tilanhallintaa.

```
class App extends Component {
  render() {
    return <LightBulb powerOn={true}></LightBulb>;
  }
}
```

Kuvio 24. Kuviossa 23 määritellyn komponentin käyttäminen ja oletustilan asettaminen.

Kuviossa 23 luokkakomponentti palauttaa painikkeen tekstillä ”Valo päällä!” tai ”Valo pois päältä!”, riippuen siitä minkä syötteen komponentti saa. Kuviossa 24 komponentille LightBulb annetaan syötteeksi powerOn totuusarvon true. Komponentissa ei ole kuitenkaan mitään logiikkaa, jolla valo voidaan laittaa päälle tai pois päältä. Olemassa olevaan button-elementtiin voidaan lisätä tapahtumakuuntelijan (engl. *event listener*) onClick, jonka avulla hiiren painalluksella voidaan muuttaa lampun tilaa. Tätä varten luokkakomponenttiin täytyy luoda tila.

```
export class LightBulb extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      powerOn: this.props.powerOn,
    };
  }

  togglePower() {
    this.setState({
      powerOn: !this.state.powerOn,
    });
  }

  render() {
    let lightStatus;
    if (this.state.powerOn === true) {
      lightStatus = "Valo päällä!";
    } else {
      lightStatus = "Valo pois päältä!";
    }
    return <button onClick={() => this.togglePower()}>{lightStatus}</button>;
  }
}
```

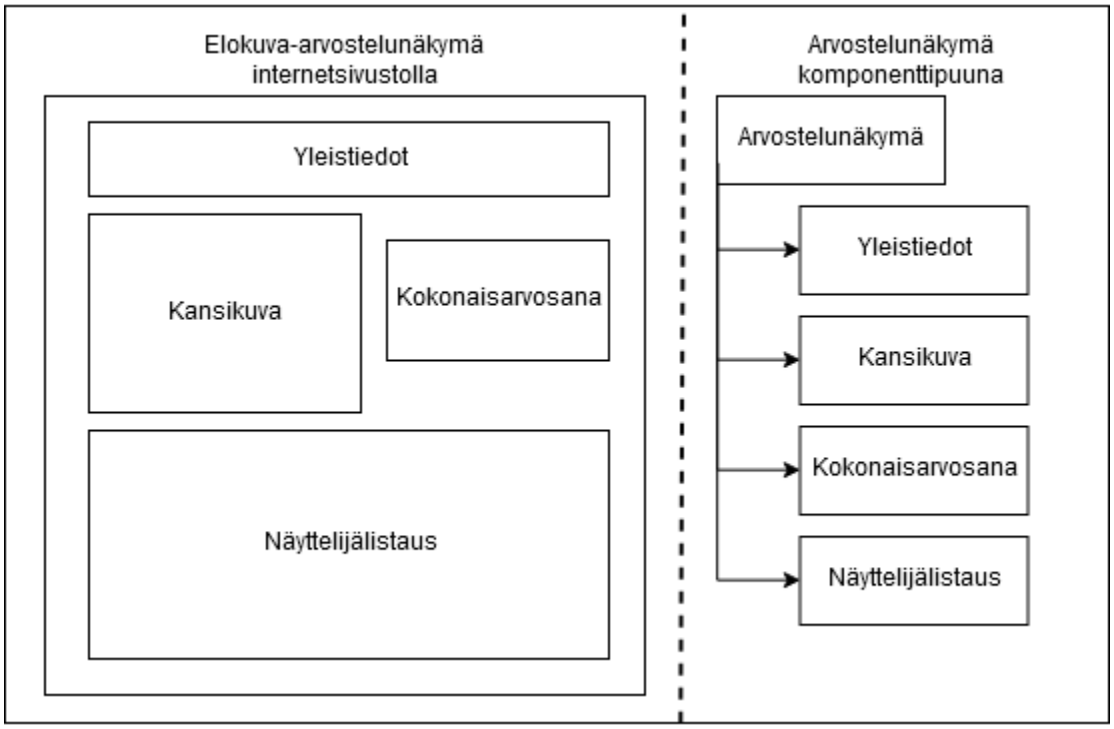
Kuvio 25. Luokkakomponentti tilanhallinnan kanssa.

Seuraavat muutokset tehtiin kuvioon 25, kuvioon 23 verrattuna: komponentti muunnettiin funktionaalisesta komponentista luokkakomponentiksi, constructor-funktio lisättiin, luokkakomponentille luotiin tila sekä funktio togglePower määriteltiin, joka muuttaa komponentin tilaa hiiren painalluksella.

Funktio constructor on JavaScript-luokille tarkoitettu funktio, jolla voidaan luoda ja alustaa (eng. *initialize*) olio luokan sisälle (Mozilla 2019), jota kutsutaan vain komponenttia luodessa. Funktiossa contructor annetaan luokkakomponentille LightBulb tila (eng. *state*), jonka yhdeksi arvoksi (eng. *property*) määritellään powerOn muuttuja, jonka arvo saadaan komponentille annetuista syötteistä (eng. *props*).

Button-elementille lisättiin myös tapahtumakuuntelijan onClick, joka kutsuu funktiota togglePower hiiren painalluksella. Joka kerta kun togglePower-funktiota kutsutaan, se lukee luokkakomponentin LightBulb tilan ja asettaa uudeksi arvoksi vastakkaisen totuusarvomuuttujan (eng. *boolean*), kuin mitä sillä hetkellä tilassa on. Toisin sanoen, jos valo on ollut päällä totuusarvolla true, niin hiiren painalluksen jälkeen valon totuusarvo muutetaan tilaan false ja päinvastoin. Tällä tavoin LightBulb-luokkakomponentin tilaa voidaan muuttaa yksinkertaisesti hiiren painalluksella.

4 React-komponenttien testaus



Kuvio 26. Vasemmalla mahdollinen React-komponentti ja oikealla komponentin rakenne komponenttipuuna.

React-komponenttien syötteitä voidaan testata karkeasti jaoteltuna kahdella eri tavalla: prosessoida (eng. *render*) yksinkertaistettu versio komponenttipuusta tai suorittaa koko ohjelma vuorovaikutuksessa palvelimen kanssa, joka tunnetaan myös käsitteenä järjestelmätestaus. (React 2019g.) Järjestelmätestauksessa ei testata yksittäistä toteutustapaa, kuten React-komponentteja, vaan enemmänkin työnkulkua (React 2019g) esimerkiksi kykeneekö käyttäjä kirjautumaan järjestelmään ja tulostamaan pdf-raportin. Komponenttipuulla tarkoitetaan komponenttien rakentamaa hierarkiaa, jossa yksittäinen komponentti voi sisältää muita komponentteja ja nämä puolestaan toisia komponentteja ja niin edelleen. Kuviossa 26 on esimerkki, jossa Arvostelunäkymä-komponentti sisältää neljä muuta komponenttia. Tässä opinnäytetyössä käsitellään yksittäisen komponentin tai yksitasoisen komponenttipuun testausta.

4.1 Sovelluskehys ja kirjasto

Testaustyökaluja valitessa on myös otettava huomioon kaksi asiaa: todenmukaisen selainympäristön käyttämisen ja testin suoritusnopeuden jännitettä, sekä missä määrin luoda valekomponentteja (eng. *mock*) testeihin. Osa työkaluista tarjoaa todenmukaisen selainympäristön, mutta se hidastaa testien suorittamista ja se toimii epävakaammin jatkuvan integraation ja toimituksen palvelimilla (eng. lyhennys CI/CD). Toisaalta osa työkaluista tarjoaa hyvinkin nopean ja tehokkaan suorituskyvyn, mutta testit eivät vastaa selaimen todenmukaista käyttöä. (React 2019g.)

Testejä kirjoittaessa yksikkö- ja integraatiotestin raja on häilyvä. Esimerkiksi lomaketta testatessa, joka sisältää useita komponentteja, on hyvä pohtia tulisiko lomakkeen testin testata myös lomakkeen painikkeita vai tulisiko niille luoda oma testitapaus? Saako lomakkeessa olevan komponentin uudelleen kirjoitus rikkoa lomakkeen testin? (React 2019g.) Kuinka paljon lomakkeen sisältämiä komponentteja tulisi korvata niin sanotulla valekomponenteilla? Näihin valintoihin löytyy useita vastauksia ja toteutustavat tulisi aina soveltaa jokaiseen projektiin ja tiimiin erikseen (React 2019g).

Reactin dokumentaatiossa suositellaan käyttämään testaussovelluskehyksenä Jestiä ja testikirjastona React Testing Librarya tai Enzymeä (React 2019g; React 2019h). Sovelluskehyksellä tarkoitetaan ennalta laadittua komponentti-, luokka- ja/tai rajapintakokoelmaa, jonka tarkoituksena on tarjota ohjelmoijalle rakenne, jonka ympärille sovellus ohjelmoidaan. Sovelluskehystä käyttämällä ohjelmoijan ei tarvitse kehittää kyseisen osa-alueen perusratkaisuja, vaan kykenee hyödyntämään sovelluskehyksen tarjoamia ratkaisuja käyttämällä sen arkkitehtuurirakennetta ja ohjelmistorajapintaa. Kirjasto on taas pikemminkin kokoelma uudelleen käytettäviä komponentteja ja funktioita, jotka toimivat arkkitehtuuriin katsomatta. (Haikala & Mikkonen 2011, 189–191.)

Tämän opinnäytetyön kontekstissa sovelluskehys käytännössä huolehtii testien suorittamisen, sanelemalla kuinka testit tulee alustaa sekä kirjoittaa. Kuitenkaan,

sovelluskehys ei välttämättä kykene tulkitsemaan tai tuottamaan testattavia arvoja. Esimerkiksi Jest-sovelluskehys ei kykene itse tulkitsemaan React-komponenttien tuottamaa tulostetta, joten tähän tarvitaan kirjastoa. Jest testeille voisi muun muassa antaa testattavia arvoja käyttämällä Enzyme-kirjaston tarjoamia funktiota. Näitä kahta käyttämällä, Jest tarjoaa rakenteen testien kirjoittamiselle, ja Enzyme tarjoaa testeihin testattavat arvot.

4.1.1 Jest

Jest on Facebookin ylläpitämä JavaScript-testaussovelluskehys. Sitä voi käyttää muun muassa Reactin kanssa. Jest suorittaa testit rinnakkain maksimoiden suorituskyvyn sekä se tähtää toimimaan suoraan asennuksen jälkeen useimmissa JavaScript projekteissa ilman konfigurointia. (Jest 2019.) Tämän lisäksi Jest tulee oletusarvoisesti Create React App-paketin mukana (React 2019h). Create React App on valmis paketti, joka luo tarvittavan vähimmäisasennuksen React ympäristön käyttämiseen. Jestin lisäksi on olemassa muitakin testaussovelluskehyksiä JavaScriptille kuten Mocha tai Ava (React 2019i), mutta Facebookin sovelluskehityksessä käytetään testaukseen Jestiä (React 2019h), joten tämän ja muiden edellä mainittujen syiden takia, käytän tässä opinnäytetyössä Jestiä testaussovelluskehyksenä.

```
1  const summa = function (a, b) {
2    return a + b;
3  };
4
5  test("Summa 1 + 2 = 3", function () {
6    expect(summa(1, 2)).toBe(3);
7  });
```

Kuvio 27. Yksinkertaisen funktion testaaminen Jestillä.

Kuvion 27 riveillä 1–3 määritellään summa-funktio, joka palauttaa kahden luvun summan. Riveillä 5–7 käytetään Jestin tarjoamaa funktiota test, jonka ensimmäinen parametri on testin kuvaus ja toinen parametri ottaa vastaan varsinaisen tes-

tifunktion. Useimmiten funktio kirjoitetaan suoraan parametrille varattuun paikkaan. Varsinainen testi on rivillä 6, jossa Jestin funktio expect ottaa vastaan yhden arvon, joka voidaan liittää vertailufunktioihin. Tässä tapauksessa käytetään funktiota toBe, joka vertaa onko except-funktion saama arvo täysin sama kuin toBe:n saama arvo. Rivien 5-7 koodin voisi suomentaa seuraavasti: ”Testissä ’Summa 1 + 2 = 3’ funktion summa(1, 2) arvon tulisi olla 3”.

Projektissa löytyvät testit voidaan suorittaa syöttämällä komentoriville komento ”yarn test”. Tällöin kaikki projektissa löytyvät tiedostot, jotka päättyvät päätteeseen .test.js suoritetaan ja niiden tulokset syötetään komentoriville.

```
1  PASS  __tests__/jest.simple.test.js
2    √ Summa 1 + 2 = 3
3
4  Test Suites: 1 passed, 1 total
5  Tests:       1 passed, 1 total
6  Snapshots:   0 total
7  Time:        0.567s, estimated 1s
8  Ran all test suites.
```

Kuvio 27. Komentorivituloste onnistuneesta testistä.

Tämän lisäksi Jestillä voi ryhmittää useita testejä käyttäen describe-funktiota.

```
1  const summa = function (a, b) {
2    return a + b;
3  };
4
5  describe("Funktion summa testit", function () {
6    test("Summa 1 + 2 = 3", function () {
7      expect(summa(1, 2)).toBe(3);
8    });
9
10   test("Summa  2 + 2 = 4", function () {
11     expect(summa(2, 2)).toBe(4);
12   });
13 });
```

Kuvio 28. Kaksi testiä ryhmitettynä describe-funktion sisälle.

Kuviossa 28 rivillä 5 funktio describe ottaa arvoiksi kuvauksen sekä testifunktion, aivan kuten test-funktio.

```
1  PASS  __tests__/jest.simpleGroup.test.js
2    Funktion summa testit
3      √ Summa 1 + 2 = 3 (3ms)
4      √ Summa  2 + 2 = 4
5
6  Test Suites: 1 passed, 1 total
7  Tests:       2 passed, 2 total
8  Snapshots:   0 totalTime:        2.467s
9  Ran all test suites.
```

Kuvio 29. Komentorivituloste onnistuneista testeistä.

Komentorivitulosteessa onnistuneet testit ryhmitellään kuviossa 29 rivillä 2 näkyvän otsikon ”Funktion summa testit” alle, mikäli helpottaa hahmottamaan mitkä yksittäiset testit kuuluvat sen ryhmittymän sisälle. Tässä tapauksessa riveillä 3–4 näkyy kyseisen ryhmän testit.

Mikäli yksinkertainen testi kuviossa 27 suoritettaisiin virheellisillä arvoilla, komentorivi antaisi seuraavan tulosteen:

```
1  FAIL  __tests__/jest.simple.test.js
2    × Summa 5 + 2 = 3 (3ms)
3
4    ● Summa 5 + 2 = 3
5
6      expect(received).toBe(expected) // Object.is equality
7
8      Expected: 3
9      Received: 7
10
11       4 |
12       5 | test('Summa 5 + 2 = 3', function () {
13     > 6 |   expect(summa(5, 2)).toBe(3);
14         |                       ^
15       7 | });
16
17       at Object.toBe (__tests__/jest.simple.test.js:6:23)
18
19 Test Suites: 1 failed, 1 total
20 Tests:       1 failed, 1 total
21 Snapshots:   0 total
22 Time:        0.531s, estimated 1s
23 Ran all test suites.
```

Kuvio 30. Epäonnistuneen testin komentorivituloste.

Kuviossa 30 rivillä 2 kirjain x ilmaisee epäonnistuneen testin otsikon. Riveillä 6–9 ilmaistaan funktio, joka antoi virheilmoituksen ja mitä syötteitä funktion sai. Tässä tapauksessa expect-funktio sai syötteeksi arvon 7, mutta arvon olisi pitänyt olla 3. Riveillä 11–15 on katkelma koodista, joka helpottaa ohjelmoijaa paikallistamaan epäonnistuneen testin sijainnin koodin lähdetiedostossa. Tiedoston sijainti on ilmaistu rivillä 1. Rivillä 13 käytetty merkki > viittaa millä lähdetiedoston rivillä virhe on sattunut. Tässä tapauksessa virhe tapahtui lähdetiedoston eli jest.simple.test.js rivillä 6. Virheilmoituksen voisi suomentaa seuraavasti: ”Expect-funktio odotti saamansa arvon 7 olevan tismalleen sama kuin toBe-funktion vastaanottama arvo 3, mutta arvot eivät täsmää. Tiedostossa jest.simple.test.js rivillä 6 löydät vian.”

Jest sisältää useita muitakin vertailufunktiota, mutta kaikkien funktioiden tarkastelu menee reilusti tämän opinnäytetyön laajuuden ulkopuolelle. Luvussa 4.2 ja sen aliluvuissa käytännön testeissä tarkastellaan Jestin tarjoamaa mock-funktiota, jolla kyetään luomaan valekomponentteja sekä muutamia muita vertailufunktiota.

4.1.2 Enzyme

Enzyme on kirjasto, joka helpottaa React-komponenttien tulosteen testaamista. Se tarjoaa muun muassa React-komponenttipuun läpi käymistä, manipuloimista sekä simuloimista, joustavan ja intuitiivisen ohjelmistorajapinnan avulla. (Enzyme 2020). Tässä luvussa esitetyt Enzyme-testien esimerkit ovat johdettu Doddsin (2018) esittämien esimerkkien pohjalta.

```
1   function HelloWorld() {
2     return <div>Hello World!</div>;
3   }
4
5   export default class Greet extends React.Component {
6     constructor(props) {
7       super(props);
8       this.state = { show: false };
9     }
10    toggle = () => {
11      this.setState({
12        show: !this.state.show,
13      });
14    };
15
16    render() {
17      return (
18        <div>
19          <button onClick={this.toggle}>Say hello</button>
20          {this.state.show && <HelloWorld />}
21        </div>
22      );
23    }
24  }
```

Kuvio 31. Yksinkertainen React-komponentti, jossa painiketta painamalla div-elementti tekstillä ”Hello World!” ilmestyy painikkeen alle tai häviää painikkeen alta.

Kuviossa 31 rivillä 8 Greet-komponentin tila-arvo show alustetaan komponentin luonnin yhteydessä totuusarvolla false. Riveillä 10–13 on määritelty toggle-funktio, jota kutsuttaessa muuttaa tila-arvo show totuusarvon false totuusarvoksi true ja päinvastoin. Lopuksi komponentti palauttaa riveillä 18–21 yhden div-elementin, joka sisältää painikkeen sekä tila-arvo show totuusarvon ollessa true HelloWorld-komponentin. Toisin sanoen, mikäli Greet-komponentin tila-arvo show on false, komponentti palauttaa vain painikkeen. Mikäli tila-arvo show on true, komponentti palauttaa painikkeen alapuolelle komponentin HelloWorld, joka puolestaan palauttaa div-elementin tekstillä ”Hello World!”. Tätä tila-arvoa voidaan muuttaa painiketta painamalla.

Kirjoitetaan Jest-testi testaamaan komponentin toimintaa käyttämällä Enzymen shallow-funktiota. Shallow-funktio ei palauta testille koko DOM-rakennetta, vaan pelkästään komponentin ensimmäisen tason. Toisin sanoen, shallow-funktio ei palauta alikomponenttien todellisia arvoja.

```
<div>
  <button onClick={[Function]}>
    Say hello
  </button>
  <HelloWorld />
</div>
```

Kuvio 32. Näkymä mitä shallow-funktio palauttaa testattavaksi, kun komponentin Greet tila-arvo show totuusarvo on true.

```
<div>
  <button>Say hello</button>
  <div>Hello World!</div>
</div>
```

Kuvio 33. Lopullinen HTML-koodi kun komponentin Greet tila-arvo show totuusarvo on true.

Shallow-funktio ei palauta todellista HTML-näkymää, vaan viittauksen komponenttiin sekä sen alikomponentteihin. Todellisuudessa käyttäjä näkee kuvion 33 mukaisen näkymän selaimessaan. Doddsin (2018) tekemän kyselyn mukaan, useat ihmiset käyttävät shallow-funktiota sen nopeuden vuoksi. Koska shallow-funktio palauttaa viittauksen React-komponenttiin, se voi ohjata testaamaan komponentin toteutuksen yksityiskohtia.

```
test("Kutsumalla toggle funktiota, komponentti toimii", () => {
  const wrapper = shallow(<Greet />);
  expect(wrapper.state().show).toBe(false);
  wrapper.instance().toggle();
  wrapper.update();
  expect(wrapper.state().show).toBe(true);
});
```

Kuvio 34. Esimerkki huonosti kirjoitetusta testistä.

Kuviossa 34 kirjoitetussa testissä muutetaan suoraan Greet-komponentin tilaa kutsumalla sen toggle-funktiota rivillä 4. Kuten kuviossa 31 rivillä 8 Greet-komponentin tila-arvo show totuusarvoksi asetetaan false, kuvion 34 rivillä 3 testataan, että komponentti on alustettu oikein. Toggle-funktion kutsumisen jälkeen näkymää päivitetään rivillä 5 ja seuraavalla rivillä tarkistetaan, muuttuiko tila-arvo show totuusarvoksi true. Testi epäonnistuu, mikäli rivillä 3 tai 6 komponentin tila-arvo ei täsmäisi oletettuihin arvoihin.

Tämänkaltaisessa testissä ongelmia ilmenee, mikäli koodia muutettaisiin Greet-komponentin sisällä. Oletetaan, että Greet-komponentin vaatimukset muuttuvat ja ohjelmoija tekee muutoksia koodin. Hän samalla kuitenkin tekee seuraavan kirjoitusvirheen:

```
   // Alkuperäinen toteutus Greet-komponentissa
19 <button onClick={this.toggle}>Say hello</button>

   // Virheellinen toteutus Greet-komponentissa
19 <button onClick={this.togle}>Say hello</button>
```

Kuvio 35. Ohjelmoija tekee virheen ja muuttaa komponentin onClick-tapahtumakuuntelijan kutsufunktiota.

Mikäli kuvion 34 mukainen testi suoritetaan uudestaan, testi menee läpi, vaikka komponentti Greet ei enää toimi. Tämä johtuu siitä, että painiketta painaessa ohjelma kutsuu Greet-komponentin funktiota togle, jota ei ole määritelty missään. Tämä ei aiheuta ohjelmistovirhettä.

Toinen ongelmallinen tilanne kuvion 34 esittämässä testissä tulee, mikäli kutsuttavan funktion nimeä muutetaan. Oletetaan, että koodikatselmuksessa toiset ohjelmoijat antavat palautetta funktion toggle epämääräisesti nimestä ja käskevät tekemään seuraavan muutoksen:

```
    // Alkuperäinen toteutus Greet-komponentissa
10  toggle = () => {
    // Uusi toteutus Greet-komponentissa
10  handleClick = () => {
```

```
    // Alkuperäinen toteutus Greet-komponentissa
19  <button onClick={this.toggle}>Say hello</button>
```

```
    // Uusi toteutus Greet-komponentissa
19  <button onClick={this.handleClick}>Say hello</button>
```

Kuvio 36. Funktio toggle nimetään uudelleen nimelle handleClick sekä onClick-tapahtumakuuntelijan viittaus muutetaan uudelleen nimettyyn funktioon.

Muutosten jälkeen komponentin toiminta ei ole muuttunut, mutta kuvion 34 mukainen testi ei mene läpi ja aiheuttaa virheilmoituksen. Tämä johtuu siitä, että kuvion 34 rivillä 4 kutsutaan Greet-komponentin funktiota toggle, jota ei ole enää olemassa. Tämä aiheuttaa ohjelmistovirheen testissä, joka aiheuttaa virheilmoituksen ja testin epäonnistumisen. Toisin sanoen tämä aiheuttaa valheellisen epäonnistumisen testissä.

Kuvion 34 esittämä testi voidaan kirjoittaa toisella tavalla. Sen sijaan, että komponentin sisäistä tilaa muutetaan kutsumalla Greet-komponentin funktiota toggle, painikkeen painamista voidaan simuloida. Tällä tavoin vältetään nojaamasta komponentin sisäiseen toteutuksen ymmärtämiseen ja kuviossa 35 sekä 36 esitettyjen virheiden sattumista vältetään.

```
1  test("Painamalla painiketta, komponentti toimii", () => {
2    const wrapper = shallow(<Greet />);
3    expect(wrapper.contains(<HelloWorld />)).toBe(false);
4    wrapper.find("button").simulate("click");
5    wrapper.update();
6    expect(wrapper.contains(<HelloWorld />)).toBe(true);
7  });
```

Kuvio 37. Esimerkki hyvin kirjoitetusta testistä, joka on vuorovaikutuksessa komponentin kanssa ja pyrkii löytämään konkreettisia muutoksia.

Kuvion 37 rivillä 3 testataan, ettei Greet-komponentin palautusarvossa löydy HelloWorld-komponenttia. Rivillä 4 painetaan painiketta ja seuraavalla rivillä päivitetään näkymää. Rivillä 6 testi odottaa löytävän HelloWorld-komponentin.

Kuvion 37 mukainen testi suoritetaan onnistuneesti alkuperäisellä Greet-komponentin toteutuksella (kuvio 34). Tämän lisäksi kuvion 36 mukainen funktion uudelleen nimeäminen ei rikkoisi testiä, sillä kuvion 37 testi ei nojaa sisäiseen toteutukseen. Kuitenkin kuvion 35 kirjoitusvirhe painikkeen tapahtumakuuntelijassa saa testin epäonnistumaan. Toisin sanoen kuvion 37 testi toimii juuri kuten sen pitäisikin.

Kuviossa 34 toteutettu testi on vaarallinen ohjelmistokehityksen kannalta, sillä se antaa valheellista varmuutta ohjelmiston vakaudesta. Se ei myöskään kykene ottamaan kiinni rikkoutunutta komponenttia, mutta toisaalta aiheuttaa ylimääräistä työtä ja testien uudelleen kirjoittamista, vaikka komponentin todellisessa toiminnassa mikään ei muuttunut. Tämän ongelman kykenee korjaamaan siten, että kouluttaa kaikki testien kirjoittajat luomaan testejä, jotka eivät nojaa komponentin toteutuksen yksityiskohtiin. Se ei kuitenkaan estä kouluttamattomia henkilöitä, kuten harjoittelijoilta, kirjoittamasta huonoja tai jopa vaarallisia testejä. Kuten luvussa 2.5.1 todettiin, ohjelmistovirheiden löytyminen ohjelmistokehityksen myöhemmissä vaiheissa tulee maksamaan paljon.

4.1.3 React Testing Library

React Testing Library on vuonna 2018 julkaistu kirjasto, joka korvasi helmikuussa 2019 Reactin dokumentaatiossa (Wayback Machine, 2019) huomattavasti suositumman Enzyme-kirjaston (Npm-stat, 2019) ensisijaisesti suositeltuna kirjastona. React Testing Libraryn tarkoitus on estää ohjelmoijaa testaamasta React-komponenttien toteutuksen yksityiskohtia ja ohjata testaamaan DOM-elementtejä, joiden kanssa React-sivujen käyttäjäkin on vuorovaikutuksessa (Testing Library, 2019b).

```
function HelloWorld() {
  return <div>Hello World!</div>;
}

export default class Greet extends React.Component {
  constructor(props) {
    super(props);
    this.state = { show: false };
  }
  toggle = () => {
    this.setState({
      show: !this.state.show,
    });
  };

  render() {
    return (
      <div>
        <button onClick={this.toggle}>Say hello</button>
        {this.state.show && <HelloWorld />}
      </div>
    );
  }
}
```

Kuvio 38. Yksinkertainen React-komponentti, jossa painiketta painamalla div-elementti tekstillä ”Hello World!” ilmestyy painikkeen alle tai häviää painikkeen alta.

Kirjoitetaan React Testing Librarylla testi testaamaan kuviossa 38 näkyvän Greet-komponentin toimintaa. Komponentti on sama kuin luvussa 4.1.2 kuviossa 31 esitetty komponentti. Koska React Testing Library luottaa DOM:ssa näkyviin elementteihin, tulisi komponenttia testata tarkastamalla tuleeko teksti ”Hello World!” näkyviin, mikäli painiketta painetaan.

```
1  test("Painamalla painiketta, komponentti toimii", () => {
2    const { queryByText } = render(<Greet />);
3    const helloWorldHidden = queryByText("Hello World!");
4    expect(helloWorldHidden).toBeNull();
5    const helloButton = queryByText("Say hello");
6    fireEvent.click(helloButton);
7    const helloWorldVisible = queryByText("Hello World!");
8    expect(helloWorldVisible).toBeInTheDocument();
9  });
```

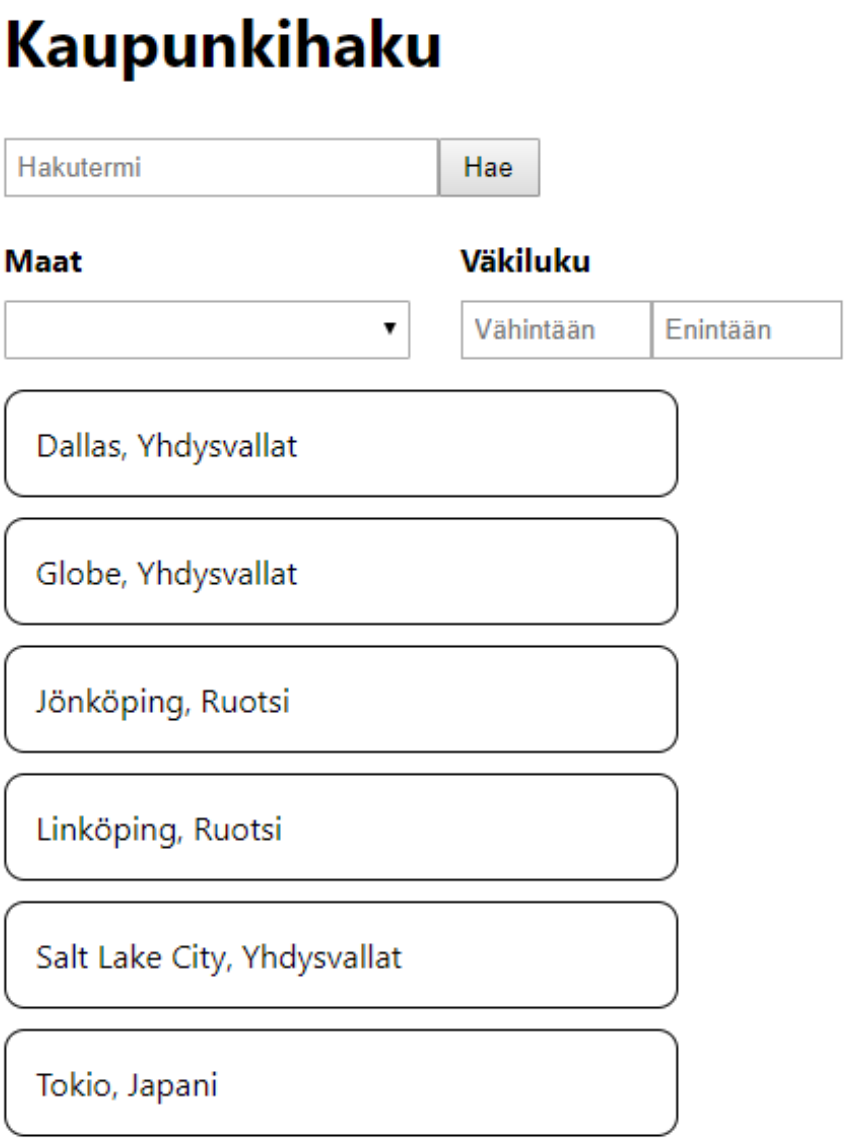
Kuvio 39. React Testing Librarylla kirjoitettu testi komponentille Greet, jonka toteutus näkyy kuviossa 38.

Kuviossa 39 rivillä 2 prosessoidaan (eng. *render*) Greet-komponentti. Funktio render antaa paluuarvoina erilaisia funktiota, joilla kyetään etsimään haluttuja elementtejä DOM:sta. Funktiolla queryByText voidaan hakea DOM:sta elementtejä tekstin perusteella. Rivillä 3 etsitään elementtiä, joka sisältää tekstin ”Hello World!” ja rivillä 4 varmistetaan ettei elementtiä löydy DOM:sta. Tämän jälkeen DOM:sta etsitään painiketta, joka sisältää teksti ”Say hello” rivillä 5. Rivillä 6 painikkeen painamista simuloidaan ja riveillä 7–8 tarkistetaan, että löytyykö tekstiä ”Hello World!” DOM:sta painikkeen painalluksen jälkeen.

Monimutkaisemmassa komponentissa voi olla hyvinkin hankalaa valita haluttuja elementtejä pelkästään näkyvien asioiden perusteella kuten otsikoiden tai painikkeiden tekstien perusteella. React Testing Library tarjoaa dokumentaatiossaan tärkeysjärjestyksen mitä tapaa hyödyntäen elementtejä tulisi etsiä testattavasta koodista. Ensisijaisesti elementit tulisi löytää käyttäjälle näkyvien asioiden kuten otsikkotekstien tai elementin sisältävän tekstin perusteella. Mikäli näkyviä tekstejä ei ole, pitäisi etsiä vaihtoehtoistekstien, kuten valokuvien tilalla olevien tekstien perusteella tai elementin tyypin perusteella. Mikäli mikään näistä ei edelleenkään auta löytämään haluttua elementtiä, voidaan viimeisenä mahdollisuutena elementeille lisätä omia test-id arvoja. (Testing Library 2019c.)

4.2 Kaupunkihaku-sovellus

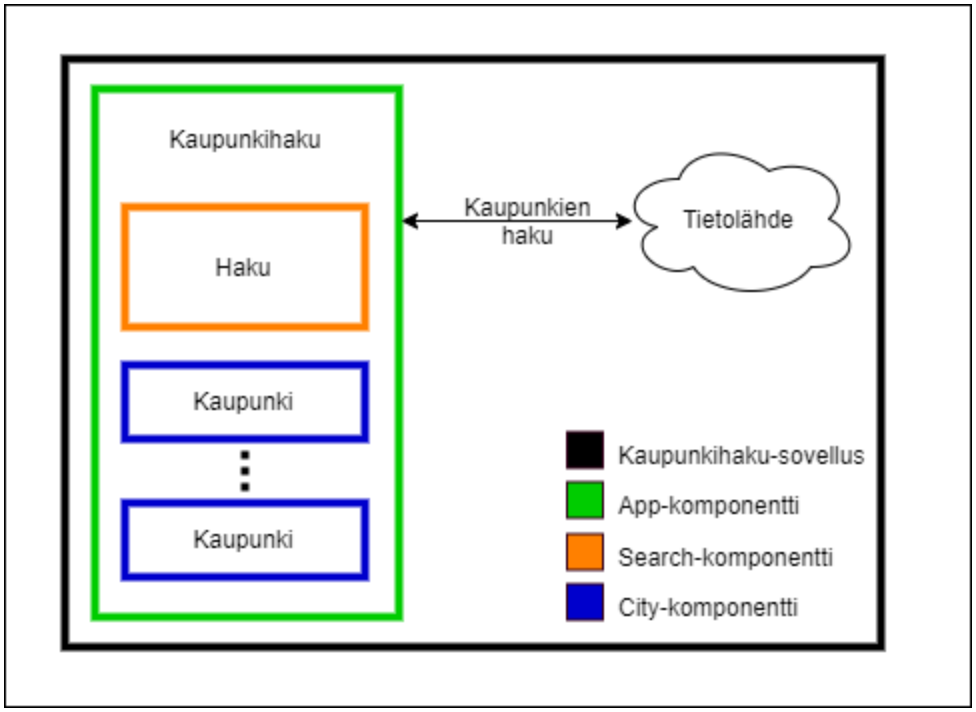
Tähän mennessä tässä opinnäytetyössä on käsitelty yksinkertaisia esimerkkejä React-komponenttien testaamisesta. Testit ovat aina kohdistuneet vain yhteen komponenttiin, eikä testeissä ole otettu huomioon React-komponentin sisältämiä muita React-komponentteja. Onkin aiheellista luoda yksinkertaistettu React-sovellus, jossa on hyödynnetty toisia React-komponentteja.



Kuvio 40. Kaupunkihaku React-sovelluksen oletusnäkymä.

Sovellus hakee tietolähteestä listauksen eri kaupungeista ympäri maailmaa. Näytettyjä kaupunkeja voi rajata sovelluksen yläosassa tarjotuilla kentillä asukasluvun, maan tai vapaa muotoisen tekstihaun perusteella. Mikäli yksittäistä kaupunkia klikkaa, se paljastaa sisältään lisätietoja, tosin tässä tapauksessa vain asukasluvun.

Kuvio 41. Kaupungin laajemmat tiedot tulevat näkyviin klikkauksen jälkeen.

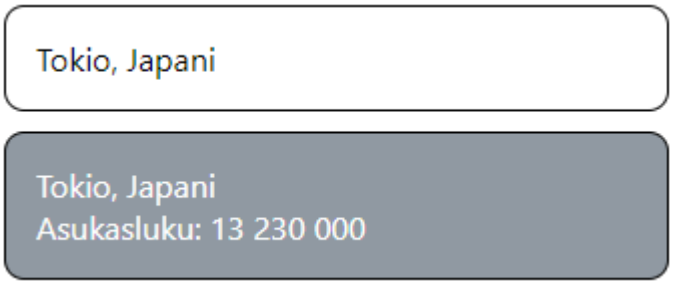


Kuvio 42. Kaupunkihaku-sovelluksen rakenne.

App-komponentti, jonka sisälle React-komponentit Search ja City sijoitetaan, hakee alustuksen (eng. *constructor*) yhteydessä tiedon ulkopuolisesta lähteestä. Tämän jälkeen tieto käsitellään ja jokaisen kaupungin tiedot tuodaan näkyviin listaan City-komponenteilla.

Tulevissa luvuissa City- ja Search-komponenteille luodaan yksikkötestit ja App-komponentille integraatiotesti. Sovellukselle voisi myös tehdä testejä varmistamaan tiedonhakuun liittyvät prosessit tai järjestelmätestin takaamaan koko sovelluksen toimivuuden, mutta ne menevät tämän opinnäytetyön laajuuden ulkopuolelle.

4.2.1 City-komponentin testaus



Kuvio 43. City-komponentti kiinni ja auki tilassa.

City-komponentti vastaanottaa syötteinä (eng. *props*) kaupungin tiedoiksi sen nimen, maan sekä asukasluvun, jotka näytetään käyttäjälle suljetussa ja laajennetussa tilassa. Laajennetussa tilassa näytetään vain asukasluku, mutta se voisi sisältää myös muita lisätietoja. Komponentti sisältää yhden tilamuuttujan open, jonka avulla voidaan näyttää ja piilottaa halutut lisätiedot hiiren klikkauksella.

```
20  render() {
21    const { city, country, population } = this.props;
22    const className = `city ${this.state.open ? 'open' : ''}`.trim();
23    return (
24      <div className={className} onClick={() => this.toggle()}>
25        <div className="title">{`${city}, ${country} `}</div>
26        {this.state.open && (
27          <div className="population">
28            Asukasluku: {formatPopulation(population)}
29          </div>
30        )}
31      </div>
32    );
33  }
```

Kuvio 44. Otos City-komponentin render-funktiosta (liite 1).

Kuviossa 44 rivillä 22 komponentin CSS-luokkaa muutetaan tilan perusteella. Tämä mahdollistaa kuviossa 43 esitetyn tilan muutokset eri värein. Rivillä 25 kaupungin nimi ja maa tuodaan käyttäjän näkyviin. Tämän lisäksi kaupungin lisätiedot näytetään riveillä 27–29 esitetyllä tavalla, mikäli komponentin tila-arvo open on saanut totuusarvon true.

Tämän pohjalta voidaan todeta, että komponentin päätehtävänä on näyttää sille annetut tiedot sekä hallita tila-arvoa open käyttäjän klikkauksen perusteella. Jokaisen komponentille syötetyn tiedon näkyvyyttä on syytä testata. Vaikkakin viimeinen testi kattaa lähes kaikkien tietojen tarkastelun, testin epäonnistuessa se ei antaisi tarpeeksi yksityiskohtaista tietoa epäonnistumisen syistä.

```
1   PASS  src/components/City/__tests__/City.test.js
2   City-komponentti
3       √ tuo kaupungin nimen näkyviin (27ms)
4       √ tuo maan nimen näkyviin (2ms)
5       √ klikatessa tuo asukasluvun näkyviin (10ms)
6       √ kahdesti klikatessa tuo asukasluvun näkyviin ja
        piilottaa asukasluvun (6ms)
7       √ kolmesti klikatessa tuo asukasluvun näkyviin,
        piilottaa asukasluvun ja tuo asukasluvun uudestaan
        näkyviin (7ms)
```

Kuvio 45. Komentorivituloste City-komponentin testeistä.

```
6   const props = {
7     city: 'Helsinki',
8     country: 'Suomi',
9     population: 652267,
10  };
```

Kuvio 46. Kaikissa City-komponentin testeissä käytetty data (liite 2).

```
13  test('tuo kaupungin nimen näkyviin', () => {
14    const { getByText } = render(<City {...props} />);
15    const city = getByText(props.city, { exact: false });
16    expect(city).toBeInTheDocument();
17  });
```

Kuvio 47. Yksittäisen City-komponentin arvon testaaminen (liite 2).

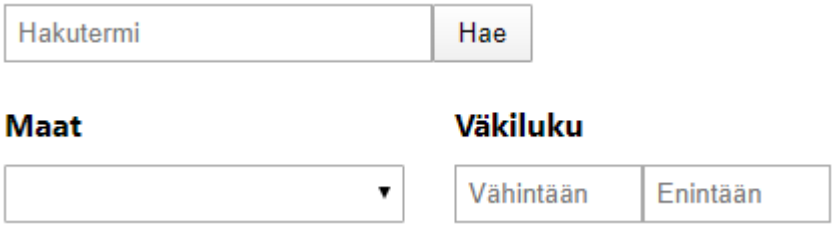
Kuviossa 47 prosessoidaan (eng. *render*) City-komponentin näkyviin kuviossa 46 näkyvillä arvoilla. Kuvion 47 rivillä 15 käytetään funktiota getByText etsimään kaupungin nimeä DOM:sta. Funktio ottaa parametreikseen kaupungin nimen sekä valinnaisia hakuasetuksia. Koska kaupungin nimen muotoilu voisi olla muukin kuin ”Kaupungin nimi, Maat” kuten kuviossa 43 on nähtävissä, on käytettävä epätarkkaa hakua. Näin ollen testi ei rikkoudu, mikäli komponentin tekstiasettelua muutettaisiin. Mikäli funktio getByText ei löytäisi mitään arvoa, se aiheuttaisi testin epäonnistumisen. Lopuksi rivillä 16 varmistetaan, että elementti löytyy DOM:sta.

```
51  test('kolmesti klikatessa tuo asukasluvun näkyviin, piilottaa asukasluvun ja tuo
    asukasluvun uudestaan näkyviin', () => {
52    const { getByText, queryByText } = render(<City {...props} />);
53    const city = getByText(props.city, { exact: false });
54    // Näkyviin
55    fireEvent.click(city);
56    const population = getByText(formatPopulation(props.population), {
57      exact: false,
58    });
59    expect(population).toBeInTheDocument();
60
61    // Piiloon
62    fireEvent.click(city);
63    const populationHidden = queryByText(formatPopulation(props.population), {
64      exact: false,
65    });
66    expect(populationHidden).toBeNull();
67
68    // Näkyviin
69    fireEvent.click(city);
70    const populationVisible = getByText(formatPopulation(props.population), {
71      exact: false,
72    });
73    expect(populationVisible).toBeInTheDocument();
74    expect(population).toStrictEqual(populationVisible);
75  });
```

Kuvio 48. City-komponentin tilanhallinnan toimivuutta kartoittava testi (liite 2).

Kuviossa 48 rivillä 53 etsitään elementti DOM:sta, joka sisältää kaupungin nimen. Tämän jälkeen klikkausta simuloidaan rivillä 55, joka kohdistetaan löydettyyn City-komponenttiin. Riveillä 56–59 tarkistetaan, että tuleeko kaupungin asukasluku näkyviin. Asukasluvun muotoilu hakua varten tehdään samalla funktiolla, jota käytetään varsinaisessa komponentissa. Mikäli asukaslukua etsittäisiin vain numeraalisella arvolla, testi ei löytäisi mitään, sillä numeraalisena muoto on 652267 ja muotoiltuna 652 267. Tämän jälkeen kaupunki elementtiä klikataan kaksi kertaa riveillä 62 ja 69 samalla tarkistaen, että teksti on joko piilotettu tai näkyvissä tilanteen mukaan. Tilanhallinnan logiikassa voisi olla virhe, joka muuttaisi tilaa vain kerran suljetusta avatuksi, muttei avatusta suljetuksi. Siksi on syytä klikata elementtiä kolme kertaa. Näiden kolmen klikkauksen jälkeen riveillä 73–74 varmistetaan, että City-komponentti näyttää asukasluvun oikeassa muodossa.

4.2.2 Search-komponentin testaus



Kuvio 48. Search-komponentti oletustilassa ilman käyttäjän syötteitä.

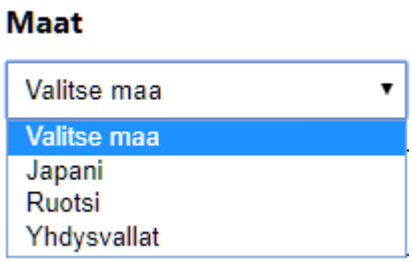
Search-komponentti vastaanottaa syötteinä (eng. *props*) listan maista, jotka sijoitetaan ”Maat” otsikon alle pudotusvalikkoon sekä funktion, jota kutsutaan ”Hae”-painikkeen klikkauksen yhteydessä. Komponentin päätehtävä on kerätä käyttäjän antamat syötteet, joita voidaan hyödyntää jatkokäsittelyssä, tässä tapauksessa App-komponentin tiedon suodatuksessa. Kappaleessa 4.2.3 tutkitaan tarkemmin mitä hakutuloksilla tehdään.

Search-komponentti käyttää neljää tilamuuttujaa tallentamaan käyttäjän syötteet, eli yksi jokaista hakukenttää kohden. Jokaisen kentän tieto tallennetaan tilamuuttujiin yksittäisen kentän arvon muutoksen yhteydessä.

```
72  <input
73    className="populationInput min"
74    placeholder="Vähintään"
75    type="text"
76    onChange={event =>
77      this.setState({
78        populationMin: formatPopulation(
79          event.target.value.replace(/[^0-9]+/g, '')
80        ),
81      })
82    }
83    value={populationMin}
84  />
```

Kuvio 49. Search-komponentin yksittäisen syötekentän tilan tallennus on nähtävissä riveillä 76–82 (liite 3).

Hakutermi- ja väkilukukentät ovat HTML input-tekstisyötekenttiä. Väkilukukentissä käyttäjän syöttämästä tiedosta suodatetaan kaikki muut paitsi numeraaliset merkit pois, kuten kuviossa 49 riveillä 78–79 on esitetty. Tämä suodatus on lisätty siitä syystä, ettei väkiluvun tallennettua syötettä tarvitse myöhemmin enää korjailla tai tarkastaa onko se sopiva jatkokäyttöä varten.



Kuvio 50. Esimerkki mitä käyttäjä näkee Maat-pudotusvalikossa.

```
52 <select
53   data-testid="countrySelector"
54   className="countrySelector"
55   onChange={event =>
56     this.setState({ selectedCountry: event.target.value })
57   }
58   defaultValue={selectedCountry}
59 >
60   <option key="default" value="">
61     Valitse maa
62   </option>
63   {countries.map(country => (
64     <option key={country} value={country}>
65       {country}
66     </option>
67   ))}
68 </select>
```

Kuvio 51. Search-komponentin Maat-pudotusvalikon toteutus (liite 3).

Maat-pudotusvalikko on HTML select-syötekenttä, jolle on myös annettu data-testid kuviossa 51 rivillä 53. Data-testid käytetään testitapauksissa, joissa HTML-elementtiin on vaikea kohdistaa testin simuloimia komentoja näkyvien elementtien perusteella. Tiedonmuutosta ei voi kohdistaa etsimällä elementtiä ”Valitse

maa” tekstillä, sillä option-elementti on osana select-elementtiä. Toisin sanoen pudotusvalikon tieto tallennetaan select-elementin arvoon. Pudotusvalikkoon luodaan ensimmäiseksi option-elementiksi ”Valitse maa”, joka näkyy oletusarvoisesti käyttäjälle rivieillä 60-62. Loput option-elementit luodaan hyödyntäen Search-komponentin syötteestä (eng. *props*) saadulla countries muuttujalla, käyttäen JavaScriptin array.map-funktiota. Map-funktio palauttaa jokaista jonon elementtiä vastaan sille annetun funktion palauttaman arvon. Kuviossa 51 riveillä 63–67 palautetaan option-elementti jokaista maata kohden. Yksittäisen elementin pudotusvalikossa näkyvä nimi on maan nimi. Samaa arvoa käytetään myös tiedontallennuksessa sisäiseen tilamuuttujaan riveillä 55-57, kun käyttäjä valitsee jonkun pudotusvalikossa näkyvistä maista kuviossa 50.

```
1  PASS  src/components/Search/__tests__/Search.test.js
2    Search-komponentti
3      √ tulee näkyviin (68ms)
4      √ hakee annetulla hakutermillä (29ms)
5      √ hakee annetulla vähimmäisväkiluvulla (11ms)
6      √ hakee annetulla enimmäisväkiluvulla (8ms)
7      √ hakee valitulla maalla (20ms)
8      √ näyttää kaikki komponentin vastaanottamat maat (6ms)
9      √ hakee kaikilla annetuilla yhtaikaa arvoilla (11ms)
```

Kuvio 52. Komentorivituloste Search-komponentin testeistä (liite 4).

Testeissä testataan jokainen kenttä yksitellen ja viimeisessä testissä kaikki kentät yhdessä. Tämä siitä syystä, että mikäli testit suoritettaisiin vain yksittäisille kentille, ei voitaisi tietää antaako Search-komponentti oikein jokaisen kentän syötteen yhtaikaa. Toisaalta, mikäli jokaista kenttää ei testattaisi yksitellen, testien epäonnistuessa tarkkaa kuvausta ei saisi mistä virhe mahdollisesti aiheutuu.

```
 9   const countries = ['Suomi', 'Ruotsi'];
10   const onSearch = jest.fn();
11   const props = { countries, onSearch };
12
13   beforeEach(() => {
14     onSearch.mockClear();
15   });
```

Kuvio 53. Search-komponentin vastaanottamat syötteet (liite 4).

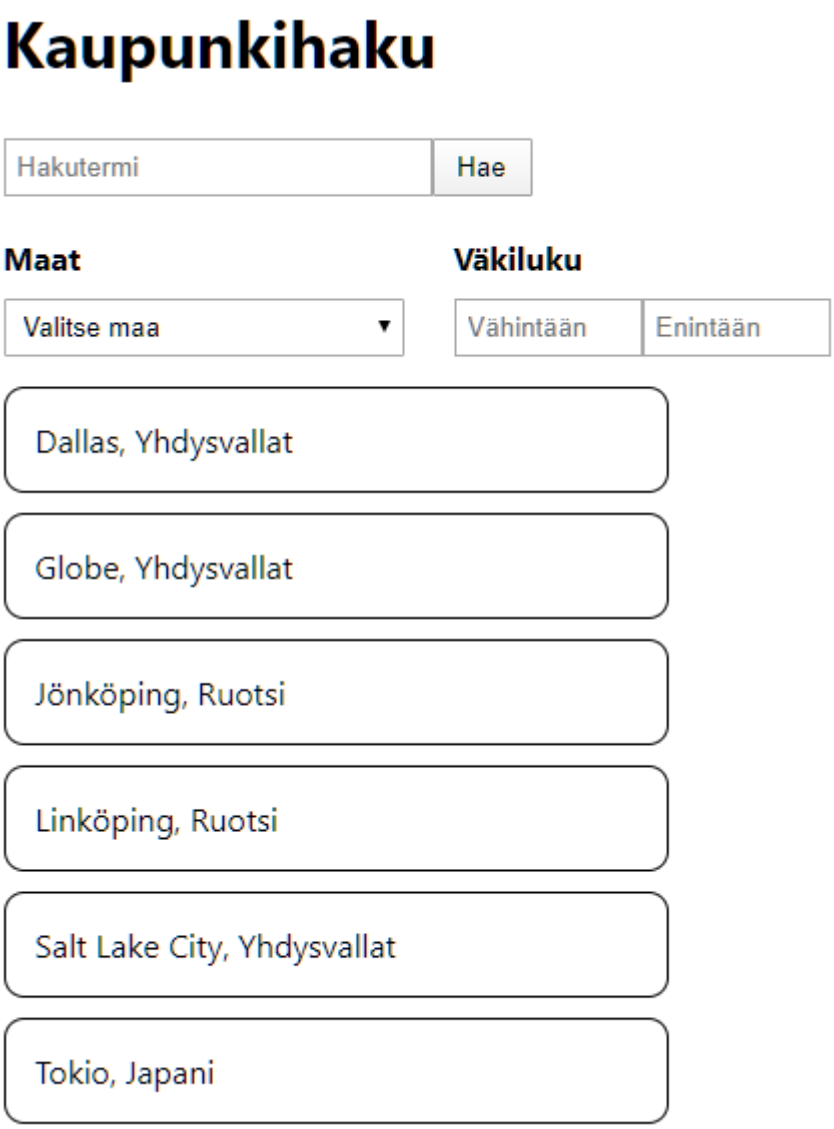
Kuviossa 53 rivillä 10 luodaan valefunktion onSearch, jota voidaan tarkastella jokaisen testin jälkeen. Tämä mahdollistaa Search-komponentin tulosteen tarkastelua ja vertailua. Riveillä 13–15 määritellään onSearch-valefunktion alustus ennen jokaista testiä. Tämä varmistaa, ettei vanhaa tietoa jää testien välillä valefunktion muistiin. Luvussa 4.2.3 tarkastellaan tarkemmin, miten App-komponentti käyttää onSearch-funktiota. Joissain tapauksissa valefunktioiden tai valekomponenttien käyttö voisi vaarantaa testattavan komponentin todenmukaisuutta, mutta tässä tapauksessa Search-komponentti luovuttaa vain omat tilatietonsa funktiota onSearch käyttäen. Toisin sanoen valefunktion käyttö ei vaikuta komponentin käyttöön millään tavalla ja toisaalta mahdollistaa näppärän tavan tarkastella Search-komponentin tulostetta.

```
87  test('hakee kaikilla annetuilla yhtaikaa arvoilla', () => {
88    const { getByTestId, getByText, getByPlaceholderText } = render(
89      <Search {...props} />
90    );
91
92    const searchButton = getByText('Hae');
93    const searchTerm = getByPlaceholderText('Hakutermi');
94    const populationMin = getByPlaceholderText('Vähintään');
95    const populationMax = getByPlaceholderText('Enintään');
96    const selectedCountry = countries[1];
97    const textValue = 'test';
98    const numericValue = formatPopulation(1000);
99
100   fireEvent.change(searchTerm, { target: { value: textValue } });
101   fireEvent.change(populationMin, { target: { value: numericValue } });
102   fireEvent.change(populationMax, { target: { value: numericValue } });
103   fireEvent.change(getByTestId('countrySelector'), {
104     target: { value: selectedCountry },
105   });
106   fireEvent.click(searchButton);
107
108   const searchResult = {
109     searchTerm: textValue,
110     populationMin: numericValue,
111     populationMax: numericValue,
112     selectedCountry,
113   };
114
115   expect(onSearch).toHaveBeenCalledWith(searchResult);
116 });
```

Kuvio 54. Search-komponentin testi (liite 4), jossa jokaiseen neljään syötekenttään annetaan syöte ja tarkistetaan palauttaako komponentti syötteet virheettömänä.

Kuviossa 54 riveillä 92–95 jokainen Search-komponentin syötekenttä etsitään sijaisarvon (eng. *placeholder*) perusteella ja viite tallennetaan muuttujaan. Riveillä 96–98 tallennetaan esimerkkisyötteet muuttujiin, jotka syötetään eri syötekentille riveillä 100–105. Kun jokaiselle syötekentälle on annettu syöte, painetaan Hae-painiketta rivillä 106, joka puolestaan laukaisee funktion onSearch. Koska onSearch on valefunktio, sen saamasta syötteestä voidaan tehdä suora vertailu Search-komponentille annetusta todellisesta syötteestä. Rivillä 115 tehdäänkin varsinainen testi, joka varmistaa, ettei Search-komponentti ole mitenkään turmellut testissä syötettyjä syötteitä.

4.2.3 App-komponentin testaus



Kuvio 55. Kuvankaappaus App-komponentista.

App-komponentti sisältää Search-komponentin sekä useamman City-komponentin. App-komponentin pääasiallinen tehtävä onkin hakea varsinainen kaupunkidata sekä sisällyttää logiikka, joka suodattaa Search-komponentin tarjoaman arvojen perusteella pois näkyvistä hakutermien ulkopuolelle jäävät kaupungit. Koska esimerkki tiedonhaku ei ole komponentin toiminnalle oleellista, ei mahdollisia tiedonhakukeinoja tarkastella lainkaan. Myös City- ja Search-komponentin toimintaa on tarkasteltu edellisissä luvuissa, joten jäljelle jää vain suodatustoiminnallisuuden tarkastelu.

```
33    const filteredCities = filterData
34      ? cities.filter(({ city, country, population }) => {
35          const results = [true];
36
37          if (filterData.selectedCountry) {
38            results.push(filterData.selectedCountry === country);
39          }
40
41          if (filterData.searchTerm) {
42            results.push(
43              sanitizeWord(`${city}${country}`).includes(
44                sanitizeWord(filterData.searchTerm)
45              )
46            );
47          }
48
49          if (filterData.populationMin) {
50            results.push(
51              sanitizeNumber(filterData.populationMin) <
52                sanitizeNumber(population)
53            );
54          }
55
56          if (filterData.populationMax) {
57            results.push(
58              sanitizeNumber(filterData.populationMax) >
59                sanitizeNumber(population)
60            );
61          }
62
63          return results.every(bool => bool);
64        })
65      : cities;
```

Kuvio 56. Suodatetut kaupungit, jota käytetään App-komponentissa näkyvien kaupunkien listaamiseen (liite 5).

Kuviossa 56 rivillä 33 käytetään ehtolausetta, jossa tarkistetaan filterData-muuttujan sisällön pätevyys. Muuttuja filterData sisältää Search-komponentin palauttamat tiedot. Mikäli filterData-muuttujan sisältö on pätevä rivillä 33, App-komponentin vastaanottamasta syötteestä (eng. *props*) suodatetaan pois näkyvistä

kaikki kaupungit, jotka eivät täytä hakukriteerejä riveillä 34–64. Mikäli filterData-muuttujan sisältö on virheellinen minkäänlaista suodatusta ei käytetä rivillä 65. Tietojen suodattamiseen käytetään JavaScriptin array.filter-funktiota hyödyksi. Se käy annetun jonon jokaisen arvon läpi ja odottaa saavansa funktion paluuarvoksi totuusarvoa true tai false. Mikäli se saa paluuarvoksi totuusarvon true, arvo säilytetään jonossa. Totuusarvon ollessa false, jonon arvoa ei palauteta lainkaan. Tällä tavoin kyetään suodatamaan helposti pitkäkin jono yksinkertaisten totuusarvovertailujen perusteella.

Koska suodatus perustuu näkyvien kaupunkien arvoihin, rivillä 35 luodaan muuttuja results, jonka jonon ensimmäiseksi arvoksi annetaan true. Mikäli filterData-muuttuja ei sisältäisi yhtäkään suodatusarvoa, kuten valittua maata riveillä 37–39, oletusarvoisesti jokainen kaupunki läpäisee suodatuksen rivillä 63 suoritettavassa vertailussa. Vertailussa käytetään JavaScriptin tarjoamaa array.every-funktiota, joka käy jokaisen jonon arvon läpi odottaen paluuarvoksi totuusarvoa true. Mikäli yksikin palautettava arvo on muu kuin true, palauttaa every-funktio puolestaan totuusarvon false. Toisin sanoen, mikäli yksikin annetuista hakutermeistä ei täsmää vertailtavan kaupungin arvoihin, kaupunki ei läpäise suodatuksesta.

Riveillä 37, 41, 49 ja 56 tarkistetaan sisältääkö hakutermi pätevää arvoa. Mikäli hakutermi on pätevä, sitä verrataan kaupungin arvoja vastaan, jonka tulos lisätään results-muuttujan hännille käyttäen array.push-funktiota. Mikäli hakutermi on virheellinen vertailua ei suoriteta lainkaan. Riveillä 37–39 verrataan, onko haettu maa sama kuin käsiteltävän kaupungin maa. Riveillä 41–47 tarkistetaan, sisältääkö kaupungin nimi tai maa vapaan tekstihaun arvoja hyödyntäen sanitizeWord-funktiota. Funktion toimintaa on selitetty auki kuviossa 57. Riveillä 49–54 tarkistetaan, onko käsiteltävän kaupungin asukasluku suurempi kuin annettu vähimmäisasukasluku. Ja viimeiseksi, riveillä 56–61 tarkistetaan, onko käsiteltävän kaupungin asukasluku pienempi annettu enimmäisasukasluku.

```
7   export function sanitizeWord(word) {
8     return word
9       .replace(/[!"#¤%&/()=?`@£$€{[\]}\\*^¨_:;,.<>+\s]+/g, '')
10      .toLowerCase();
11  }
```

Kuvio 57. SanitizeWord-funktio, joka poistaa annetusta tekstiarvosta kaikki erikoismerkit sekä muuntaa kaikki kirjaimet pieniksi kirjaimiksi (liite 5). Tämä funktio mahdollistaa tekstien vertailun helposti.

```
74  {filteredCities.map(props => (
75    <City key={props.city} {...props} />
76  ))}
```

Kuvio 58. Suodatuksen läpäisseet kaupungit prosessoidaan array.map-funktiolla, palauttaen paluuarvoksi City-komponentin.

Kun App-komponentin vastaanottamat kaupungit ovat suodatettu kuviossa 56 näkyvän koodin mukaisesti, voidaan komponentin toiminnallisuutta testata. App-komponentin toiminnassa ei tarvitse testata lainkaan toimivatko Search- ja City-komponentit oikein, sillä ne testattiin jo aiemmin. Täten App-komponentin testauksessa tuleekin keskittyä vain tiedon suodatukseen liittyvien toiminnallisuuksien testaukseen.

```
1   PASS  src/__tests__/App.test.js
2     Kaupunkihaku ohjelma
3       √ tulee näkyviin (48ms)
4       √ sisältää hakukomponentin (7ms)
5       √ näyttää kaikki kaupungit oletusarvoisesti (9ms)
6       √ näyttää kaupungit valitun maan perusteella (24ms)
7       √ näyttää kaupungit hakutermin perusteella (14ms)
8       √ ei näytä kaupunkeja väärän hakutermin perusteella (12ms)
9       √ näyttää kaupungit vähimmäisväkiluvun perusteella (16ms)
10      √ ei näytä kaupunkeja väärän vähimmäisväkiluvun perusteella (15ms)
11      √ näyttää kaupungit enimmäisväkiluvun perusteella (12ms)
12      √ ei näytä kaupunkeja väärän enimmäisväkiluvun perusteella (12ms)
13      √ funktio sanitizeWord suodattaa erikoismerkit pois
14      √ funktio sanitizeNumber suodattaa muut merkit paitsi numerot pois (1ms)
```

Kuvio 59. Komentorivituloste App-komponentin testeistä (liite 6).

Kuviossa 59 riveillä 3–4 tarkistetaan prosessoiko (eng. *render*) App-komponentti näkyviin staattiset HTML-elementit, jotka eivät ole riippuvaisia App-komponentin vastaanottamista syötteistä (eng. *props*). Rivien 5–14 testit testaavat App-komponentin toiminnallisuutta.

```
6   const actions = require('../actions');
7   const cities = [
8     {
9       city: 'Linköping',
10      country: 'Ruotsi',
11      population: 104232,
12    },
13    {
14      city: 'Jönköping',
15      country: 'Ruotsi',
16      population: 89396,
17    },
18    {
19      city: 'Helsinki',
20      country: 'Suomi',
21      population: 652267,
22    },
23  ];
24  jest.spyOn(actions, 'dataFetch').mockImplementation(() => cities);
```

Kuvio 60. App-komponentin tarvittavien arvojen alustus.

Kuviossa 60 rivillä 6 testiin sisällytetään funktio actions, jota käytetään App-komponentissa kaupunkien hakuun. Tämän funktio muutetaan valefunktioksi rivillä 24, jossa määritellään funktion paluuarvoksi riveillä 7-23 määritellyt kaupungit. Tällä tavoin testejä suorittaessa on helppo todeta näyttääkö App-komponentti oikeat kaupungit listattuina annettujen kriteerien perusteella ja toisaalta arvojen vertailu on helppoa, sillä ne ovat jokaiselle testille helposti saatavilla pelkällä viittauksella muuttujaan cities.

```
70  test.each([
71    [
72      'näyttää kaupungit hakutermin perusteella',
73      'Hakutermi',
74      'ping',
75      ['Linköping', 'Jönköping'],
76    ],
95    [
96      'näyttää kaupungit enimmäisväkiluvun perusteella',
97      'Enintään',
98      10000000,
99      ['Linköping', 'Jönköping', 'Helsinki'],
100   ],
101   [
102     'ei näytä kaupunkeja väärän enimmäisväkiluvun perusteella',
103     'Enintään',
104     1,
105     [],
106   ],
107 ])('%s', (header, placeholder, input, expectedFoundCities) => {
108   const { getByText, queryByText, getByPlaceholderText } = render(<App />);
109
110   const inputField = getByPlaceholderText(placeholder);
111   const searchButton = getByText('Hae');
112   fireEvent.change(inputField, { target: { value: input } });
113   fireEvent.click(searchButton);
114
115   const foundCities = cities.filter(({ city }) => {
116     const cityVisible = expectedFoundCities.find(
117       cityName => cityName === city
118     );
119     if (cityVisible) {
120       expect(getByText(city, { exact: false })).toBeInTheDocument();
121       return true;
122     } else {
123       expect(queryByText(city, { exact: false })).toBeNull();
124       return false;
125     }
126   });
127
128   expect(foundCities).toHaveLength(expectedFoundCities.length);
129 });
```

Kuvio 61. Jestin tarjoama test.each-funktio joka mahdollistaa samankaltaisten testien suorittamisen hyvinkin helposti. Each-funktiolle annetaan syötteinä jono, joka sisältää jono arvoja (eng. *array of arrays*). Jokainen näistä jonoista puretaan auki funktion parametreiksi rivillä 107.

Kuviossa 61 rivillä 71 on esimerkki syötteestä, jossa annetaan testille arvoiksi testin kuvausteksti, tekstimuotoinen sijaisarvo (eng. *placeholder*), syötettävä arvo sekä kaupungit, jotka odotetaan löytyvän DOM:sta. Rivillä 110 etsitään haluttu syötekenttä sijaisarvotekstin perusteella App-komponentista ja rivillä 111 hakupainike ”Hae”-tekstin perusteella. Rivillä 112 syötekenttään syötetään testattava arvo ja rivillä 113 haku suoritetaan klikkaamalla painiketta. Riveillä 115–126 määritellään muuttujalle foundCities arvoksi kaikki kaupungit, jotka löytyvät DOM:sta. Tämän muuttujajonon pituutta verrataan rivillä 128 annetun jonosyötteen expectedFoundCities pituuteen. Täten voidaan varmistaa, että kaikki halutut syötteet ovat löytyneet DOM:sta.

4.2.4 Yhteenveto

Luvun 4.2 tavoitteena oli luoda mahdollisimman ehyt kokonaisuus Kaupunkihaku-sovelluksen testaamiseksi. Testien tarkastelu aloitettiin kaikista yksinkertaisimmasta komponentista City, josta siirryttiin Search-komponentin testaamiseen ja viimeisenä testattiin monimutkaisinta komponenttia App.

Komponenttien testitulosteista tarkastelemalla voidaan havaita muutamia asioita kuviossa 45, 52 ja 59. Ensinnäkin jokaiselle komponentille suoritettiin luvussa 2.3.1 esitelty savutesti. Tällä testillä voidaan taata, että komponentti tulee käyttäjälle näkyviin, ennen mitään käyttäjän antamia syötteitä. Mikäli testi kaatuisi jo tähän, komponentin jatkotestaus olisi turhaa.

Jokaista komponenttia myös testattiin sen odotettua toiminnallisuutta vastaan. Esimerkiksi City-komponentin päätehtävä on näyttää sille annetut syötteet sekä hallita sisäistä tilamuuttujaa, jonka tilaa kykenee vaihtamaan komponenttia klikkaamalla. Siispä komponentin toiminnallisuuden näkökulmasta oli tärkeää testata, että komponentti näyttää sille annetut syötteet sekä tilanhallinta toimii oikein.

App-komponentti poikkeaa City- sekä Search-komponenteista siten, että se sisälsi React-komponentteja eli tässä tapauksessa City- ja Search-komponentit.

Onkin tärkeää olla testaamatta uudestaan näiden komponenttien toiminnallisuutta App-komponenttia testatessa. Siksi tulee rajata mikä on App-komponentin haluttu toiminnallisuus. Ilman City- ja Search-komponentteja, App-komponentti ei olisi tuonut käyttäjälle mitään muuta näkyviin, paitsi yksittäisen otsikon ”Kaupunkihaku” (liite 5). App-komponentin päätehtävä on hakea listaus näytettävistä kaupungeista sekä suodattaa tätä listausta hyödyntäen Search-komponentin tarjoamilla hakukriteereillä. Tiedonhakua ei tässä tapauksessa testattu, sillä se ei kuulunut App-komponentin vastuulle. App-komponentti vain kutsui tietolähdettä, joten halutessa voisi testata kykeneekö hakufunktio saamaan yhteyden tietokantaan, mutta se olisi mennyt opinnäytetyön laajuuden yli. Testaus keskittyikin näytettävien kaupunkien suodatuslogiikan testaamiseen. Tämän lisäksi App-komponentin sisältämät apufunktiot sanitizeWord ja sanitizeNumber testattiin, sillä ne olivat olennainen osa komponentin toimivuuden näkökulmasta.

Kaikissa testeissä käytettiin Jestiä testaussovelluskehyksenä sekä React Testing Libraryä testauskirjastona. Syy React Testing Libraryn valintaan Enzymen sijaan johtui siitä, että shallow-funktio palauttaa viittauksen React-komponenttiin, joka voi ohjata testaamaan komponentin toteutuksen yksityiskohtia (Dodds 2018). Enzymen sisältämien ominaisuuksien vuoksi, kuten shallow-funktion, sillä on vaikeampaa testata samalla tavalla kuten React Testing Libraryllä, vaikka se onkin mahdollista (Testing Library 2019b). Toisin sanoen kirjaston valinnassa päädyttiin niin sanotusti idioottivarmempaan ratkaisuun. React Testing Libraryn lähestymistapa testaamiseen on testata websivujen käyttäytymistä testaamalla DOM elementtejä ja välttää komponenttien testaamista (Testing Library 2019a).

5 Pohdinta

Tässä luvussa tarkastellaan opinnäytetyön tuloksia ja prosessia. Löydettyjä tuloksia peilataan asetettuihin tutkimuskysymyksiin sekä muuhun tutkimukseen aiheesta. Tulosten merkittävyyttä puntaroidaan myös minun näkökulmastani. Lopuksi tämän opinnäytetyön kirjoitusprosessia ja aikataulua tarkastellaan sekä vedetään viimeiset johtopäätökset.

5.1 Tavoitteet

Tämän opinnäytetyön tavoitteena oli tutkia React JavaScript -kirjastolla toteutetun websovelluksen yksikkö- ja integraatiotestaamista. Aihe oli minulle pintapuolisesti tuttu ennen opinnäytetyön aloittamista, mutta syvällisempää ymmärrystä minulla ei ollut testaamisesta. Tästä lähtökohdasta lähdin selvittämään mitä vaihtoehtoja React-komponenttien testaamiseen oli tarjolla Reactin virallisen dokumentaation puitteissa.

Hyvin aikaisessa vaiheessa opinnäytetyöprosessia kävi ilmi, että suurin osa dokumentaatiosta sekä lähdekirjallisuudesta React-komponenttien testaamisesta löytyi vain ja ainoastaan internetlähteistä. Opinnäytetyöni kirjoittamisen aikana maaliskuusta 2019 toukokuuhun 2020 React on kirjastona päivittynyt versiosta 16.8 versioon 16.13. Tämän lisäksi Reactin virallisesti suosima testauskirjasto on muuttunut React Testing Librarystä ja Enzymestä vain React Testing Libraryyn. Nämä antavat siis selkeitä viitteitä siihen, että ajantasaista informaatiota ei voi saada oikein muualta kuin suoraan Reactin virallisesta ja alati muuttuvasta dokumentaatiosta heidän internetsivustoltaan. Onneksi testaamisen yleiseen teoriaan on tehty jo vuosikymmenien ajan kirjallisuutta, joita löytyi internetin lisäksi myös paikallisista kirjastoista, joko sähköisenä tai paperisena versiona.

5.2 Tutkimuskysymykset

Ensimmäinen tutkimuskysymykseni oli ”Miksi webohjelmistoa tulisi testata?”. Vastasin tähän kysymykseen pääosin johdannossa, josta kävi ilmi testaamattoman ohjelmiston aiheuttamat kustannukset yritykselle. Tämän lisäksi luvussa 2.5.3, tuotiin esille testauksen ja valekomponenttien käytön tuovan esille mahdollisuuksia parantaa koodin laatua. Toisin sanoen, testaus tuo ohjelmistoprojekteihin koodiin laatua, mutta parantaa myös projektia suorittavan yrityksen tulosta.

Toinen tutkimuskysymykseni oli ”Mitä testausmenetelmiä on olemassa koodikieleen ja -arkkitehtuuriin katsomatta?”. Aiheitta käsiteltiin laajasti luvussa 2, tarkas-

tellen eri testaustasoja, -menetelmiä, -lajeja sekä testauksen periaatteita ja käytänteitä. Tiivistäen voidaan todeta, että testattavien komponenttien muuttuessa yhä monimutkaisemmiksi, testit tulisi kohdistaa komponenttien käyttäytymisen testaukseen, eikä niinkään mitä ohjelmapolkuja komponentin sisällä suoritetaan. Tämän lisäksi automaatiotestaus kykenee hoitamaan usein toistuvat testit, mutta manuaalisilla testeillä saadaan paremmin kiinni ohjelmiston odottamattomia toiminnallisuuksia eli virheellisiä suorituksia. Toisin sanoen sekä automaattisia että manuaalisia testejä tarvitaan mahdollisimman laajan testikattavuuden tavoittamiseksi.

Kolmas tutkimuskysymykseni oli ”Mitä tulisi ottaa huomioon React-komponenttien testauksessa?”. Varsinaiset testit suoritettiin Kaupunkihaku esimerkkiohjelmalle luvussa 4.2 React Testing Library -kirjastolla, sillä luvussa 4.1.2 todettiin Enzyme-kirjaston mahdollistavan virheellisten testien luomista. Yleisesti ottaen luvussa 4.1. tuotiin esille, että React-komponenttien testauksessa tulisi erityisesti pyrkiä kohdistamaan simuloidut syötteet käyttäjälle näkyviin HTML-elementtien arvojen perusteella, kuten otsikoilla, ja välttää täysin React-komponenttien sisäiseen toteutukseen nojaamista.

5.3 Tulosten merkittävyys

Tulokset ovat toimeksiantajalleni merkittäviä, sillä ne tarjoavat hyvän lähtökohdan React-komponenttien testaukseen. Vaikka opinnäytetyöni ei tarjoa laajoja esimerkkejä monimutkaisten komponenttien testaukseen, voi esitettyjä tuloksia hyödyntää lähes kaikkiin React-komponentteihin. Tosin opinnäytetyöstäni puuttuu esimerkkejä testien alustuksesta tapauksissa, jossa käytetään muun muassa Redux JavaScript-kirjastoa, joka vaikuttaa merkittävästi, miten Reduxiin kytketyt React-komponentit saadaan Jest-testeissä toimimaan.

Webkehityksen ja yleisesti ottaen alan kehittymisen kannalta on hyvin tärkeää lisätä testejä sovelluskehitysprosessiin, kuten johdannossa sekä luvussa 2.5.3 tuotiin ilmi. Erityisesti suomenkielisin lähdemateriaalin löytyminen internetistä on haastavampaa kuin englanninkielisen, joten opinnäytetyöni tarjoaa erityisesti

suomenkielisille React JavaScript -kirjastoon tutustuville henkilöille oivan ponnahduslaudan aiheeseen.

Opinnäytetyön esittämät tulokset ovat myös itselleni merkittäviä, sillä pääsin tutustumaan automaatiotestaukseen syvemmin sekä opin kuinka webkehittäjän arkea voidaan helpottaa erilaisten testien kautta. Automaatio muodossa kuin toisessa on itselleni hyvinkin tärkeä ja mielenkiintoinen aihe. Olen myös karttuneen työkokemuksen kautta saanut käytännössä kokea minkälaisia ongelmia testaamattomat React-komponentit aiheuttavat lisätyön muodossa. Tämän vuoksi koen saavuttaneeni merkittävän askelen eteenpäin ammatillisen osaamiseni näkökulmasta.

5.4 Muu tutkimus aiheesta

Lassi Peltolan kirjoittama opinnäytetyö ”Ohjelmistotestaus Node.js & React.js -kehitysympäristöissä” keskittyy React-komponenttien testaukseen, mutta järjestelmätason testien näkökulmasta. Koska järjestelmätason testit eivät keskity vain käyttöliittymäkomponenttien testaukseen, Peltolan opinnäytetyön tulokset eivät ole päällekkäisiä tämän opinnäytetyön tulosten kanssa. Tämän lisäksi hänen valitsemallaan testityökalulla voidaan testata muitakin kuin React-komponentteja. Mielestäni Peltolan valitsema aihe on tärkeä testauksen näkökulmasta, sillä parhaan testikattavuuden saavuttamiseksi ohjelmistoa tulee testata yksikkö- ja integraationtestien lisäksi kokonaisuutena järjestelmätason testeillä.

Bogdan Morozin kirjoittama opinnäytetyö ”Unit Test Automation of a React-Redux Application with Jest and Enzyme” on aihepiiriltään hyvin lähellä tämän opinnäytetyön aihetta. Kuten johdannossa ja luvussa 5.3 tuotiin esille, ei tässä opinnäytetyössä testattu lainkaan Redux JavaScript-kirjaston käyttöä React-komponenttien yhteydessä. Moroz käytti opinnäytetyössään testauskirjastona Enzymeä ja erityisesti shallow-funktiota. Kuten luvuissa 4.2.4 ja 5.2 todettiin, shallow-funktio aiheuttaa ongelmia testauksessa, eikä sen käyttö ole suositeltavaa.

Shallow-funktion käytöstä huolimatta Morozin suorittamat testit Redux-kirjastoa käyttäville React-komponenteille on hyvin tärkeä tutkimuksen aihe, sillä Reduxia käytetään laajasti React-komponenttien kanssa. Hänen työssään tuotiin myös esille, kuinka testeissä voidaan luoda valekomponentteja palvelinkutsuille, joka puuttui täysin tästä opinnäytetyöstä. Tämän vuoksi mielestäni Morozin löytämät tulokset ovat tärkeitä ja merkittäviä React-komponenttien testauksessa.

Tässä opinnäytetyössä keskityttiin React Testing Libraryn ohjeistuksen mukaisesti testaamaan testeissä käyttäjän näkemiä asioita. Peltolan eikä Morozin opinnäytetyö lähesty React-komponenttien testausta samasta näkökulmasta, joten tämän opinnäytetyön tulokset tukevat sekä laajentavat heidän löytämiä tuloksia. Mielestäni jokaisen opinnäytetyö auttaa testaajaa saamaan kokonaisvaltaisemman käsityksen, kuinka React-komponentteja voi testata käytännössä.

5.5 Opinnäytetyöprosessi

Opinnäytetyön tekeminen prosessina oli itselleni hyvinkin haasteellinen. Suurimpia vaikeuksia koin jatkuvasti epäonnistuneen aikataulutuksen puitteissa. Vaikka prosessin aikana oli muutamia hyvinkin tehokkaita vaiheita, joissa tekstiä tuli ja tutkimus edistyi, myöhästyin merkittävästi alustavasti aikataulustani, joka oli joulukuu 2019. Nämä samat aikataululliset vaikeudet vaivasivat myös muuta koulutyötäni.

Prosessin aikana jouduin myös opettelemaan stressinhallintaa, mikä oli itselleni aiemmin sangen vierasta. Suurimpia stressin lähteitä oli subjektiivisesti tuntuva ajan vähyys kaiken säännöllisen tekemisen ohella, vaikkakin objektiivisesti tarkasteltuna aikaa oli paikoin runsaastikin tarjolla. Tämän lisäksi kokemattomuus laajan kirjallisen projektin tuottamisesta aiheutti lisää stressiä, joka johtui asian tuntemattomuudesta.

Koin myös hyvin piinaavaksi lukea paikoin runsaan teoreettista lähdemateriaalia, jonka lainaamista jouduin opettelemaan. Asiaa ei myöskään helpottanut lainkaan osittain hyvinkin vaikeiden käännösten luominen englannista suomen kielelle,

sillä suomenkielisiä termejä ei yksinkertaisesti löydy jokaiselle tekniselle termille. Olen kuitenkin tyytyväinen valintaan tehdä opinnäytetyöni suomen kielellä.

Opinnäytetyön tekninen osuus oli kaikista miellyttävin itselleni, sillä tykkään tehdä konkreettisia asioita. Erityisesti JavaScript esimerkkien keksiminen sekä testien luominen React-komponenteille oli kaikista paras osuus opinnäytetyössäni. Siitä on myös valtavasti apua työelämääni. En kohdannut juurikaan muita teknisiä ongelmia prosessin aikana, paitsi Word-ohjelmiston kanssa. Nämä Wordin tuomat dokumentin muotoiluun liittyvät ongelmat eivät suoranaisesti vaikuttaneet opinnäytetyön lopputulokseen, mutta nakersivat merkittävästi motivaatiotani jatkaa opinnäytetyöni tekemistä joka ikinen kerta.

Mikäli aloittaisin projektin nyt alusta uudelleen, tekisin huomattavasti tarkemmin laaditun aikataulun, jossa realistisesti antaisin myös varaa mokailla. En myöskään odottaisi liian pikaista etenemistä kirjallisen osuuden kanssa. Tämän lisäksi aloittaisin projektin kokeellisella vaiheella, jossa tutkisin runsaasti eri internetistä löytyviä lähteitä käytännön testeistä ja kokeilisin niitä itse käytännössä. Uppoutuisin tarkemmin teoriaan vasta tämän jälkeen, sillä huomasin usein teoriaa lukiessani palaavani lähdemateriaaliin useampaan otteeseen, kun ymmärsin asian selkeämmin vasta soveltaessani aihetta käytännössä.

5.6 Johtopäätökset

Opinnäytetyön tuottamista tuloksista voidaan todeta suht yksinkertaisten React-komponenttien testaamisen olevan mahdollista. Kuitenkaan tuloksista ei voida vetää johtopäätöstä, että kaikki React-komponentit ovat testattavissa. Kolmannen osapuolen kirjaston kuten karttakirjaston käyttäminen React-komponenteissa voi johtaa tilanteeseen, jossa niitä ei voida testata lainkaan.

Myös aktiivisesti muuttuvasta dokumentaatiosta voidaan päätellä, ettei React-komponenteille suunnatut testikirjastot ole löytäneet niin sanottua lopullista muotoa. Myös Reactin virallinen kanta on muuttunut vuoden 2019 aikana suosimaan pelkästään React Testing Libraryä, joka kielii alati elävistä käytänteistä.

Vaikkakin ohjelmistokehitystä on harjoitettu jo vuosikymmenien ajan, silti aiheesta tuntuu löytyvän joka vuosikymmenelle uusia ulottuvuuksia. Esimerkiksi webkehitys on ohjelmistokehityksen alalajina melko tuore uutuus. React on tästä yksi selkeimmistä esimerkeistä, sillä kirjasto on ollut julkisesti saatavilla vasta vuodesta 2013 lähtien. Tämänkin takia, React-komponenttien testaamisesta riittää muillekin vielä paljon tutkittavaa. Hyviä jatkotutkimusaiheita olisi muun muassa selvittää kuinka React Testing Library toimii syvemmällä tasolla tai mikä on ollut sitä edeltäneen Enzyme-kirjaston suosion laskuun merkittävin syy. Myös Reactin kanssa käytettyjen kirjastojen kuten Reduxin käyttöä testauksen näkökulmasta olisi hyvä tutkia syvemmin.

6 Lähteet

Abramov, D. 2019. React v16.8: The One With Hooks. React blog. 06.02.2019 https://reactjs.org/blog/2019/02/06/react-v16.8.0.html. 15.4.2019.

Anastasov, M. 2019. CI/CD Pipeline: A Gentle Introduction. Semaphore. 14.3.2019. https://semaphoreci.com/blog/cicd-pipeline 04.05.2020

Dodds, K. 2018. Why I Never Use Shallow Rendering. Kent C. Dodds. 23.07.2018. https://kentcdodds.com/blog/why-i-never-use-shallow-rendering 17.10.2019

Dustin, E., Rashka, J. & Paul, J. 2008. Automated software testing: introduction, management and performance. New York: Addison-Wesley. 14.8.2019

Elliott, E. 2017. Mocking is a Code Smell. Eric Elliott, Medium. 21.10.2017. https://medium.com/javascript-scene/mocking-is-a-code-smell-944a70c90a6a 6.11.2019

Enzyme. 2020. Etusivu. https://enzymejs.github.io/enzyme/ 4.5.2020

Haikala, I & Mikkonen, T. 2011. Ohjelmistotuotannon käytännöt. Hämeenlinna: Talentum Media Oy. 23.08.2019

Jest. 2019. Etusivu. https://jestjs.io/. 13.10.2019

Jorgensen, P. 2008. Software testing: a craftsman's approach 3rd edition. New York: Auerbach Publications. 14.8.2019

Kasurinen, J. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo. 14.8.2019

Moroz, B. 2019. Unit Test Automation of a React-Redux Application with Jest and Enzyme. http://urn.fi/URN:NBN:fi:amk-2019060615157 16.5.2020

Mozilla. 2019. Constructor. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor 18.4.2019

Myers, G. 2011. The art of software testing 3rd edition. New Jersey: John Wiley & Sons, Inc. 23.08.2019

Npm-stat. 2019. React-testing-library, Enzyme https://npm-stat.com/charts.html?package=react-testing-library&package=enzyme&from=2018-01-01&to=2019-10-13 13.10.2019

Peltola, L. 2020. Ohjelmistotestaus Node.js & React.js -kehitysympäristöissä. http://urn.fi/URN:NBN:fi:amk-202002282847 16.5.2020

React 2019a. Tutorial: Intro to React. https://reactjs.org/tutorial/tutorial.html 11.3.2019

React 2019b. Introducing JSX. https://reactjs.org/docs/introducing-jsx.html 11.3.2019

React 2019c. Rendering Elements. https://reactjs.org/docs/rendering-elements.html 11.3.2019

React 2019d. Components and Props.

https://reactjs.org/docs/components-and-props.html 11.3.2019
React 2019e. Thinking in React.
https://reactjs.org/docs/thinking-in-react.html 11.3.2019
React 2019f. React.Component.
https://reactjs.org/docs/react-component.html 11.3.2019
React 2019g. Testing Overview.
https://reactjs.org/docs/testing.html 11.3.2019
React 2019h. Test Utilities.
https://reactjs.org/docs/test-utils.html 11.3.2019
React 2019i. Testing Environments.
https://reactjs.org/docs/testing-environments.html 11.3.2019
STF 2019a. Unit Testing
http://softwaretestingfundamentals.com/unit-testing/ 14.8.2019
STF 2019b. Integration Testing
http://softwaretestingfundamentals.com/integration-testing/ 14.8.2019
STF 2019c. System Testing
http://softwaretestingfundamentals.com/system-testing/ 14.8.2019
STF 2019d. Acceptance Testing
http://softwaretestingfundamentals.com/acceptance-testing/
14.8.2019
STF 2019e. Black Box Testing
http://softwaretestingfundamentals.com/black-box-testing/ 23.8.2019
STF 2019f. White Box Testing
http://softwaretestingfundamentals.com/white-box-testing/ 23.8.2019
STF 2019g. Ad hoc Testing
http://softwaretestingfundamentals.com/ad-hoc-testing/ 23.8.2019
STF 2019h. Software Testing Types
http://softwaretestingfundamentals.com/software-testing-types/
23.8.2019
STF 2019i. Smoke Testing
http://softwaretestingfundamentals.com/smoke-testing/ 23.8.2019
STF 2019j. Functional Testing
http://softwaretestingfundamentals.com/functional-testing/ 23.8.2019
STF 2019k. Usability Testing
http://softwaretestingfundamentals.com/usability-testing/ 23.8.2019
STF 2019l. Regression Testing
http://softwaretestingfundamentals.com/regression-testing/
23.8.2019
STF 2019m. Software Testing Methods
http://softwaretestingfundamentals.com/software-testing-methods/
4.5.2020
Testing Library. 2019a
https://testing-library.com/docs/guiding-principles 17.10.2019
Testing Library. 2019b
https://testing-library.com/docs/react-testing-library/intro 13.10.2019
Testing Library. 2019c
https://testing-library.com/docs/guide-which-query 12.12.2019
Wayback Machine 2019. React Test Utilities
https://web.archive.org/web/20190206050443/https://re-
actjs.org/docs/test-utils.html 13.10.2019

Wikipedia. 2019. Document Object Model.
https://fi.wikipedia.org/wiki/Document_Object_Model 13.11.2019

```jsx
import React from 'react';
import './City.css';
import PropTypes from 'prop-types';

export const formatPopulation = population =>
  Number(population).toLocaleString('fi-FI');

You, a month ago | 1 author (You)
export default class City extends React.Component {
  constructor() {
    super();
    this.state = {
      open: false,
    };
  }

  toggle() {
    this.setState({ open: !this.state.open });
  }

  render() {
    const { city, country, population } = this.props;
    const className = `city ${this.state.open ? 'open' : ''}`.trim();
    return (
      <div className={className} onClick={() => this.toggle()}>
        <div className="title">{`${city}, ${country} `}</div>
        {this.state.open && (
          <div className="population">
            Asukasluku: {formatPopulation(population)}
          </div>
        )}
      </div>
    );
  }
}

City.propTypes = {
  city: PropTypes.string,
  country: PropTypes.string,
  population: PropTypes.number,
};
```

```jsx
import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import '@testing-library/jest-dom/extend-expect';
import City, { formatPopulation } from '../City';

const props = {
  city: 'Helsinki',
  country: 'Suomi',
  population: 652267,
};

describe('City-komponentti', () => {
  test('tuo kaupungin nimen näkyviin', () => {
    const { getByText } = render(<City {...props} />);
    const city = getByText(props.city, { exact: false });
    expect(city).toBeInTheDocument();
  });

  test('tuo maan nimen näkyviin', () => {
    const { getByText } = render(<City {...props} />);
    const country = getByText(props.country, { exact: false });
    expect(country).toBeInTheDocument();
  });

  test('klikatessa tuo asukasluvun näkyviin', () => {
    const { getByText } = render(<City {...props} />);
    const city = getByText(props.city, { exact: false });
    fireEvent.click(city);
    const population = getByText(formatPopulation(props.population), {
      exact: false,
    });
    expect(population).toBeInTheDocument();
  });
```

```
34
35  test('kahdesti klikatessa tuo asukasluvun näkyviin ja piilottaa asukasluvun', () => {
36    const { getByText, queryByText } = render(<City {...props} />);
37    const city = getByText(props.city, { exact: false });
38    fireEvent.click(city);
39    const population = getByText(formatPopulation(props.population), {
40      exact: false,
41    });
42    expect(population).toBeInTheDocument();
43
44    fireEvent.click(city);
45    const populationHidden = queryByText(formatPopulation(props.population), {
46      exact: false,
47    });
48    expect(populationHidden).toBeNull();
49  });
50
51  test('kolmesti klikatessa tuo asukasluvun näkyviin, piilottaa asukasluvun ja tuo
    asukasluvun uudestaan näkyviin', () => {
52    const { getByText, queryByText } = render(<City {...props} />);
53    const city = getByText(props.city, { exact: false });
54    // Näkyviin
55    fireEvent.click(city);
56    const population = getByText(formatPopulation(props.population), {
57      exact: false,
58    });
59    expect(population).toBeInTheDocument();
60
61    // Piiloon
62    fireEvent.click(city);
63    const populationHidden = queryByText(formatPopulation(props.population), {
64      exact: false,
65    });
66    expect(populationHidden).toBeNull();
67
68    // Näkyviin
69    fireEvent.click(city);
70    const populationVisible = getByText(formatPopulation(props.population), {
71      exact: false,
72    });
73    expect(populationVisible).toBeInTheDocument();
74    expect(population).toStrictEqual(populationVisible);
75  });
76 });
```

```jsx
import React from 'react';
import './Search.css';
import PropTypes from 'prop-types';

export const defaultState = {
  populationMin: '',
  populationMax: '',
  selectedCountry: '',
  searchTerm: '',
};

export const formatPopulation = population =>
  Number(population).toLocaleString('fi-FI');

You, a month ago | 1 author (You)
export default class Search extends React.Component {
  constructor() {
    super();
    this.state = defaultState;
  }

  render() {
    const { countries, onSearch } = this.props;
    const {
      searchTerm,
      selectedCountry,
      populationMin,
      populationMax,
    } = this.state;

    return (
      <div className="search">
        <div className="textFilter">
          <input
            className="searchTerm"
            placeholder="Hakutermi"
            type="text"
            onChange={event =>
              this.setState({ searchTerm: event.target.value })
            }
            defaultValue={searchTerm}
          />
          <button
            className="searchButton"
            type="button"
            onClick={() => onSearch(this.state)}
          >
            Hae
          </button>
        </div>
```

```
          <div className="countryFilter">
            <h4>Maat</h4>
            <select
              data-testid="countrySelector"
              className="countrySelector"
              onChange={event =>
                this.setState({ selectedCountry: event.target.value })
              }
              defaultValue={selectedCountry}
            >
              <option key="default" value="">
                Valitse maa
              </option>
              {countries.map(country => (
                <option key={country} value={country}>
                  {country}
                </option>
              ))}
            </select>
          </div>
          <div className="populationFilter">
            <h4>Väkiluku</h4>
            <input
              className="populationInput min"
              placeholder="Vähintään"
              type="text"
              onChange={event =>
                this.setState({
                  populationMin: formatPopulation(
                    event.target.value.replace(/[^0-9]+/g, '')
                  ),
                })
              }
              value={populationMin}
            />
            <input
              className="populationInput max"
              placeholder="Enintään"
              type="text"
              onChange={event =>
                this.setState({
                  populationMax: formatPopulation(
                    event.target.value.replace(/[^0-9]+/g, '')
                  ),
                })
              }
              value={populationMax}
            />
          </div>
        </div>
      );
    }
  }

  Search.propTypes = {
    countries: PropTypes.array,
    onSearch: PropTypes.func,
  };
```

```
import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import '@testing-library/jest-dom/extend-expect';
import Search, {
  defaultState as defaultSearchResult,
  formatPopulation,
} from '../Search';

const countries = ['Suomi', 'Ruotsi'];
const onSearch = jest.fn();
const props = { countries, onSearch };

beforeEach(() => {
  onSearch.mockClear();
});

describe('Search-komponentti', () => {
  test('tulee näkyviin', () => {
    const { getByPlaceholderText, getByText } = render(<Search {...props} />);

    expect(getByText('Hae')).toBeInTheDocument();
    expect(getByText('Valitse maa')).toBeInTheDocument();
    expect(getByPlaceholderText('Hakutermi')).toBeInTheDocument();
    expect(getByPlaceholderText('Vähintään')).toBeInTheDocument();
    expect(getByPlaceholderText('Enintään')).toBeInTheDocument();
  });

  test.each([
    ['hakutermillä', 'searchTerm', 'Hakutermi', 'test'],
    [
      'vähimmäisväkiluvulla',
      'populationMin',
      'Vähintään',
      formatPopulation(1000),
    ],
    [
      'enimmäisväkiluvulla',
      'populationMax',
      'Enintään',
      formatPopulation(1000),
    ],
  ])('hakee annetulla %s', (label, key, placeholder, input) => {
    const { getByPlaceholderText, getByText } = render(<Search {...props} />);

    const inputField = getByPlaceholderText(placeholder);
    const searchButton = getByText('Hae');

    fireEvent.change(inputField, { target: { value: input } });
    fireEvent.click(searchButton);

    const searchResult = {
      ...defaultSearchResult,
      [key]: input,
    };

    expect(onSearch).toHaveBeenCalledWith(searchResult);
  });

```

```
59    test('hakee valitulla maalla', () => {
60      const { getByTestId, getByText } = render(<Search {...props} />);
61
62      const selectedCountry = countries[0];
63      fireEvent.change(getByTestId('countrySelector'), {
64        target: { value: selectedCountry },
65      });
66      const searchButton = getByText('Hae');
67      fireEvent.click(searchButton);
68
69      const searchResult = {
70        ...defaultSearchResult,
71        selectedCountry,
72      };
73
74      expect(onSearch).toHaveBeenCalledWith(searchResult);
75    });
76
77    test('näyttää kaikki komponentin vastaanottamat maat', () => {
78      const { getByText } = render(<Search {...props} />);
79
80      const selectedCountry1 = getByText(countries[0]);
81      const selectedCountry2 = getByText(countries[1]);
82
83      expect(selectedCountry1).toBeInTheDocument();
84      expect(selectedCountry2).toBeInTheDocument();
85    });
86
87    test('hakee kaikilla annetuilla yhtaikaa arvoilla', () => {
88      const { getByTestId, getByText, getByPlaceholderText } = render(
89        <Search {...props} />
90      );
91
92      const searchButton = getByText('Hae');
93      const searchTerm = getByPlaceholderText('Hakutermi');
94      const populationMin = getByPlaceholderText('Vähintään');
95      const populationMax = getByPlaceholderText('Enintään');
96      const selectedCountry = countries[1];
97      const textValue = 'test';
98      const numericValue = formatPopulation(1000);
99
100     fireEvent.change(searchTerm, { target: { value: textValue } });
101     fireEvent.change(populationMin, { target: { value: numericValue } });
102     fireEvent.change(populationMax, { target: { value: numericValue } });
103     fireEvent.change(getByTestId('countrySelector'), {
104       target: { value: selectedCountry },
105     });
106     fireEvent.click(searchButton);
107
108     const searchResult = {
109       searchTerm: textValue,
110       populationMin: numericValue,
111       populationMax: numericValue,
112       selectedCountry,
113     };
114
115     expect(onSearch).toHaveBeenCalledWith(searchResult);
116   });
117 });
```

```
1  import React from 'react';
2  import './App.css';
3  import Search from './components/Search';
4  import City from './components/City';
5  import { dataFetch } from './actions.js';
6
7  export function sanitizeWord(word) {
8    return word
9      .replace(/[!"#¤%&/()=?`@£$€{[\]}\\*^¨_:;,.<>+\s]+/g, '')
10     .toLowerCase();
11 }
12
13 export function sanitizeNumber(text) {
14   return Number(String(text).replace(/[^0-9]+/g, ''));
15 }
16
   You, a month ago | 1 author (You)
17 export default class App extends React.Component {
18   constructor(props) {
19     super(props);
20     this.state = {
21       cities: this.getData(),
22       filterData: null,
23     };
24   }
25
26   getData() {
27     return dataFetch().sort((a, b) => (a.city > b.city ? 1 : -1));
28   }
29
```

```
  render() {
    const { cities, filterData } = this.state;

    const filteredCities = filterData
      ? cities.filter(({ city, country, population }) => {
          const results = [true];

          if (filterData.selectedCountry) {
            results.push(filterData.selectedCountry === country);
          }

          if (filterData.searchTerm) {
            results.push(
              sanitizeWord(`${city}${country}`).includes(
                sanitizeWord(filterData.searchTerm)
              )
            );
          }

          if (filterData.populationMin) {
            results.push(
              sanitizeNumber(filterData.populationMin) <
                sanitizeNumber(population)
            );
          }

          if (filterData.populationMax) {
            results.push(
              sanitizeNumber(filterData.populationMax) >
                sanitizeNumber(population)
            );
          }

          return results.every(bool => bool);
        })
      : cities;

    return (
      <React.Fragment>
        <h1>Kaupunkihaku</h1>
        <Search
          countries={[...new Set(cities.map(({ country }) => country))].sort()}
          onSearch={data => this.setState({ filterData: data })}
        />
        {filteredCities.map(props => (
          <City key={props.city} {...props} />
        ))}
      </React.Fragment>
    );
  }
}
```

```
import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import '@testing-library/jest-dom/extend-expect';
import App, { sanitizeWord, sanitizeNumber } from '../App';

const actions = require('../actions');
const cities = [
  {
    city: 'Linköping',
    country: 'Ruotsi',
    population: 104232,
  },
  {
    city: 'Jönköping',
    country: 'Ruotsi',
    population: 89396,
  },
  {
    city: 'Helsinki',
    country: 'Suomi',
    population: 652267,
  },
];
jest.spyOn(actions, 'dataFetch').mockImplementation(() => cities);

const specialCharactes = '!"#¤%&/()=?`@£$€{[]}\\*^¨_:;,.<>+';

describe('Kaupunkihaku ohjelma', () => {
  test('tulee näkyviin', () => {
    const { getByText } = render(<App />);
    expect(getByText('Kaupunkihaku')).toBeInTheDocument();
  });

  test('sisältää hakukomponentin', () => {
    const { getByPlaceholderText } = render(<App />);
    expect(getByPlaceholderText('Hakutermi')).toBeInTheDocument();
  });

  test('näyttää kaikki kaupungit oletusarvoisesti', () => {
    const { getByText } = render(<App />);
    cities.forEach(({ city }) =>
      expect(getByText(city, { exact: false })).toBeInTheDocument()
    );
  });

```

```
46  test('näyttää kaupungit valitun maan perusteella', () => {
47    const { getByTestId, getByText, queryByText } = render(<App />);
48
49    const selectedCountry = 'Suomi';
50    fireEvent.change(getByTestId('countrySelector'), {
51      target: { value: selectedCountry },
52    });
53    const searchButton = getByText('Hae');
54    fireEvent.click(searchButton);
55
56    const foundCities = cities.filter(({ city, country }) => {
57      const sameCountry = country === selectedCountry;
58      if (sameCountry) {
59        expect(getByText(city, { exact: false })).toBeInTheDocument();
60        return true;
61      } else {
62        expect(queryByText(city, { exact: false })).toBeNull();
63        return false;
64      }
65    });
66
67    expect(foundCities).toHaveLength(1);
68  });
69
70  test.each([
71    [
72      'näyttää kaupungit hakutermin perusteella',
73      'Hakutermi',
74      'ping',
75      ['Linköping', 'Jönköping'],
76    ],
77    [
78      'ei näytä kaupunkeja väärän hakutermin perusteella',
79      'Hakutermi',
80      'Lorem ipsum',
81      [],
82    ],
83    [
84      'näyttää kaupungit vähimmäisväkiluvun perusteella',
85      'Vähintään',
86      1,
87      ['Linköping', 'Jönköping', 'Helsinki'],
88    ],
89    [
90      'ei näytä kaupunkeja väärän vähimmäisväkiluvun perusteella',
91      'Vähintään',
92      10000000,
93      [],
94    ],
```

```
    [
      'näyttää kaupungit enimmäisväkiluvun perusteella',
      'Enintään',
      10000000,
      ['Linköping', 'Jönköping', 'Helsinki'],
    ],
    [
      'ei näytä kaupunkeja väärän enimmäisväkiluvun perusteella',
      'Enintään',
      1,
      [],
    ],
  ])('%s', (header, placeholder, input, expectedFoundCities) => {
    const { getByText, queryByText, getByPlaceholderText } = render(<App />);

    const inputField = getByPlaceholderText(placeholder);
    const searchButton = getByText('Hae');
    fireEvent.change(inputField, { target: { value: input } });
    fireEvent.click(searchButton);

    const foundCities = cities.filter(({ city }) => {
      const cityVisible = expectedFoundCities.find(
        cityName => cityName === city
      );
      if (cityVisible) {
        expect(getByText(city, { exact: false })).toBeInTheDocument();
        return true;
      } else {
        expect(queryByText(city, { exact: false })).toBeNull();
        return false;
      }
    });

    expect(foundCities).toHaveLength(expectedFoundCities.length);
  });

  test('funktio sanitizeWord suodattaa erikoismerkit pois', () => {
    const expected = 'test';
    expect(
      sanitizeWord(
        `${specialCharactes}t${specialCharactes}es${specialCharactes}t`
      )
    ).toBe(expected);
  });

  test('funktio sanitizeNumber suodattaa muut merkit paitsi numerot pois', () => {
    const expected = 678;
    expect(sanitizeNumber(`6${specialCharactes}7Lorem Ipsum8`)).toBe(expected);
  });
});
```