Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Jerry Hällfors

Testausautomaation toteutus web-sovelluksessa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

24.11.2019

Tekijä Otsikko Sivumäärä Aika	Jerry Hällfors Testausautomaation toteutus web-sovelluksessa 27 sivua 24.11.2019
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Yliopettaja Auvo Häkkinen Teknologiajohtaja Jari Laurila

Insinöörityön aiheena oli testiautomaation toteutus Futunio Oy:n aSuite-tuoteperheen web-sovellukseen. Työn tavoitteena oli helpottaa henkilöstön manuaalista regressiotestausta automatisoimalla työläimmät testitapaukset ja samalla luoda pohja aSuiten-tuotteiden testausautomaatiolle tulevaisuuden jatkokehitystä varten.

Työn aikana tutkittiin myös hyvän testausautomaation periaatteita, joiden pohjalta lopulta laadittiin suunnitelma testausautomaation kehitystä varten. Hyvää testausautomaatiota voidaan tehdä, kun tiimin kaikki kehittäjät kirjoittavat suhteellisesti sopivassa määrässä eritasoisia testejä.

Työn tuloksena saatiin kehitettyä asiakaskohtaisia toiminnallisia testejä sekä sovelluksen käyttöliittymälle yksikkötestausympäristö ja useampi esimerkkiyksikkötesti, mikä on ensimmäinen askel kohti toimivaa testausautomaatiota.

Avainsanat	Testausautomaatio, Jasmine, Karma, TestCafe, Web

Author Title Number of Pages Date	Jerry Hällfors Implementation of test automation in web-software 27 pages 24 November 2019
Degree	Bachelor of Engineering
Degree Programme	Information technology
Professional Major	Software engineering
Instructors	Auvo Häkkinen, Senior Teacher Jari Laurila, Chief Technology Officer

The subject of the thesis was the implementation of test automation in a web application of Futunio corporation's aSuite product family. During the work, manual regression testing done by the staff was facilitated by automating the most demanding test cases while providing a basis for automation of aSuiten product testing for future development.

Additionally, the principles of good test automation were investigated, and a plan for the development of test automation based on these said principles was devised. Effective test automation can be achieved when all developers in the team write appropriate amount of tests at different granularity.

As a result of this work, customer-specific functional tests were developed, as well as a unit testing environment and several sample unit tests for the applications user interface which is the first step towards effective test automation.

Keywords	Test automation, Jasmine, Karma, TestCafe, Web

Sisällys

Lyhenteet

CSS	Cascading style sheets on web-kehityksessä käytetty tyylittelykieli, jolla voi muokata web-elementtien näkymää ja ulkonäköä.
DOM	Document object Model on rajapinta, joka käsittelee HTML-dokumenttia puurakenteena, jossa jokainen solmu on objekti, joka edustaa osaa dokumentista.
E2E	End-to-end, eli Toiminnallinen testi, jolla testataan sovelluksen toimivuutta käyttöliittymän kautta, simuloimalla oheislaitteita sovelluksessa.
HTTP	Hypertext transfer protocol on protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon.
HTML	Hypertext markup language on web-kehityksessä käytetty merkintäkieli, jolla voi määritellä sivun sommitelman.

1 Johdanto

Insinöörityön tavoite oli kehittää Futunio Oy:n aSuite-tuoteperheelle testausautomaatiota. Futunio Oy on ammattiliittojen ja työttömyyskassojen toimialojen erityistarpeisiin erikoistunut vuonna 1978 perustettu IT-palveluyritys. aSuite on ammattiliitoille kehitetty kokonaisvaltainen jäsenhallintajärjestelmä.

aSuiten tuotteissa ei ollut testausautomaatiota ollenkaan muutamaa poikkeusta lukuun ottamatta, vaan testaus suoritettiin manuaalisesti henkilöstön toimesta. Testausautomaation tärkeys nykyaikaisessa sovelluskehityksessä on kasvanut, sillä se on oleellinen osa sovelluksen pitkäikäisyyttä ja laadun takausta. Insinöörityössä otetaan ensimmäinen askel aSuiten tuotteiden testausautomaatiota kohti.

Tavoitteena oli ensisijaisesti helpottaa henkilöstön manuaalista regressiotestausta automatisoimalla työläimmät testitapaukset ja samalla luoda pohja aSuiten-tuotteiden testausautomaatiolle tulevaisuuden jatkokehitystä varten. Työn alussa tutustutaan aSuiteen ja sovellustestaukseen, minkä jälkeen katsotaan, miten testausautomaatio toteutettiin aSuiten kohdalla. Testausautomaation ensisijaiseksi kehittämiskohteeksi valittiin ne sovellusalueet, joissa oli eniten sovelluskoodia testattavana, mikä tässä tapauksessa ilmentyy pääosin sovelluksen käyttöliittymän testausautomaation kehityksenä. Pohja testausautomaatiolle koostui toiminnallisista testeistä ja yksikkötesteistä. Yksikkötestausta varten valittiin yksikkötestausta helpottavat teknologiat, kehitettiin testausta helpottava testipeti ja kirjoitettiin esimerkki yksikkötestejä, joista kehittäjät voivat ottaa mallia tulevaisuudessa. Tulevaa testiautomaation kehitystä varten tarkasteltiin hyvän testausautomaation periaatteita ja kehitettiin sen perusteella testausautomaatio suunnitelma, johon tutustutaan työn lopussa.

Työssä esitetyt koodipätkät ovat opinnäytetyötä varten kirjoitettua oikeaa koodia muistuttavaa pseudokoodia, jota ei todellisuudessa esiinny työn sovelluskoodissa. Pseudokoodi havainnollistaa ongelmien ratkaisut paremmin, sillä siitä voidaan jättää pois testattavan tuotteen sovelluskohtaisia toiminnallisuuksia, jotka ovat työn rajauksien ulkopuolella.

2 aSuite-tuoteperhe

aSuite on useasta sovelluksesta koostuva jäsenhallintajärjestelmä. Sen asiakkaita ovat ammattiliitot kuten Suomen lähi- ja perushoitajaliitto SuPer, Akavan erityisalat Ae, Julkisten ja hyvinvointialojen liitto JHL, Opetusalan ammattijärjestö OAJ ja Julkis- ja yksityisalojen toimihenkilöliitto Jyty.

Futunio kuvaa asiakkaille suunnatussa koosteessaan aSuitea seuraavasti:

aSuite on yhdistelmä yhteensopivia työvälineitä suomalaisten ammattiliittojen toiminnan tueksi. aSuite koostuu moduleista (kuva 1), joista jokainen ammattiliitto voi koostaa itselleen sopivan kokonaisuuden. (aSuite dokumentaatio.)

Kuva 1. aSuiten tuoteperhe (aSuite dokumentaatio)

aSuite kokonaisjärjestelmä mahdollistaa tietojen koostamisen, ajantasaisuuden ja saatavuuden. Järjestelmä tarjoaa käyttäjälle juuri hänen tarvitsemansa ja häntä kiinnostavat tiedot. Kehittyneet hakutoiminnot mahdollistavat tiedon nopean löytämisen. Kaiken perusta eli tietokanta on suunniteltu vastaamaan ammattiliittojen nykypäivän toimintaa mahdollistaen tulevaisuuden muutostarpeet. Tuoteperheen moduuleilla ammattiliitto voi koostaa itselleen kattavan kokonaisuuden toimintansa tueksi. Kokonaisuutta täydentävät koko järjestelmää palvelevat kirjautumis-, viestintä-, raportointi- ja tilastointipalvelut (aSuite dokumentaatio.)

Testausautomaatiota kehitettiin jäsenten asiointi varten kehitettyyn oma-sivuun, “Oop-peraan”. Sen avulla käyttäjät voivat muun muassa liittyä liittojen ja työttömyyskassojen jäseniksi. Testausautomaation kehitys keskittyi erityisesti jäseneksi liittymiseen.

3 Ohjelmistotestaus

Ohjelmistotestauksella varmistetaan sovelluksen toimivuus ja käytettävyys. Ohjelmisto-testausta on syytä tehdä sovelluskehityksen aikana jatkuvasti, sillä sen avulla virheet löytyvät aikaisemmassa vaiheessa ja säästävät näin projektiin käytettyjä resursseja ja ylläpitokuluja (Blundell & Torres Milano 2015: 29).

3.1 Regressiotestaus

Regressiotestaus on ohjelmiston testausta, jolla varmistetaan, että ohjelman tai koodin muutos ei ole vaikuttanut haitallisesti sovelluksen olemassa oleviin ominaisuuksiin. Kun ohjelmistoa päivitetään, muutetaan tai kehitetään, uusien vikojen ilmestyminen tai van-hojen vikojen uudelleen esiintyminen on melko yleistä. Joskus jonkin alueen ongelman korjaaminen aiheuttaa vahingossa ohjelmistovirheen toisella alueella. Toisinaan taas, kun ominaisuus suunnitellaan uudelleen, jotkut samat virheet, jotka tehtiin ominaisuuden alkuperäisessä toteutuksessa, tehdään uudelleensuunnittelussa. Siksi regressiotestaus on tärkeä osa sovelluskehitystä. Sillä varmistetaan, että uusilla koodimuutoksilla ei ole sivuvaikutuksia nykyisiin toimintoihin ja että vanha koodi toimii edelleen, kun uudet koo-dimuutokset on tehty.

Perinteisesti regressiotestauksen on suorittanut testaustiimi sen jälkeen, kun kehitystiimi on saanut työnsä valmiiksi. Regressiotestausta varten tyypillisesti tehdään suunnitelma, jossa pyritään järjestelmällisesti kattamaan mahdollisimman paljon sovelluksen toimin-nasta. Kun virhe löydetään ja korjataan, voidaan virheen paljastava testi lisätä osaksi testaussuunnitelmaa. Kun sovellukseen katsotaan tulleen tarpeeksi muutoksia, voidaan tehdä regressiotestausta suunnitelmaa noudattaen. Sovelluksen kasvaessa testaus-suunnitelma kasvaa luonnollisesti myös, jolloin testausta on syytä automatisoida.

3.2 Testausautomaatio

Testausautomaatio on testattavasta ohjelmistosta erillisen ohjelmiston käyttöä testien suorittamisen ohjaamiseksi käyttäen erilaisia testaustyökaluja ja tekniikoita. Testausautomaatiolla pyritään ehkäisemään sovelluksen osa-alueiden regressiota kehityksen aikana. Testien alussa sovellus tyypillisesti alustetaan keinotekoisesti tilaan, jossa testauksen kohteena olevaa toiminnallisuutta voidaan testata syöttämällä sovellukselle arvoja tai komentoja. Testin lopuksi saatua tulosta verrataan kyseisen toiminnallisuuden määrittelyyn.

3.2.1 Miksi testausautomaatiota tehdään?

Automatisoiduilla ohjelmistotesteillä on kolme keskeistä hyötyä: kumulatiivinen kattavuus virheiden havaitsemiselle ja virheiden kustannusten vahentämiselle, toistettavuudella säästettävä aika ja kustannukset sekä sijoitettujen resurssien tuottavuuden parantaminen. (Axelrod 2018: 5.)

On fakta, että sovellukset muuttuvat ja monimutkaistuvat käyttöikänsä aikana, ja siksi sovelluksen ominaisuuksien määrä kasvaa tasaisesti ajan myötä. Siksi riittävän kattavuuden kannalta tarpeellisten testien määrä kasvaa myös jatkuvasti. Vain 10 %:n muutos koodissa vaatii silti, että 100 % ominaisuuksista testataan. Siksi manuaalinen testaus ei voi pysyä ajan tasalla - ellei jatkuvasti lisätä testiresursseja ja sykliaikaa, testikattavuus laskee jatkuvasti. Automaatio voi auttaa kumuloimaan testitapauksia sovelluksen käyttöiän ajan niin, että sekä olemassa olevat että uudet ominaisuudet voivat olla aina testattavissa. (Hayes 2004: 5.)

Koska aSuite-tuoteperheessä ei muutamaa osa-aluetta lukuun ottamatta ollut aikaisempia testejä, on yrityksen pitänyt käyttää merkittävä määrä resursseja sovelluksen manuaaliseen testaamiseen jokaisen muutoksen jälkeen. Sovelluksen kattava manuaalinen regressiotestaus on rasite tuotteen kehitykselle. Aikarajoitteiden vuoksi manuaalisella regressiotestauksella ei voida aina kattaa koko tuotteen toiminnallisuutta, eikä se siksi aina takaa tuotteen täydellistä virheettömyyttä. Kun tuote kasvaa ja monimutkaistuu, manuaaliseen regressiotestaukseen kuluva aika kasvaa hallitsemattomasti.

Kattavalla testausautomaatiolla voidaan taata tuotteen toimivuus paremmin, vähentää manuaalisen testaukseen kuluvia resursseja, ja tuoda vakautta sovelluksen kehitykseen.

Ohjelmiston testauksen automatisointi voi viedä kehityksen alkuvaiheessa huomattavasti resursseja, mutta testausautomaatioon sijoitetut resurssit maksavat itseään takaisin, kun sovellusta laajennetaan tai sen toiminnallisuuksia muutetaan. Tällöin kattava testausautomaatio vähentää manuaalisen testauksen tarvetta.

Testien automatisointi nopeuttaa testien suorittamista kehityksen aikana. Automatisoinnin avulla testit pysyvät myös yhtenäisinä ja poistavat inhimillisistä tekijöistä johtuvia poikkeamia testausprosessissa. (Vocke 2018.)

Ajamalla automaattiset testit voidaan varmistaa se, että regressiota ei ole tapahtunut sovellukseen tehtyjen muutoksien jälkeen. Testit helpottavat uusien ominaisuuksien lisäämistä ja vanhojen muokkaamista, koska sovelluksen muokkaamisesta johtuvat virheet tulevat esille jo testien ajovaiheessa eikä loppukäyttäjien käyttäessä sovellusta. Näin testausautomaation tuoma laadunhallinta antaa osviittaa sovelluksen toimivuudesta ja ehkäisee virhetilojen syntyä, mikä helpottaa arvioimaan sovelluksen todellista kehittymistä.

Lisäksi testeillä on positiivinen vaikutus sovelluksen sisäiseen laatuun, sillä ne myös epäsuorasti parantavat sovelluksen lähdekoodin laatua. Ohjelmallisten testien kirjoittaminen myös auttaa kehittäjää ymmärtämään paremmin ominaisuuden vaatimukset ja mahdolliset ongelmakohdat, sillä testejä on mahdotonta kirjoittaa sellaisella ohjelmakoodille, jota ei ymmärrä. (Blundell & Torres Milano 2015: 29). Helposti testattavan sovelluksen suunnittelu merkitsee suunnittelua sovelluksen modulaarisuutta, laajennettavuutta ja uudelleenkäyttöä varten. Toiminnallisuuden ja testien kehittäminen rinnakkain pakottaa kehittäjän noudattamaan hyviä ohjelmistokehityksen käytäntöjä, mistä on lopulta hyötyä sovellukselle kokonaisuudessaan. (Axelrod 2018: 75.)

Kuitenkin, jotta hyvää testausautomaatiota voidaan tehdä, on testattavan sovelluksen täytynyt olla kehitetty hyvien ohjelmistokehityksen periaatteiden mukaisesti.

3.2.2 Minkälaista on hyvä testausautomaatio?

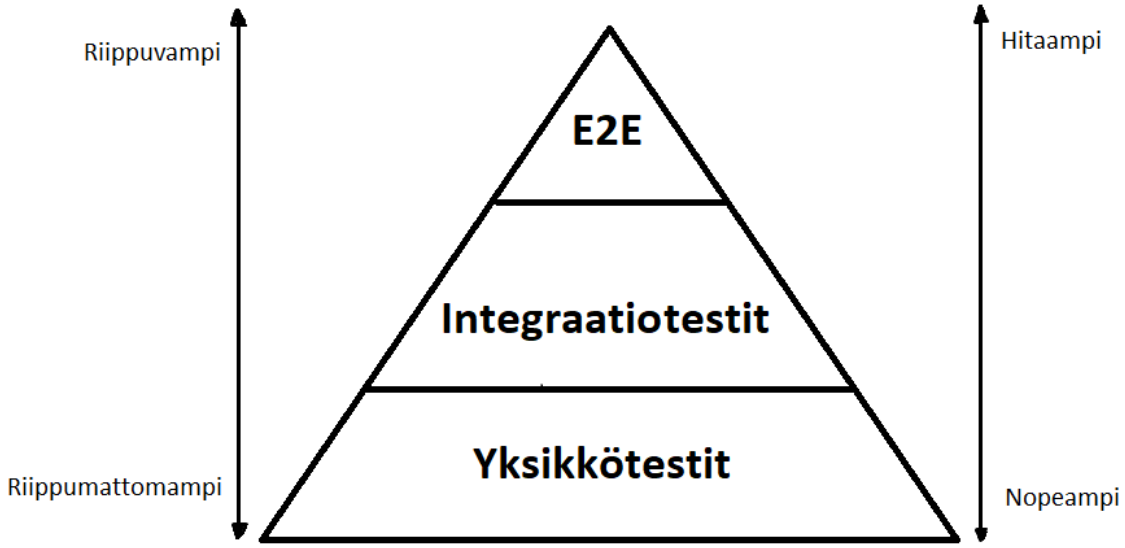
Perinteisesti ohjelmistojen testausautomaatio on toteutettu lähes yksinomaan käyttöliittymäläheisesti, toiminnallisilla testeillä, jolloin testien kirjoittaminen on jätetty testaajien vastuulle. Tämä on kuitenkin luonteeltaan epävakaa ja tehoton lähestymistapa.

Päästä päähän -testeissä on omat ongelmansa. Ne ovat tunnetusti oikullisia ja epäonnistuvat usein odottamattomista ja ennakoimattomista syistä. Melko usein niiden virhetilanteet johtuvat niin sanotuista vääristä positiivisista -tapauksista. Mitä hienostuneempi käyttöliittymä, sitä oikullisemmiksi testit yleensä tulevat. Korkeiden ylläpitokustannusten vuoksi päästä päähän -testien määrä tulisi pyrkiä vähentämään minimiin, kuten vain käyttäjille sovelluksessa eniten arvoa tuottaviin toiminnallisuuksiin. (Vocke 2018.)

Toiminnalliset testit ovat herkkiä sovelluksen muutoksille, minkä vuoksi testien ylläpitoon kuluu usein liikaa aikaa, jos niiden määrä on kasvanut liian suureksi. Kun kehittäjiä on enemmän kuin testaajia ja testausautomaation menetelmät ovat tehottomia, tulee sovelluksen kattavasta testauksesta väistämättä mahdotonta.

Mike Cohn, kirjassaan Succeeding With Agile: Software Development Using Scrum (2009: 307-315), pyrkii vastaamaan tähän ongelmaan kääntämällä tilanteen asettelun ylösalaisin. Testausautomaation piiriin on tuotava koko kehitystiimi. Tällöin testausautomaation kehityksen vastuu jakaantuu usealle henkilölle, jolloin sen ylläpitäminen on helpompaa ja testauksessa voidaan käyttää tilannekohtaisempia menetelmiä.

Cohnin mukaan hyvä testausautomaatio koostuu suhteellisesti sopivasta määrästä yksikkö- ja integraatio- sekä toiminnallisia testejä siten, että testien määrä vähenee testausasteen kasvaessa. Tavoiteltavaa testien suhdelukua kuvastaa testausautomaatiopyramidi (kuva 2), jonka pohjana toimivat yksikkötestit, keskustana integraatiotestit ja huippuna toiminnalliset päästä päähän, eli end-to-end (E2E) -testit.



Kuva 2. Testiautomaatiopyramidi

Yksikkötesteillä voidaan testata sovelluksen yksityiskohtaisempia ominaisuuksia, kuten yksittäisiä luokkia, komponentteja tai jopa funktioita. Yksikkötesteillä voidaan kattaa näiden ominaisuuksien muusta sovelluksesta riippumaton toiminnallisuus. Integraatiotesteillä katetaan useamman komponentin välinen toiminnallisuus. Tyypillisiä integraatiotestejä ovat palvelimen ja tietokannan väliset testit, missä testataan sovelluksessa käytettyjen rajapintojen toimivuutta. Lopuksi testataan sovelluksen toimivuus kokonaisuudessaan päästä päähän -testeillä, joissa käydään läpi tuotteen tyypillisimmät käyttötapaukset sovelluksen käyttöliittymän kautta.

4 Testauksen tasot

Kullakin testausmenetelmällä on omat tekniikkansa, joilla testataan sovelluksen toimivuutta eri tasoilla. Tässä kappaleessa tutustutaan menetelmien toteutukseen ja niiden vahvuuksiin ja heikkouksiin.

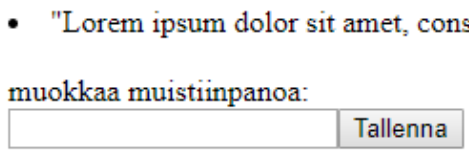
4.1 Yksikkötestaus

Yksikkötesteissä testataan komponentin muusta sovelluksesta riippumatonta toiminnallisuutta korvaamalla komponentin tarvitsemien riippuvuuksien toteutukset jollain vakiopaluuarvolla.

Yksikkötestit tarkastavat vaatimusten ja toiminnallisuuden määritteiden mukaisen toimivuuden yksikkö tasolla. Yksikköjä voivat olla esimerkiksi komponentit, luokat tai funktiot. Yksikkötestit suoritetaan yleensä komponenteille, joita voidaan testata muusta sovelluksesta erikseen, itsenäisenä yksikkönä. Tämä vaatii usein toimiakseen testitynkiä tai mock-objekteja, joilla voi simuloida yksikön eri riippuvaisuuksien toiminnallisuutta. Yksikkötestit ovat sidottuina lähdekoodiin, minkä vuoksi tyypillisesti kehittäjät itse kirjoittavat ne. Yksikkötestejä voidaan ajaa nopeasti jo sovelluksen kehitysvaiheessa, milloin usein tunnistetut viat voidaan korjata välittömästi, ilman että ne edes ehtivät ilmetä missään vian seurantaprosessissa. (Homes 2011: 59–60.)

4.1.1 Yksikkötestaustekniikat

Tarkastellaan esimerkkinä web-komponenttia (kuva 3, esimerkkikoodi 1), joka koostuu muistiinpanosta, jonka sisältö saadaan palvelimelta sekä tekstikentästä, josta voi muokata kyseistä muistiinpanoa ja painikkeesta, josta muokattu muistiinpano lähetetään palvelimelle.



Kuva 3. Esimerkkikomponentti piirrettynä selaimessa.

```
<div>
     <p class=”note-text-content”>{note}<p>
</div>
<br>
<div>
     <input class=”note-text-field”>
     <button class=”note-save-button” onclick=”saveNote()”>
          Tallenna
     </button>
</div>
```

Esimerkkikoodi 1. Esimerkkikomponentin HTML-sisältö

Jotta Web-elementtejä voidaan käsitellä, on ne ensin löydettävä sivulta. Elementtien etsimiseen on tarjolla monia tapoja, joista suurin osa perustuu CSS-valitsimiin. CSS-valitsimilla voidaan löytää elementti CSS-luokan, ID:n tai minkä tahansa muun attribuutin, attribuutti-arvon tai joidenkin edellä mainittujen yhdistelmän perusteella. Erottamalla valitsimet välilyönnillä voidaan etsiä elementin lapsia.

Testausta varten on tärkeää, että sovelluskehittäjät käyttävät yksilöiviä määritteitä, jotta elementtien löytäminen testattaessa on helppoa. Hyvä käytäntö on antaa elementeille attribuutti, joka kuvaa elementin tarkoitusta ja kontekstia, kuten esimerkiksi “note-text-field” kuvaamaan kenttää, josta voidaan asettaa muistiinpanon sisältö. Parhaimmassa tapauksessa näkymän muuttuessa kyseisen attribuutin sijaintia voi vaihtaa vastaavalle, muutosta edeltävälle paikalle, jolloin testejä ei tarvitse muuttaa (esimerkkikoodi 2). Jopa tulevien muutoksien jälkeen merkkijonot #note-text-field, #note-save-button ja #note-text-content osoittavat oikeisiin elementteihin.

```
<custom-note-display class=”note-text-content” note=”{note}”>
</custom-note-display>
<div>
     <textarea class=”note-text-field”></textarea>
     <input type =”button” class=”note-save-button”
          onclick=”saveNote()” value=”Tallenna”>
</div>
```

Esimerkkikoodi 2. Esimerkkikomponentin muutettu koodi

Esimerkkikomponenttia testatessa ensin korvattaisiin palvelimen kanssa kommunikoiva palvelu omalla toteutuksella, eli mockilla. Omassa mock-toteutuksessa komponentin muistiinpano-tiedonhaku palvelimelle "kaapattaisiin", ja hakuun vastattaisiin itse asetetulla arvolla, minkä jälkeen voitaisiin katsoa onko annettu arvo ilmestynyt komponenttiin oikealle paikalle (esimerkkikoodi 3).

```
const mockNote = 'Duis aute irure dolor';
spyOn(noteService, 'getNote').and.returnValue(mockNote);

helpers.renderComponent(testComponent);

const noteCOntent = helpers.findElement('-note-text-content');
expect(noteContet.innerText).toEqual(mockNote);
```

Esimerkkikoodi 3. esimerkki yksikkötesti 1.

Vastaavasti voidaan myös testata, mitä tapahtuu, jos palvelin vastaisikin tyhjällä arvolla. Toisaalta muistiinpanon muokkausta voitaisiin testata kirjoittamalla jotain komponentin tekstikenttään, klikkaamalla Lähetä-painiketta ja kaappaamalla komponentin kutsu palvelimelle. Kaapatun kutsun parametreistä voidaan tarkistaa, että palvelinta on kutsuttu samoilla arvoilla kuin mitä tekstikenttään on kirjoitettu. Tällä testillä varmistettaisiin, että palvelimelle lähetetään teksti, joka saadaan tekstikentästä ja että painike laukaisee kutsun palvelimelle (esimerkkikoodi 4).

Hyvä rakenne kaikille testeille on seuraava:

- alusta testitiedot
- kutsu testattavaa metodia
- vahvista, että odotetut tulokset palautetaan.

Toisin sanoen “alusta, toimi, vertaa”.

```
const mockNote = 'Duis aute irure dolor';
const noteSpy = spyOn(noteService, 'postNote');

helpers.renderComponent(testComponent);

const noteSaveButton = helpers.findElement('.note-save-button');
const noteTextField = helpers.findElement('.note-text-field');

helpers.typeText(noteTextField, mockNote);
noteSaveButton.click();

expect(noteSpy).toHaveBeenCalledTimes(1);
expect(noteSPy).toHaveBeenCalledWith(MockNote);
```

Esimerkkikoodi 4. Esimerkkiyksikkötesti 2.

4.1.2 Testipedin ja yksikkötestien toteutus

Itse yksikkötestien kirjoittaminen on huomattavasti suoraviivaisempaa kuin toiminnallisten testien kirjoittaminen. Yksikkötestaus kuitenkin tarvitsee oikein konfiguroidun testausympäristön ja testien kirjoittamista helpottavat apufunktiot.

Oopperassa käytetty KnockoutJS on monen muun modernin web-teknologian tavoin komponenttipohjainen sovelluskehys, missä komponentit muodostuvat näkymästä ja siihen kytketystä logiikasta eli mallista. Komponentit rekisteröidään sovelluskehykseen kutsumalla register-funktiota (esimerkkikoodi 5), joka saadaa KnockoutJS-sovelluskehysobjektin components ominaisuudesta.

```
ko.components.register('uusi-komponentti', {
    viewModel: komponentinLogiikka,
    template: <komponentin-html>
});
```

Esimerkkikoodi 5. Komponentin rekisteröintifunktio.

Kun komponentti on rekisteröity, sitä voidaan kutsua nimellä HTML-elementtinä (esimerkkikoodi 6).

```
<body>
     <uusi-komponentti></uusi-komponentti>
</body>
```

Esimerkkikoodi 6. Rekisteröidyn komponentin kutsuminen HTML:ssä

Kun kutsuttu komponentti piirretään ympäristöön oikein, voidaan katsoa yksikkötestausympäristön toimivan.

Rekisteröidään komponentti nimeltä example-component, jolle annetaan 3 tekstityyppistä muuttujaa, joiden arvo on kytketty komponentin näkymään (esimerkkikoodi 7).

```
<div data-bind="visible: true">
     <span data-bind="text: example_text_a"></span>
     <span data-bind="text: example_text_b"></span>
</div>
<div data-bind="text: example_text_c, visible: false"></div>
```

Esimerkkikoodi 7. Esimerkkikomponentin näkymä.

Kyseistä komponenttia voidaan kutsua nimellä testiympäristössä ja antaa sille testattavat parametrit (esimerkkikoodi 8).

```
<body>
     <example-component
          params="example_text_a: 'text a',
                    example_text_b: 'text b',
                    example_text_c: 'text c'">
     </example-component>
</body>
```

Esimerkkikoodi 8. Esimerkkikomponentin kutsuminen kolmella parametrilla HTML:ssä.

Ympäristön toimivuus voidaan varmistaa, kun KnockoutJS-sovelluskehyksen piirtofunktiota kutsuttaessa komponentin sisältö piirtyy ympäristöön oikein (esimerkkikoodi 9).

```
<body>
     <div>
          <span>text_a</span>
          <span>text_b</span>
     </div>
<body>
```

Esimerkkikoodi 9. Testiympäristöön piirretty esimerkkikomponentti.

Testipedin apufunktio (esimerkkikoodi 10), jolla alustetaan komponentti testausta varten, ottaa parametrikseen testattavan komponentin ja komponentin mahdolliset parametrit. Kun funktiota kutsutaan, komponentti rekisteröidään KnockoutJS:ssä ja asetetaan testiympäristöön. On syytä huomioida, että välillä komponentit eivät ehdi piirtyä testiympäristöön ennen kuin yksikkötesti alkaa, jolloin on syytä lisätä viive komponentin piirtämisen ja yksikkötestin välille.

```
function renderComponent (testComponent, param) {
     ko.components.register('test-component', testComponent);
     const $el = $(`<div data-bind="component:
                         {name: 'test-component',
                         params:  example_param: 'param'}">
                    </div>`);
     $('body').html($el);
     ko.tasks.runEarly();
     ko.applyBindings();
     let promise = new Promise((resolve, reject) => {
           return setTimeout(() => {
              resolve($el);
           }, 10); // wait for component to render completely
     });
     return promise;
}
```

Esimerkkikoodi 10. Komponentin testiympäristöön piirtävä apufunktio

Jotta yksikkötestejä voidaan kirjoittaa millekään sovellukselle, on luotava testattavaa sovellusta mallintava testausympäristö ja sen käyttöä helpottavat apufunktiot, eli testipeti. Testipeti luotiin käyttäen karma-runneria, joka on yksi suosituimmista web-komponenttien yksikkötestausta varten kehitetyistä testiympäristöistä. Karman testiympäristön voi konfiguroida identtiseksi testattavan sovelluksen kanssa. Tällöin testattava web-

komponentti voidaan sijoittaa karman HTML-dokumenttiin. Kun komponentti on piirretty Karmaan, voidaan sitä käsitellä tyypillisillä DOM-funktioilla.

Oopperan web-moduulien hallintaan käytettiin RequireJs-kirjastoa, jota varten Karma tarjoaa omat konfigurointityökalut. Karmalle voidaan antaa sovelluksen RequireJs-konfiguraatio lähes sellaisenaan, jolloin Karma saa viitteet kaikkiin sovelluksessa käytettäviin moduuleihin ja niiden tiedostopolkuihin. Karma-testiympäristö alustetaan hakemalla tiedostopolkujen perusteella kaikki sovelluksen tarvitsemat tiedostot, joista kukin asetetaan testiympäristöön script-elementtien sisälle. On kuitenkin syytä erotella tiedostopoluista kolmannen osapuolen kirjastojen yksikkötestit siten, että kirjastot saadaan testiympäristön piiriin, mutta niiden testejä ei ajeta. Tiedoston haun yhteydessä voi sanella, mitkä tiedostot ajetaan saman tien.

Testiympäristöä varten luotiin tiedosto, johon kopioitiin sovelluksen startup-tiedostosta kaikki osat, jotka olivat oleellisia sovelluksen alustamiselle. Kun tiedosto ajetaan, saadaan testiympäristöstä testattavan sovelluksen kaltainen.

Yksikkötestien kirjoittamisessa käytettiin Jasminea. Jasmine on web-sovelluksien testaamista varten kehitetty kaiken kattava yksikkötestaussovelluskehys, joka tarjoaa testaukselle oleelliset työkalut, kuten vertailu- ja mock-funktiot. Jasminen vertailu-funktioilla voidaan verrata testattavaa arvoa oikeaan arvoon, ja mikäli verrattavat oliot eivät täsmää, antaa Jasmine siitä kuvaavan virheilmoituksen. Jasminen vertailu-funktioilla voi testata muun muassa olion totuudellisuutta (ei null tai undefined), identtisyyttä (deep equal) ja samanarvoisuutta, joiden lisäksi Jasmineen voi lisätä omia vertailuja. Mock-funktiot auttavat tekemään koodin välisten linkkien testaamisen helpoksi poistamalla funktion todellisen toteutuksen kaappaamalla funktioon liittyvät kutsut ja näissä kutsuissa välitetyt parametrit.

Erään komponentin määritelmässä saneltiin, että käyttäjälle tulisi näyttää info-teksti jäsenyyden tilasta vain, jos asiakas on jäsen. Näinkin yksinkertaisen toiminnallisuuden manuaalinen testaus vaatii suhteellisen suuren työpanoksen. Testikäyttäjällä on ensin täytettävä ja lähetettävä jäsenhakemus, minkä jälkeen jäsenhakemus täytyy hyväksyä jäsenhallintaohjelmassa, jolloin vasta testaukseen vaaditut esiehdot täyttyvät.

Yksikkötestiä käyttämällä toiminnallisuus voidaan testata nopeammin sekä kirjoitettu testi voidaan ajaa nopeasti jatkossa.

Yksikkötestestissä voidaan korvata komponentin lähettämät pyynnöt palvelimelle ja antaa paluuarvoiksi omat käsin syötetyt arvot. Kun komponentti kysyy käyttäjän tietoja, tilaksi saadaan jäsennäkymä vastaamalla kyselyyn jäsentä vastaavalla arvolla. Näkymästä voidaan etsiä määritelmän tekstille varattu paikka ja varmistaa, että teksti on näkyvissä (esimerkkikoodi 11).

```
spyOn(userService, 'isMember').and.returnValue(true);
const infoTextMock = 'Lorem ipsum dolor sit amet';
spyOn(translateService, 'translateInfoText')
     .and.returnValue(infoTextMock);

const testComp = helpers
     .renderComponent(testComponent);
const infoText = testComp
     .findElement('#membership-info-text');

expect(infoText).toBeTruthy();
expect(infoText.innerText).toEqual(infoTextMock);
```

Esimerkkikoodi 11. Komponentin yksikkötesti

Näin yksikkötestin myötä toiminnallisuuden testaukseen kuluva aika tippuu millisekunteihin.

4.2 Integraatiotestaus

Integraatiotestauksen määrittely on testipyramidin kerroksista yleisluonteisin, mutta pohjimmiltaan se tarkoittaa useamman komponentin tai palvelun välisen toimivuuden testausta. Integrointitestit testaavat toimivatko itsenäisesti kehitetyt ohjelmistoyksiköt oikein kun ne on kytketty yhteen.

Vaikka termiä "integraatiotesti" käytetään hyvin laajasti, tälle käsitteelle ei ole yhtään tiivistä määritelmää. Integraatiotesti on testi, joka on tasoltaan korkeampi kuin yksikkötesti, mutta matalampi kuin toiminnallinen testi (Axelrod 2018: 122.) Kuten toiminnallisella

testillä, integraatiotestillä saadaan yksikkötestejä parempi varmuus komponenttien välisestä toiminnasta, mutta se ei vaadi toimiakseen yhtä montaa riippuvuutta kuin toiminnallinen testi.

Toisin kuin toiminnallisissa testeissä, integraatiotesteissä käytetään vähemmän komponentteja, jolloin testeissä löydettyjen vikojen syy on helpompi rajata. Vian perimmäisen syyn tunnistaminen on entistä vaikeampaa, kun samanaikaisesti integroitujen komponenttien lukumäärä kasvaa lisättyjen rajapintojen suuren määrän vuoksi (Homes 2013: 60).

Hitaiden osien, kuten tiedostojärjestelmien ja tietokantojen, integraatiotestaaminen on paljon hitaampaa kuin yksikkötestien ajaminen mock-objekteja käyttäen. Integraatiotestejä voi myös olla vaikeampaa kirjoittaa kuin pieniä ja eristettyjä yksikkötestejä, koska on huolehdittava integroitavien osien ajamisesta osana testejä. Integraatiotestien etuna on silti luottamus siihen, että sovellus toimii oikein kaikkien niiden ulkoisten osien kanssa, joiden kanssa sen tarvitsee keskustella, missä yksikkötesteistä ei ole apua. (Vocke 2018.) Sovelluksen toimivuudelle saadaankin hyvä varmuus, kun useamman kattavasti testatun komponentin yhteistoiminta varmistetaan vielä integraatiotesteillä.

Tyypillisiä integraatiotestejä ovat palvelimen ja tietokannan välisen toimivuuden testaus. Hyvä nyrkkisääntö on, että integraatiotestejä tulisi kirjoittaa ainakin niille sovellusosille, joissa siirretään tietoa, kuten esimerkiksi:

- Kutsut palvelun REST-rajapintaan
- Lukeminen ja kirjoittaminen tietokantoihin
- Kutsut toisen sovelluksen rajapintaan
- Kirjoittaminen tiedostojärjestelmään

(Vocke 2018).

Integraatio testillä voitaisiin esimerkiksi testata, että tietokantaan voidaan tallentaa uutta dataa kutsumalla palvelinta. Esimerkki integraatiotestillä voitaisiin lähettää palvelimelle post-kutsulla uusi objekti, ja varmistaa, että objekti on tallentunut testitietokantaan hakemalla tallennettua objektia palvelimelta get-pyynnöllä (esimerkkikoodi 12).

```
const person = {name: ”John”, age: 32};
api.post(’/person’, person);

const result = api.get(’/person/John’);

expect(result).toBeTruthy();
expect(result).toEqual(person);
```

Esimerkkikoodi 12. Esimerkki integraatiotesti.

Kun integraatiotestejä kirjoitetaan, oman sovelluksen lisäksi täytyy ajaa myös komponenttia, jonka kanssa integroidaan. Jos testataan integraatiota tietokannan kanssa, on testitietokantaa ajettava testien suorittamisen ajan. Jotta voidaan testata tiedostojen lukemista levyltä, on tiedosto tallennettava levylle ja sitten ladattava integraatiotestissä. (Vocke 2018.)

Kun kirjoitetaan Integraatiotestejä kahden komponentin välillä, jotka sijaitsevat samassa sovelluksessa, tulisi testin tarvitsemia komponentteja ajaa paikallisesti. Kun on testattava, että oman sovelluksen komponentti toimii osana jonkin ulkoisen palvelun tarjoajan komponenttia voi testaamisesta tulla vaivalloista. Tällaista integraatiotestien tekemistä voidaan helpottaa käyttämällä testiklooneja.

Kun testikloonit ovat identtisiä testattavan komponentin kanssa, voidaan testata kaikki integroitavan komponentin keskinäiset toimivuudet aktivoimatta täydellistä instanssia komponentista. Tämä ei välttämättä ole iso asia, jos komponentit ovat erillisiä moduuleja samassa sovelluksesta, mutta sillä on merkitystä, jos integroitava komponentti on erillinen palvelu, joka vaatii omat käännöstyökalut, ympäristönsä ja verkkoyhteydet. (Fowler 2018.)

Integraatiotesteissä testikloonit voivat olla esimerkiksi tilannevedoksia mallinnettavasta sovelluksesta. Jotta testiklooneja voidaan käyttää integraatiotesteissä, on jaksoittain suoritettava sopimustesti. Sopimustesti on integraatiotestistä erillinen kokonaisuus, jossa kutsutaan mallinnettavan komponentin palveluja ja katsotaan, että paluuviesti vastaa edelleen testiklooniin tallennettua tilannevedosta.

Sopimustesteillä tarkistetaan ulkoisten palvelupyyntöjen sopimuksen eheys, mutta ei välttämättä sen tarkkaa sisältöä. Usein testiklooniksi tallennetaan tilannevedos integroitavan komponentin vastauksesta, koska tiedon muodolla on enemmän merkitystä varsinaisen tiedon sijaan. Tässä tapauksessa sopimustestissä on tarkistettava, että vastauksen tietorakenne on sama, vaikka kenttien todelliset arvot olisivat muuttuneet. (Fowler 2011.) Sopimustestin epäonnistumisen tulisi käynnistää toimenpiteen, jossa testikloonin toiminnallisuus päivitettäisiin vastaamaan todellista toiminnallisuutta.

Komponenttien integroimiseksi on monia tapoja, jokaisella on etuja ja haittoja. Kehittäjien on tärkeää ymmärtää ohjelmiston arkkitehtuuri, eri komponenttien vaikutus ja koordinoida ne kehitys- ja suunnittelutiimien kanssa ennen päätöksentekoa yhdestä integrointimenetelmästä. (Homes 2013: 60.)

4.3 Toiminnallinen testaus

Toiminnallisilla eli päästä päähän -testeillä testataan sovelluksen toimintaa käyttöliittymän kautta eli loppukäyttäjän näkökulmasta. Usein päästä päähän -testit testaavat sovelluksen käyttäjille tyypillisten käyttötapausten toimivuutta kokonaisuudessaan. Toiminnallisten testien tarkoitus ei siis ole olla kattava, sillä testien ajo vie muita testejä huomattavasti pidempään. Toiminnalliset testit ovat tyypillisesti skriptejä, jotka simuloivat oheislaitteiden toimintaa sovelluksessa.

Toiminnalliset testit toteutettiin TestCafe-sovelluskehyksellä, joka on yksi suosituimmista web-sovelluksien päästä päähän -testausten sovelluskehyksistä. TestCafen kehityskielinä toimivat web-kehittäjille tutut TypeScript ja JavaScript, ja TestCafe tarjoaa kattavan määrän toimintoja sovelluksen testaamista varten. TestCafen kehityskielenä käytettiin TypeScriptiä, joka on Microsoftin kehittämä Javascript-pohjainen kehityskieli, joka tukee tyypityksiä, luokkia ja periytymistä. TypeScriptiä suositaan web-kehityksessä, sillä se tarjoaa selkeämpää koodin rakennetta ja luettavuutta.

Testien toteutuksessa käytettiin funktionaalisille testeille tyypillistä "sivuobjekti"-kaavaa (esimerkkikoodi 13). Kullekin testin aikana käydylle sivulle luodaan luokka, jolla on instanssimuuttujinaan sivulle oleelliset web-elementit kuten painikkeet ja tekstikentät.

Luokat sisältävät funktioita, jotka suorittavat tyypillisiä käyttäjän toimintoja sivulla käyttäen apunaan aikaisemmin mainittuja luokan instanssimuuttujia elementtien paikantamiseen.

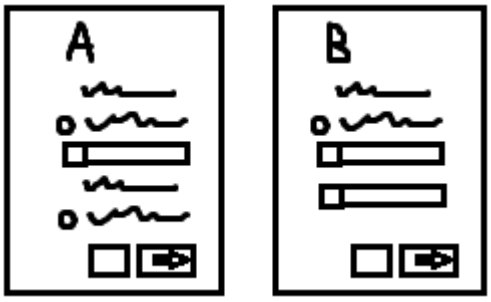
```
class PersonInfoFormPage
     firstNameInput = input#first-name’;
     lastNameInput = ‘input#second-name’;
     emailInput = ‘input#email’;
     nextPageButton = ‘button#next-page’;

     fillForm(info) {
          element(this.firstNameInput).setText(info.firstName);
          element(this.lastNameInput).setText(info.lastName);
          element(this.emailInput).setText(info.email);
          element(this.nextPageButton).click();
     }
}
```

Esimerkkikoodi 13. Sivuobjektin luokka

Sivuobjektikaavan perussääntö on, että kaavan pitäisi antaa asiakasohjelman tehdä sivulla mitä tahansa ja nähdä siellä kaiken mitä käyttäjä voi nähdä. Sen pitäisi myös tarjota rajapinta, jolla on helppo ohjelmoida ja piilottaa alla oleva pienoisohjelma. Eli tekstikentän käsittelyä varten käytettävissä tulisi olla apufunktioita, jotka ottavat ja palauttavat merkkijonon, valintaruuduissa tulisi käyttää boolean-elementtejä ja painikkeita tulisi kuvastaa toiminnallisilla funktio-nimillä. Sivuobjektin tulisi kapseloida toiminnallisuus, jota tarvitaan tietojen etsimiseen ja käsittelemiseen itse käyttöliittymän ohjaimessa. Hyvä nyrkkisääntö on kuvitella konkreettisen ohjaimen muuttaminen - missä tapauksessa sivuobjektin rajapinnan ei pitäisi muuttua. Sivuobjekti on klassinen esimerkki kapseloinnista - se piilottaa käyttöliittymän rakenteen ja pienoisohjelman yksityiskohdat muista komponenteista (testeistä). Kuten kapseloinnissa yleensä, tällä on kaksi etua. Kun käyttöliittymää manipuloivan logiikka rajataan yhteen paikkaan sitä voidaan muokata siellä vaikuttamatta järjestelmän muihin komponentteihin. Seurauksellisena etuna on, että se tekee asiakas (testin) -koodista helpommin ymmärrettävän, koska olemassa oleva logiikka liittyy testin tarkoitukseen, eikä koodi ole käyttöliittymää käsittelevien yksityiskohtien täyttämä. (Fowler 2013.)

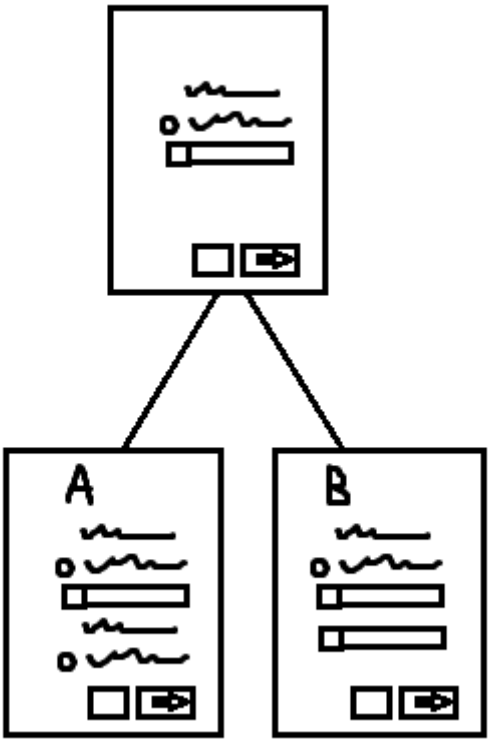
Sivuobjektikaavan avulla koodia voidaan siis abstrahoida, mikä helpottaa testien luettavuutta ja ylläpitoa. Kun testattava sovellus muuttuu, voidaan muutosta koskettavan sivun sivuobjektia muuttaa vastaamaan uutta toteutusta, jolloin testitapauksiin ei tarvitse koskea.

aSuite-tuoteperheellä on useampi asiakas, joilla jokaisen asiakaskohtainen sivu näyttää hieman erilaiselta, kuten eri tyylittelyt, värit tai eri kentät. Esimerkiksi sovelluksen käyttäjätiedot sivulla asiakkaan A näkymässä käyttäjällä voisi olla valinnanvarana tilata tai olla tilaamatta asiakkaan oheistuotteita, mitä kuvastaisi valintanappi, kun taas asiakkaalla B voisi olla tarjolla useampi tuote, mitä kuvastaisi alasvetoikkuna (kuva 4). Sivujen erilaisuus tuottaa päänvaivaa sovelluksen testauksen automatisoinnissa, sillä jokaista testattavaa käyttötapausta kohden on tehtävä jokaiselle asiakkaalle hieman erilainen testi, vaikka sivut ovat yleispiirteisesti samat.



Kuva 4. Piirustus kahden eri asiakkaan (A ja B) sivun näkymästä.

Jokaiselle asiakkaalle voitaisiin tehdä oma projekti, joka sisältäisi asiakaskohtaiset sivuobjektit, ja tehdä jokaiselle asiakkaalle omat testit. Tämä ei kuitenkaan ole kovin hieno ratkaisu, sillä eroavuuksistaan huolimatta sivujen rakenne on suurin piirtein sama, ja käyttötapaukset noudattavat samaa prosessia. Näin testeihin tulisi paljon toistuvaa koodia, jolloin pieni muutos testattavan sovelluksen toiminnassa vaatisi lukuisia muutoksia testien koodissa. Koodin toistuvuutta voidaan ehkäistä hahmottamalla sivujen samanlaisuuksia ylemmän tason sivuun, joka koostuu kaikille asiakkaille yhteisistä elementeistä. Esimerkkitapauksessamme sivun kaikki muut elementit paitsi Oheistuote-kenttä kuuluisi ylemmän tason sivuun (kuva 5).



Kuva 5. Piirustus kahden eri asiakkaan sivujen näkymistä ja sivujen näkymiä yhdistävät elementit hahmoteltuna ylemmän tason sivuun.

Yläsivukaava voidaan toteuttaa hyödyntämällä testeissä olio-ohjelmoinnille tyypillistä periytymis-paradigmaa, jota tukee TestCafen toinen kehityskieli TypeScript. Näin voidaan tehdä yhteen luokkaan kaikille asiakkaille yhteiset toiminnallisuudet, jotka voidaan periä asiakaskohtaiseen sivuobjektiluokkaan, jolloin luokan vastuulle jää vain asiakaskohtaiset ominaisuudet (esimerkkikoodi 14).

```
class Brand_A_PersonInfoFormPage extends PersonInfoFormPage {
     ssInput = 'input#ssn';

     fillForm(info) {
          element(this.ssnInput).typeText(info.ssn);
          super.fillForm(info);
     }
}
```

Esimerkkikoodi 14. Asiakaskohtainen sivuobjekti, joka perii asiakkaille yhteiset toiminnallisuudet.

Yksi testattavista käyttötapauksista on jäsenliittyminen. Asiakkaasta riippumatta kaikki liittyjät käyvät läpi samat sivut, joissa on aina käytössä useampi kaikille asiakkaille yhteinen elementti. Voidaan siis luonnehtia abstrakteja sivuobjektiluokkia, joissa on instanssimuuttujina kaikille asiakkaille yhteiset elementit ja funktiot. Abstrakteille luokille voidaan tehdä toteuttava asiakaskohtainen luokka, jolla on omien asiakaskohtaisten ominaisuuksien lisäksi yläluokasta perityt, kaikille asiakkaille yhteiset elementit ja funktiot. Jäsenliittymishakemusta tehtäessä voidaan kutsua asiakaskohtaisen sivuobjektin "Täytä lomake" -funktiota, joka puolestaan kutsuu toteutuksensa jälkeen yläluokan vastaavaa funktiota.

Lopuksi jokaiselle asiakkaalle luotiin luokka, jonka instanssimuuttujina ovat asiakkaan omat sivuobjektit ja funktioina tyypilliset käyttötapaukset, joita halutaan testata (esimerkkikoodi 15). On syytä huomioida, että vaikka olisi hyvien käytäntöjen mukaista pyytää sivu, jolle siirrytään, niin tässä tapauksessa se ei ole käytännöllistä, sillä sivujen järjestys ei ole aina sama.

```
class Brand_A {
     private personInfoFormPage
          = new Brand_A_PersonInfoFormPage();
     private memberShipFormPage =
          = new Brand_A_MembershipFormPage();
     private paymentFormPage
          = new Brand_A_PaymentFormPage();
     private finalApplicationPage
          = new Brand_A_FinalApplicationPage();

     public sendAppliation(info) {
          this.personInfoFormPage.fillForm(info.personInfo);
          this.membershipInfoFormPage
               .fillForm(info.membershipInfo);
          this.paymentFormPage.fillForm(info.paymentInfo);
          this.finalApplicationPage.sendApplication();
     }
}
```

Esimerkkikoodi 15. Asiakasluokka.

Asiakasluokan funktioina on testattavia käyttötapauksia ja instanssimuuttujina niissä tapauksissa käytettäviä sivuobjekteja.

Kun testattiin käyttäjän liittymistä liiton jäseneksi, haasteeksi ilmeni testien toistettavuus, sillä liiton jäseneksi voi oletusarvoisesti liittyä vain kerran. On siis keksittävä tapa aloittaa jokainen testi tuoreella käyttäjällä. Huonoin tapa olisi testijäsenen erottaminen liitosta automatisoimalla kyseinen prosessi jäsenhallintasovelluksessa, TestCafella. Testit ovat riippuvaisia jäsenen tietojen nollauksesta, jolloin automatisoidun jäsenen erottamisprosessin epäonnistuessa myös prosessia seuraavat testit epäonnistuisivat.

Parempia ratkaisuja olisivat esimerkiksi päivittäin uusilla testikäyttäjillä alustettava testitietokanta tai palvelimelle luotava endpoint, jota kutsumalla voidaan nollata tietyn käyttäjän arvot tietokannassa. aSuiten testausautomaation toteutuksessa päädyttiin jälkimmäiseen ratkaisuun, koska se oli helpointa toteuttaa. Kyseinen ratkaisu osoittautui varsin hyödylliseksi, sillä näin testien alustus onnistuu HTTP-kutsulla, joka on helppo tehdä TestCafesta käsin. Käyttäjän nollaus nopeutti myös manuaalista testausta, koska se poisti tarpeen tehdä uusi testikäyttäjä jokaisen testin jälkeen.

5 Testaussuunnitelma

Hyvä testausautomaation ylläpito vaatii usean henkilön työpanosta. Kun yksikkötestausta helpottava testipeti ja useampi esimerkkitesti ovat valmistuneet, on aika tehdä testausautomaatiosta osa koko kehitystiimin prosesseja.

Koska testipedin kehitykseen eivät osallistu sen loppukäyttäjät eli kehitystiimin kehittäjät, on syytä testata testipedin käyttökelpoisuutta ensin käytännössä. Automatisoitujen testien kirjoittamista voidaan ensin kokeilla yhdellä loppukäyttäjällä ennen niiden tuomista kehitystiimiin, jolloin mahdolliset ilmi tulevat puutteet voidaan korjata tuhlaamatta useamman kehittäjän aikaa.

Kun testipeti on todettu hyväksi, tulee jokainen kehittäjä kouluttaa testausautomaation käyttöön. Jokaisen kehittäjän on kyettävä kirjoittamaan testit tekemilleen toiminnallisuuksille, sillä kullekin toiminnallisuudelle voivat tehdä testit vain henkilöt, jotka ymmärtävät kyseisen toiminnallisuuden määritelmät. Lisäksi kehittäjien on pystyttävä korjaamaan rikkomansa testit. Muuten testien uskottavuus kärsii testien antaessa niin sanottuja vääriä positiivisia virheviestejä, kun testit jäävät päivittämättä kehityksen aikana.

Testausautomaatioprosessien kehittyessä siirtymäaikana voidaan kehittää testiautomaatiota strategisesti valittuihin sovelluskohtiin, joista Axelrod kirjoittaa seuraavasti:

Ensisijaisesti olisi syytä keskittyä ominaisuuksiin, jotka tuovat eniten arvoa yritykselle ja asiakkaille. Virheiden löytäminen näistä ominaisuuksista ja niiden ehkäiseminen vaikuttavat suoraan tuotteen tuomiin tuloihin. Jos komponentti tai ominaisuus on tarkoitus korvata pian, jolloin sen koko toiminnallisuus tulee muuttumaan, ei ole syytä luoda testejä sille, vaan odottaa, kunnes uusi toiminnallisuus kehittyy, ja vasta sitten kehittää uudet testit. Jos ominaisuus on erittäin vakaa eikä sitä tulla muuttamaan pian, ei ole myöskään kustannustehokasta kattaa sitä automatisoinnilla. On hyvin pieni riski, että muutos sovelluksessa vaikuttaa jotenkin kyseiseen ominaisuuteen. Tämä pätee erityisesti vanhoihin komponentteihin, jotka saattavat olla järjestelmässä erittäin kriittisiä, mutta joiden sovelluskoodiin kukaan ei uskalla koskea. Korkeampi tuotto automaatioon sijoitetulla työlle saadaan etsimällä riskialttiimpia ominaisuuksia. Jos ominaisuus on valmis ja toimii oikein, mutta suunnitelmana on korvata sen taustalla oleva tekniikka tai tehdä massiivinen uudelleenkorjaus sen sisäiseen rakenteeseen, on ominaisuus loistava ehdokas testiautomaatiolle. Testejä ei tulisi muuttaa, kun perustana oleva tekniikka tai sisäinen rakenne muuttuu, ja niiden tulisi jatkua edelleen sen jälkeenkin varmistaen, että poistuva käyttäytyminen säilyy. (Axelrod 2018: 79.)

Kun toimivan testausautomaation menetelmät on tuotu kehitystiimiin, voidaan tehdä testaussuunnitelma. Pitkäjänteinen tavoite on kattaa mahdollisimman paljon sovelluksen koodista testeillä. Testausautomaation välittömiä hyötyjä voidaan saada hyvällä strategialla.

Kun automatisoitua testiä kehitetään rinnakkain sen testaaman toiminnallisuuden kehittämisen kanssa, sovelluskehittäjän ja testiautomaatiokehittäjän välinen yhteistyö on paljon parempaa, mikä tuottaa paremman automatisoinnin paljon lyhyemmässä ajassa. On selvää, että automatisoidun testin kehittäminen yhdessä testatun toiminnallisuuden kanssa on helpompaa, jos sama kehittäjä toteuttaa sekä toiminnallisuuden että sen automatisoidun testin. (Axelrod 2018: 70.)

Testien kirjoittaminen on nopeinta rinnakkain toiminnallisuuksien kehityksen kanssa. Tällöin kehittäjät tietävät tarkalleen, miten heidän kehittämänsä toiminnallisuuden tulisi

toimia ja siten myös, miten se voidaan testata parhaiten. Kehitystiimin on varattava aikaa testien kirjoittamiseen siten, että testausautomaatiota voidaan kehittää vähintään yhtä nopeasti kuin uusia toiminnallisuuksia. Muuten testien määrä kasvaa, mutta suhteellinen testikattavuus laskee jatkuvasti. Testien kirjoittamiseen kuluu merkittävästi vähemmän aikaa, jos testien kirjoittajat ovat testattavan toiminnallisuuden kehittäjiä.

Strategiana on kattaa tuotteen kaikki uudet toiminnallisuudet testeillä ja siirtää manuaalisen testauksen fokusta tuotteen vanhojen, automatisoimattomien ominaisuuksien testaamiseen sitä mukaa, kun luotto testausautomaation laatuun kasvaa. Tässä vaiheessa testausautomaatio alkaa maksaa itseään takaisin, sillä automatisoitu testaus on merkittävästi tehokkaampaa kuin manuaalinen testaus, ja virhetilojen korjaus on kehitysvaiheessa nopeampaa kuin missään muussa vaiheessa sen jälkeen. Yksikkötesteihin sijoitettu aika on murto-osa menneissä toiminnallisuuksissa havaittujen virheiden korjauksesta. Jos testausautomaation kehitys on nopeampaa kuin uusien toiminnallisuuksien lisäys, täysi kattavuus tullaan väistämättä saavuttamaan, minkä jälkeen manuaalista testausta voidaan käyttää paljon haastavampien ja enemmän luovuutta vaativien ongelmien testaukseen.

6 Tulokset ja yhteenveto

Testausautomaatio on lähes välttämätön ominaisuus ketterien menetelmien sovelluskehityksessä, jossa on jatkuvasti kasvava paine julkaista uusia ominaisuuksia. Ilman testausautomaatiota testaukseen tarvittavat resurssit voivat nopeasti moninkertaistua sovelluksen ominaisuuksien lisääntyessä. Hyvillä testausautomaation käytännöillä sovelluksen testauksen taakkaa voidaan jakaa kehittäjille, jolloin tulokseksi syntyy täsmällisempää ja tehokkaampaa testausta, jota voidaan kumuloida koko sovelluksen elinkaaren ajan. Hyvä testausautomaatio syntyy hyvässä suhteessa olevasta määrästä eriasteisia testejä, joiden ylläpitäminen ja kehittäminen on koko kehitystiimin vastuulla. Hyvä testausautomaatio vaatii kuitenkin kehitysaikaa ja kehityskulttuurin muutosta, eivätkä sen välittömät hyödyt eivät ole aina ilmeisiä, minkä vuoksi testiautomaatio onkin valmistelua vaativa, pitkän aikavälin sijoitus.

Testausautomaation implementointi sovelluksiin ei ole aina suoraviivaista vaan usein monitahoinen prosessi. Testausautomaation implementoinnissa on otettava huomioon sovelluksen kehittämiselle varattu aika, käytettävät resurssit sekä sovelluksen testattavuus. Aikataulun ollessa tiukka testausautomaatiossa joustetaan usein ensimmäisenä, sillä siitä ei saada välittömiä hyötyjä. Kun testaukseen kuluvien resurssien määrä ylittää tietyn kipukynnyksen, testausautomaatiosta saatetaan myös etsiä pelastusta, jolloin voi olla houkuttelevaa oikaista ja rikkoa hyviä käytäntöjä. Testausautomaation toimivuutta voidaan kuitenkin arvioida vasta kauan kehityksen alkamisen jälkeen, minkä vuoksi hyväksi todettujen, toimivien menetelmien seuraaminen alusta alkaen on tärkeää.

Opinnäytetyön aikana saatiin valmiiksi hyvä testipeti ja useampia esimerkkiyksikkötestejä. Näillä katettiin suurin osa mahdollisista käyttötapauksista, mikä tukee testauksen kehitystä tulevaisuudessa. Oopperan jäsenliittymiseen saatiin myös tehtyä useammalle asiakkaalle toiminnalliset testit, joita jatkokehitettiin toimimaan myös jatkuvan integraation palvelimella sekä manuaalista testausta avustavana työkaluna. Testauksen prosesseja ja menetelmiä tuodaan vähitellen kehitystiimin piiriin, mutta Oopperan ohjelmakoodissa on kuitenkin vielä osa-alueita, joita on kehitettävä testausystävällisemmäksi ennen kuin Oopperaan voidaan kehittää kattavaa testausautomaatiota.

Lähteet

Arnon Axelrod. 2018. Complete Guide to Test Automation Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects.

Bernard Homes. 2013. Fundamentals of Software Testing.

Futunion confluence-arkisto. aSuite dokumentaatio. Luettu heinäkuussa 2019.

Ham Vocke. 26.2.2018. Verkkoaineisto <https://martinfowler.com/articles/practical-test-pyramid.html>. Luettu heinäkuussa 2019.

Linda G. Hayes. 1.3.2004. Automated Testing Handbook.

Martin Fowler. 16.1.2018. Verkkoaineisto <https://martinfowler.com/bliki/IntegrationTest.html>. Luettu marraskuussa 2019.

Martin Fowler. 10.9.2013. Verkkoaineisto <https://martinfowler.com/bliki/PageObject.html>. Luettu heinäkuussa 2019.

Martin Fowler. 12.01.2011. Verkkoaineisto <https://martinfowler.com/bliki/ContractTest.html> Luettu marraskuussa 2019.

Mike Cohn. 30.10.2009. Succeeding With Agile: Software Development Using Scrum.

Paul Blundell & Diego Torres Milano. 2015. Learning Android Application Testing.