Reaktiivinen ohjelmointi web-sovellusten selainpuolella

Arto Chydenius

Helsinki 24.08.2014

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta Fakultet – Faculty			Laitos Institution Department
Matemaattis-luonnontieteellinen tiedekunta			Tietojenkäsittelytieteen laitos
Tekijä Författare Author			
Arto Chydenius			
Työn nimi Arbetets titel Title			
Reaktiivinen ohjelmointi web-sovellusten selainpuolella			
Oppiaine Läroämne Subject			
Tietojenkäsittelytiede			
Työn laji Arbetets art Level	Aika Datum Month and year		Sivumäärä Sidoantal Number of pages
Pro gradu -tutkielma	24.8.2014		68 sivua + 9 liitesivua

Tiivistelmä Referat Abstract

Reaktiivisen ohjelmoinnin on esitetty yksinkertaistavan sen avulla toteutettavia sovelluksia. Ohjelmointiparadigman on myös esitetty soveltuvan erityisen hyvin web-sovellusten selainpuolen ohjelmointiin. Tässä pro gradussa tutkitaan, mitkä ovat reaktiivisen ohjelmoinnin vaikutukset lähdekoodin kokoon ja monimutkaisuuteen. Tutkimusta varten on suunniteltu web-sovellus, josta on toteutettu kolme eri versiota: kaksi reaktiivista ja yksi ei-reaktiivinen versio. Versioiden lähdekoodia on vertailtu staattisen analyysin menetelmin sekä tutkimalla, miten jonkin toiminnallisuuden toteuttaminen tapahtuu jokaisen toteutusversion kohdalla.

ACM Computing Classification System (CCS):
Software and its engineering: Data flow languages
Software and its engineering: Language features

Avainsanat – Nyckelord Keywords

web-ohjelmointi, selainpuolen ohjelmointi, reaktiivinen ohjelmointi, ohjelmointiparadigmat, JavaScript

Säilytyspaikka Förvaringställe Where deposited

Muita tietoja Övriga uppgifter Additional information

Sisältö

1 Johdanto 1

2 Reaktiivinen ohjelmointi 3

2.1 Reaktiivinen ohjelmointi (RP) – eli reaktiivinen ohjelmointiparadigma 3

2.2 Funktionaalinen reaktiivinen ohjelmointi (FRP) 5

2.3 Reaktiiviset järjestelmät 6

3 Reaktiivisten ohjelmointikielten ominaisuudet 8

3.1 Riippuvuusverkko 8

3.2 Nostaminen (lifting) 9

3.3 Laskentamalli ja häiriöt (glitch) 12

4 Web-ohjelmointi 15

4.1 Verkkoselain ohjelmointiympäristönä 15

4.2 JavaScript 17

4.3 JavaScriptiksi käännettävät kielet 24

5 Reaktiivinen web-ohjelmointi 25

5.1 Flapjax 26

5.2 Reactive Extensions for JavaScript (RxJS) 28

5.3 Vaihtoehto reaktiiviselle ohjelmoinnille: lupaukset (promise) 29

6 Esimerkkisovellus: Flickr & Google Maps Mashup 30

6.1 Toiminta ja toiminnalliset vaatimukset 31

6.2 Arkkitehtuuri 35

6.3 Toiminnalliset testit 39

7 Sovelluksen analyysi ja tulokset 43

7.1 Mitattavat attribuutit 44

7.2 Muutosskenaariot 50

7.3 Staattisen lähdekoodianalyysin tulokset 51

7.4 Muutosskenaarioiden tulokset 59

7.5 Johtopäätökset 63

8 Yhteenveto 65

Lähteet **66**

Liite 1. Toiminnalliset testit **69**

Liite 2. Tiedostorakenne **71**

Liite 3. Rajapintakuvaukset **72**

Liite 4. Käyttöliittymäskenaarion lähdekoodi **74**

Liite 5. Palvelupyyntöskenaarion lähdekoodi **76**

1 Johdanto

Reaktiivinen ohjelmointi (Reactive Programming, RP) on muutaman viimeisen vuoden aikana suosiota saavuttanut ohjelmointiparadigma, joka soveltuu hyvin asynkronisten ja tapahtumapohjaisten (event-driven) sovellusten ohjelmointiin. Reaktiivisen ohjelmoinnin avulla ohjelmia pystytään ilmaisemaan deklaratiivisesti sen sijaan, että kaikki yksityiskohdat tarvitsisi ohjelmoida itse. Reaktiivisen ohjelmoinnin keskeisessä roolissa ovat tietovuot (data flow) ja tiedon automaattinen päivittyminen (propagation of change). Yksinkertainen esimerkki tästä on lauseke $c = a + b$. Jos lauseke toteutetaan jollain reaktiivisella ohjelmointikielellä, niin muuttujien a tai b arvojen muuttaminen aiheuttaa aina muuttujan c arvon automaattisen uudelleenlaskennan. Tätä automaattisesti etenevää päivittymistä voidaan ajatella tietovoina.

Yhdeksi reaktiivisen ohjelmoinnin suosituista sovelluskohteista on vakiintunut verkkopalvelut. Tämä johtuu toisaalta perinteisten, suoraan käyttöjärjestelmän päälle toteutettujen natiivien ohjelmistojen siirtymisestä verkkoon erilaisiksi pilvipalveluiksi [Tai11] ja toisaalta siitä, että modernit verkkopalvelut ovat luonteeltaan erittäin asynkronisia ja tapahtumapohjaisia [KBD13]. Verkkopalvelujen asynkroninen ja tapahtumapohjainen luonne johtuu siitä, että verkkoselaimessa suoritettavat palvelut ovat usein yhteydessä erilaisiin taustajärjestelmiin, joiden välillä kommunikointi tapahtuu asynkronisten palvelupyyntöjen avulla. Lisäksi suuri osa verkkopalveluiden käyttöliittymäkerroksen toiminnallisuudesta on erilaisiin käyttäjän syötteisiin reagointia. Monimutkaisten ketjutettujen asynkronisten kutsujen käsittely hajauttaa ohjelman suoritusta eri puolille koodia, tehden lähdekoodin seuraamisesta vaikeaa. Tätä kutsutaan toisinaan ”asynkroniseksi spagetiksi” [MiT07]. Reaktiivisen ohjelmoinnin on osoitettu tarjoavan tähän ratkaisun [KBD13].

Esimerkkejä reaktiivista ohjelmointia hyödyntävistä suuren mittakaavan verkkopalveluista ovat suositut Netflix-videopalvelu [Net14a] sekä GitHub hosting -palvelu [Git14]. Molemmat palvelut ovat ottaneet reaktiivisen ohjelmoinnin käyttöönsä ja saavuttaneet sillä etuja suorituskyvyn, skaalautuvuuden ja arkkitehtuurin selkeyden suhteen [Mic14a]. Netflix ja GitHub hyödyntävät reaktiivista ohjelmointia sekä asiakassovellusten käyttöliittymäkerroksessa että asiakassovellusten taustajärjestelmäintegraatioissa. Netflix hyödyntää reaktiivista ohjelmointia myös taustajärjestelmissään.

Reaktiivisesta ohjelmoinnista on tehnyt ajankohtaista myös funktionaalisen

reaktiivisen ohjelmoinnin (Functional Reactive Programming, FRP) [ElH97] viimeaikainen aktiivinen tutkimus. Funktionaalinen reaktiivinen ohjelmointi yhdistää sekä reaktiivisen ohjelmoinnin että funktionaalisen ohjelmoinnin. Reaktiivinen ohjelmointi ja FRP ovat läheisiä käsitteitä, mutta niiden erot eivät ole eksaktisti määriteltyjä, ja termejä saatetaankin käyttää epäjohdonmukaisesti.

Tässä pro gradussa tutkitaan reaktiivisen ohjelmoinnin vaikutusta lähdekoodin kokoon ja monimutkaisuuteen. Tutkimus on suoritettu analysoimalla tutkielmaa varten suunnitellun web-sovelluksen lähdekoodia staattisen analyysin menetelmin. Web-sovelluksesta on toteutettu kolme versiota, joista kaksi on reaktiivisia ja yksi ei-reaktiivinen. Versiot ovat toiminnallisuudeltaan identtiset. Lisäksi työssä on tutkittu, miten sovelluksen toiminnallisuuteen kohdistuvien muutosten tekeminen tapahtuu jokaisen toteutusversion kohdalla. Tämän avulla on pyritty saamaan esiin reaktiivisen ohjelmoinnin vaikutusta ylläpidettävyyteen sekä saamaan tietoa siitä, miten tutkimustulokset yleistyvät suurempiin ohjelmistoihin.

Tutkielman luvussa 2 määritellään terminologiaa niin, että tekstissä käytettävät termit ovat selkeästi rajattu. Luvussa 3 esitellään reaktiivisille ohjelmointikielille ja kirjastoille yhteisiä käsitteitä ja ominaisuuksia. Luvussa 4 käydään läpi yleisesti web-ohjelmointia etenkin selainkerroksen osalta. Luvussa 5 käsitellään reaktiivisen ohjelmoinnin soveltamista web-ohjelmointiin ja esitellään myös kahta reaktiiviseen web-ohjelmointiin tarkoitettua tekniikkaa: Flapjax-ohjelmointikieltä ja RxJS-kirjastoa. Luvussa 6 esitellään tutkielmaa varten toteutetun web-sovelluksen toimintaa ja arkkitehtuuria. Luvussa 7 määritellään tutkielmassa käytetyt tutkimusmenetelmät, suoritetaan esimerkkisovelluksen analyysi ja esitetään analyysista saadut tulokset. Luvussa 8 tehdään lopuksi yhteenveto tutkielmasta.

2 Reaktiivinen ohjelmointi

Tällä hetkellä termiä reaktiivinen ohjelmointi käytetään erittäin laajasti: sillä voidaan tarkoittaa erilaisten järjestelmien toteuttamista reaktiivisesti [The13] tai johdantoluvussa esiteltyä tietovoihin ja automaattiseen päivittymiseen pohjautuvaa ohjelmointiparadigmaa. Jälkimmäisen tapauksessa myös termejä reaktiivinen ohjelmointi ja funktionaalinen reaktiivinen ohjelmointi käytetään monesti tarkoittamaan samaa asiaa. Esimerkkinä tästä on reaktiivisten ohjelmointikielten ominaisuuksia kartoittava ja vertaileva tutkimus ”A Survey on Reactive Programming” [Bai12], jossa Bainomugisha et. al käyttävät termiä reaktiivinen ohjelmointi, vaikka suurin osa tutkimuksessa käsitellyistä ohjelmointikielistä luokittelee itsensä nimenomaan funktionaalisiksi reaktiivisiksi ohjelmointikieliksi.

Toisaalta myös reaktiivista ohjelmointia kutsutaan toisinaan funktionaaliseksi reaktiiviseksi ohjelmoinniksi. Reactive Extensions -sovelluskehyksen (Rx) dokumentaatiossa mainitaan, että Rx:n lähestymistapaa kutsutaan funktionaaliseksi reaktiiviseksi ohjelmoinniksi, mutta kyseessä ei ole kuitenkaan Franin [ElH97] tapaisten kielten funktionaalisen reaktiivisen ohjelmointiparadigman puhdas toteutus [Net14b].

Epäselvyys terminologiassa johtuu todennäköisesti siitä, että toisin kuin funktionaalinen reaktiivinen ohjelmointi, joka on syntynyt akateemisen tutkimuksen tuotteena tarkkoine malleineen, reaktiivinen ohjelmointi on otettu käyttöön geneerisenä terminä pikku hiljaa kuvaamaan kaikenlaisia tapahtumiin reagoivia ohjelmointitapoja ja ohjelmistoratkaisuja.

Tämän luvun aliluvuissa käydään läpi reaktiivinen ohjelmointi -termillä tarkoitettua kolmea erillistä käsitettä: reaktiivista ohjelmointia – eli reaktiivista ohjelmointiparadigmaa, funktionaalista reaktiivista ohjelmointia ja reaktiivisten järjestelmien ohjelmointia. Tässä tekstissä termillä reaktiivinen ohjelmointi tarkoitetaan jatkossa nimenomaan luvun 2.1 mukaista ohjelmointiparadigmaa.

2.1 Reaktiivinen ohjelmointi (RP) – eli reaktiivinen ohjelmointiparadigma

Reaktiivisesta ohjelmoinnista puhuttaessa tarkoitetaan useimmiten automaattisesti päivittyvistä arvoista ja näiden muodostamista tietovoista koostuvaa ohjelmointiparadigmaa, jonka soveltamiseen käytetään reaktiivista ohjelmointikieltä tai jollekin

ohjelmointikielelle toteutettua reaktiivisen ohjelmoinnin mahdollistavaa kirjastoa. Näin on myös tämän tutkielman tapauksessa. Arvojen väliset riippuvuudet ja niiden automaattinen päivittyminen ovat koko reaktiivisen ohjelmoinnin perusta.

Suoritettaessa jollain ei-reaktiivisella ohjelmointikielellä toteutettua lauseketta $c = a + b$, sijoitus muuttujaan c tapahtuu ainoastaan kerran, jonka jälkeen muuttujan arvo pysyy aina samana, mikäli sitä ei erikseen muuteta. Reaktiivisella ohjelmointikielellä toteutetussa versiossa taas a:n tai b:n muuttaminen aiheuttaa c:n automaattisen uudelleenlaskennan, koska sen arvo on riippuvainen muuttujista a ja b. Jos edellisen lausekkeen jälkeen kirjoitettaisiin uusi lauseke $d = c + 1$, määrittelisi se d:n riippuvaiseksi c:stä. Tällöin a:n tai b:n päivittäminen johtaisi c:n automaattiseen päivittymiseen, joka puolestaan johtaisi d:n automaattiseen päivittymiseen. Tällaista muuttujien välisten riippuvuuksien muodostamaa verkkoa ja verkon päivittämistä kutsutaan tietovoiksi. Tietovuomallia voidaan hyödyntää esimerkiksi käsittelemällä käyttöliittymästä tulevia syötteitä tällaisina voina. Tietovoiden etuna on myös tapahtumien tehokas abstraktio: reagointia vaativat tapahtumat voivat olla yhtä hyvin peräisin sovelluksen paikallisesta käyttöliittymästä tai fyysisesti muualla sijaitsevasta taustajärjestelmästä [Mey09].

Nykyisten reaktiivisten ohjelmointikielten voidaan katsoa pohjautuvan 1980- ja 1990-lukujen tietovuo-ohjelmointikieliin, kuten Estereliin [BeG92] ja toisaalta 1990- ja 2000-lukujen funktionaalisen reaktiivisen ohjelmoinnin tutkimukseen. Käytännössä nykyisen reaktiivisen ja funktionaalisen reaktiivisen ohjelmoinnin määritteli Conal Elliotin ja Paul Hudakin tutkimus Functional Reactive Animation [ElH97]. Tutkimus esitteli Haskell-ohjelmointikielen laajennokseksi toteutetun Fran-ohjelmointikielen, joka oli tarkoitettu animaatioiden deklaratiiviseen ohjelmoimiseen.

Fran mahdollistaa reaktiivisen ohjelmoinnin erityisillä reaktiivisilla tietotyypeillä, jotka ovat samalla tavalla ensimmäisen luokan kansalaisia (first-class citizens) kuin alkeistyypit tai oliot olio-ohjelmoinnissa. Käytännössä kaikki nykyiset reaktiiviset ohjelmointikielet noudattavat Franin mallia toteuttamalla erillisiä reaktiivisia tietotyyppejä. Franin jälkeen on julkaistu useita reaktiivisia ohjelmointikieliä, jotka pohjautuvat useisiin eri isäntäkieliin. Franin lisäksi esimerkiksi Yampa [Has14] on Haskell-ohjelmointikielen laajennos. Flapjax [Mey09] on suunniteltu selain-ympäristöön, ja se on JavaScript-kielen laajennos. Flapjaxia käsitellään tarkemmin tämän tutkielman luvussa 5.1.

Fran määrittelee reaktiiviset tietotyypit osana ohjelmointikieltä, mutta reaktiivista ohjelmointia on mahdollista toteuttaa myös laajentamalla ei-reaktiivisia

ohjelmointikieliä erilaisilla kirjastoilla. Tällaisia kirjastoja ovat esimerkiksi Scala.React Scala-ohjelmointikielelle [MaO12] ja Reactive Extensions -sovelluskehyksen [Mic14b] versiot, joita löytyy muun muassa Javalle, JavaScriptille ja C#:lle. Reactive Extensions -sovelluskehyksen JavaScript-versiota RxJS:aa käsitellään tarkemmin tämän tutkielman luvussa 5.2. Flapjaxia on myös mahdollista hyödyntää itsenäisen ohjelmointikielen sijasta JavaScript-kielisenä kirjastona.

Tässä tekstissä reaktiivinen ohjelmointi on määritelty samaan tapaan kuin Reactive Extensions -sovelluskehyksessä: FRP luetaan osaksi reaktiivista ohjelmointia, mutta reaktiiviselta ohjelmoinnilta ei edellytetä kaikkia FRP:n ominaisuuksia [Net14b]. Aiemmin mainitun Bainomugishan et al. tutkimuksen mukainen luokittelu [Bai12] vaatii reaktiiviselta ohjelmointikieleltä tai kirjastolta käytännössä FRP:n ominaisuudet. Toisin kuin Bainomugishan et al. tutkimuksessa, niin tässä tekstissä esimerkiksi Reactive Extensions -sovelluskehys lasketaan täysiveriseksi reaktiivisen ohjelmoinnin mahdollistavaksi kirjastoksi, koska se perustuu muuttujien automaattiseen päivittymiseen tietovoiden avulla. Tämän tekstin mukaiselle luokittelulle on perusteena myös se, että Rx-sovelluskehys on ollut yksi suurimmista ”reaktiivinen ohjelmointi” -nimellä kulkevan ohjelmointiparadigman suosion edistäjistä.

2.2 Funktionaalinen reaktiivinen ohjelmointi (FRP)

Funktionaalisen reaktiivisen ohjelmoinnin tutkimus on ollut hyvin pitkälti perustana nykyiselle reaktiiviselle ohjelmoinnille. Toisin kuin geneerisempi reaktiivinen ohjelmointi -termi, FRP on tarkasti määritelty akateemisessa tutkimuksessa Elliotin ja Hudakin ensimmäisestä julkaisusta [ElH97] lähtien. Julkaisussa esitelty Fran-ohjelmointikieli määrittelee kaksi reaktiivista tietotyyppiä: reaktiiviset arvot (behavior) ja tapahtumavuot (event). Reaktiiviset arvot ovat jatkuvia ja aikasidonnaisia. Niillä on voimassa aina jokin arvo, joka vaihtelee ajanhetkestä riippuen. Tapahtumavuot puolestaan kuvaavat diskreettien tapahtumien voita, joiden arvot ovat olemassa vain tiettyinä ajanhetkinä. FRP-kielissä reaktiivisten arvojen ja tapahtumavoiden käsittely tapahtuu pitkälti funktionaalisen ohjelmoinnin menetelmin, josta johtuen isäntäkieleltä edellytetään aina joitain funktionaalisen ohjelmoinnin työkaluja.

Yksi esimerkki reaktiivisesta arvosta on aika, ja esimerkkejä tapahtumavoista ovat hiireltä tai näppäimistöltä tulevat syötteiden virrat. Koska reaktiiviset arvot ja tapahtumavuot ovat ensimmäisen luokan kansalaisia, niitä on mahdollista yhdistellä

ja käsitellä luontevasti: esimerkiksi reaktiivisille arvoille voidaan tehdä aritmeettisia operaatioita ja tapahtumavoita pystytään yhdistelemään ja suodattamaan.

Reaktiiviset arvot ja tapahtumavuot löytyvät edelleen lähes kaikista RP- ja FRP-kielistä, tosin nimeämiskäytännöissä on joitain eroja kielten välillä: esimerkiksi bacon.js-kirjasto [Paa14] käyttää reaktiivisista arvoista nimeä property ja Scala.React -kirjasto nimeä signal. Yksi keskeisistä Bainomugishan et al. katsauksen luokittelukriteereistä on tuki reaktiivisille arvoille ja tapahtumavoille. Luokittelu jättää Reactive Extensions -sovelluskehyksen keskeisimpien RP-kielten ja kirjastojen ulkopuolelle nimenomaan siitä syystä, että se toteuttaa ainoastaan diskreetit tapahtumavuot reaktiivisten arvojen puuttuessa.

Reaktiivisten arvojen ja tapahtumavoiden lisäksi Franissa on esitelty myös nostaminen (lifting), joka on reaktiivisessa ohjelmoinnissa yleinen käsite. Nostaminen määrittelee, miten reaktiivisia tietotyyppejä käsitellään isäntäkielen operaattoreilla ja funktioilla. Nostamistapoja on erilaisia, ja nämä vaikuttavat kielen syntaksiin ja käyttötapaan. Nostamista käsitellään tarkemmin luvussa 3.2.

2.3 Reaktiiviset järjestelmät

Reaktiivisten järjestelmien toteuttamisesta käytetään toisinaan termiä reaktiivinen ohjelmointi. Tällaisten järjestelmien piirteistä ja vaatimuksista hyvänä kuvauksena toimii The Reactive Manifesto [The13]. Manifestin takana on nimekkäitä ja tunnustettuja alan asiantuntijoita, kuten asynkronisen Akka-kirjaston [Typ14] kehittäjä Jonas Boner, Reactive Extensions -sovelluskehyksen [Mic14b] kehittäjä Erik Meijer sekä Scala-ohjelmointikielen [Eco14] ja Scala.React-kirjaston kehittäjä Martin Odersky.

Manifesti määrittelee reaktiiviselle järjestelmälle neljä vaatimusta: tapahtumapohjaisuus (event-driven), skaalautuvuus (scalability), sitkeys (resiliency) ja responsiivisuus (responsivity). Tapahtumapohjaisuudella tarkoitetaan järjestelmän pohjautumista asynkroniseen ja ei-blokkaavaan kommunikaatioon, joka mahdollistaa paremman suorituskyvyn varsinkin järjestelmän ollessa kuormitettuna. Skaalautuvuudella tarkoitetaan etenkin horisontaalista skaalautuvutta, eli järjestelmän suorituksen hajauttamista useammalle ytimelle, prosessorille tai erilliselle palvelimelle. Sitkeydellä tarkoitetaan järjestelmän kykyä palautua virhetilanteista nopeasti. Virheidenkäsittelyn tulee tapahtua useilla eri tasoilla, jolloin virhetilanne saadaan eristettyä tehokkaasti ja siten vaikuttamaan ainoastaan

järjestelmän yhteen osaan. Responsiivisuudella tarkoitetaan järjestelmän kykyä palvella käyttäjiä riippumatta sen kuormasta. Vaikka reaktiivisessa manifestissa ei sinänsä ole uusia keksintöjä, manifesti kokoaa moderneilta verkkopalveluilta vaaditut ominaisuudet yhteen hyväksi kehykseksi. Esimerkiksi Netflixin taustajärjestelmä voidaan monilta osin lukea reaktiivisen manifestin mukaiseksi [ChH13].

Reaktiivinen ohjelmointi -termin käyttäminen järjestelmän ominaisuuksista puhuttaessa sekoittaa jo muutenkin hieman epäselvästi rajattua RP ja FRP-käsitteistöä. Toisaalta yhteistä reaktiivisten järjestelmien sekä RP ja FRP-ohjelmointiparadigmojen välillä on se, että jälkimmäiset soveltuvat hyvin yhdeksi reaktiivisten järjestelmien toteuttamisessa käytettäviksi työkaluksi.

3 Reaktiivisten ohjelmointikielten ominaisuudet

Tässä tekstissä termillä reaktiivinen ohjelmointikieli voidaan tarkoittaa joko reaktiivisen ohjelmointiparadigman mukaista ohjelmointikieltä tai sen mahdollistavaa kirjastoa, mikäli ei ole erityistä syytä tehdä selväksi kummasta on kyse. Reaktiivisen ohjelmoinnin mahdollistavat kielet ja kirjastot sisältävät paljon yhteisiä piirteitä ja ominaisuuksia. Keskeisimpiä näistä ominaisuuksista ovat tietovuot muodostava riippuvuusverkko, nostaminen, jolla mahdollistetaan reaktiivisten muuttujien käsittely ja rekisteröiminen riippuvuusverkkoon, sekä laskentamalli ja häiriöt, jotka ovat oleellisessa roolissa toteutettaessa reaktiivisia ohjelmointikieliä. Näitä ominaisuuksia käsitellään tämän luvun aliluvuissa.

3.1 Riippuvuusverkko

Kaikki reaktiivisten ohjelmointikielten toteutukset perustuvat pohjimmiltaan riippuvuusverkkoon (dependency graph), joka määrittelee reaktiivisten muuttujien väliset riippuvuudet. Riippuvuusverkko on suunnattu verkko (directed graph), joka sisältää operaatioita ja muuttujia. Joissain reaktiivisissa kielissä verkko on myös syklitön (acyclic).

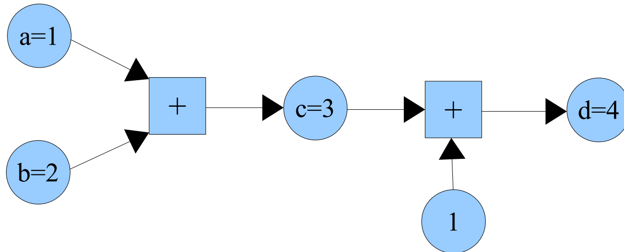
Reaktiivisten muuttujien oikeasta tilasta huolehditaan läpikäymällä verkkoa ja päivittämällä riippuvia arvoja. Alla esitetty, reaktiivisia arvoja käsittelevistä lausekkeista koostuva koodi muodostaa kuvan 3.1 mukaisen riippuvuusverkon.

```
a = 1
b = 2
c = a + b
d = c + 1
```

Kuvan 3.1 esittämässä verkossa muuttujista *a* ja *b* on viittaus yhteenlaskuoperaation kautta muuttujaan *c*. Muuttujasta *c* ja vakiosta *1* on viittaus yhteenlaskuoperaation kautta muuttujaan *d*. Nämä riippuvuudet aiheuttavat sen, että muuttujien *a* tai *b* päivittäminen laukaisee aina muuttujan *c* automaattisen päivittymisen, joka puolestaan laukaisee muuttujan *d* päivittymisen.

Riippuvuusverkko on normaalisti käyttäjälle näkymätön, jolloin ohjelmoija pystyy keskittymään vain reaktiivisten muuttujien käyttämiseen. Verkosta on hyvä olla kuitenkin tietoinen, koska syklien muodostaminen verkkoon saattaa aiheuttaa ongelmia riippuen kielen toteutuksesta. Joissain tapauksissa syklit saattavat aiheuttaa esimerkiksi riippuvuusverkon loppumattoman läpikäynnin. Yleisesti syklejä

on syytä välttää kokonaan, mutta esimerkiksi Flapjaxissa vaihtoehtona on sisällyttää yhteen syklin solmuista viivefunktio [Mey09].



Kuva 3.1: Esimerkki riippuvuusverkosta

3.2 Nostaminen (lifting)

Nostaminen määrittelee, miten reaktiivisia tietotyyppejä käsitellään isäntäkielen operaattoreilla ja funktioilla. Muuttujien rekisteröiminen riippuvuusverkkoon tapahtuu myös nostamisen yhteydessä. Bainomugishan et al. mukaan nostaminen voi tapahtua kolmella eri tavalla: implisiittisesti, eksplisiittisesti tai manuaalisesti [Bai12]. Nämä nostamistavat esitellään seuraavaksi.

Implisiittinen nostaminen

Implisiittinen nostaminen tapahtuu ohjelmoijan näkökulmasta täysin automaattisesti. Tällöin ohjelmoija voi käsitellä reaktiivisia tietotyyppejä aivan kuten kielen muitakin ensimmäisen luokan tyyppejä. Koska reaktiivisuutta ei tarvitse huomioida erityisellä syntaksilla tai ohjelmointikäytännöillä, voidaan puhua läpinäkyvästä reaktiivisuudesta (transparent reactivity) [Bai13]. Esimerkiksi Flapjax-ohjelmointikielessä nostaminen tapahtuu implisiittisesti.

```
// Asetetaan x:n arvoksi sekunnin välein päivittyvä reaktiivinen arvo,
// jonka arvo on aina päivitysajankohta millisekunteina
var x = timerB(1000);

// Lisätään reaktiiviseen arvoon yksi millisekunti:
// reaktiivista arvoa voidaan käyttää kuten muitakin tyyppejä.
// y:n arvoksi tulee uusi reaktiivinen arvo, joka riippuu x:n arvosta
// ja päivittyy aina automaattisesti x:n päivittyessä
var y = x + 1;
```

Yllä oleva lähdekoodi on toteutettu implisiittisen nostamisen omaavalla Flapjax-

ohjelmointikielellä. Koodissa *timerB*-funktiolla luotu, sekunnin välein automaattisesti päivittyvä reaktiivinen arvo sijoitetaan muuttujaan *x*. Seuraavaksi muuttujaan *y* asetetaan arvo, joka saadaan, kun muuttujan *x* arvoon lisätään yksi millisekunti. Tämän jälkeen muuttuja *y* on myös reaktiivinen arvo, joka on aina *x*:n arvo lisättynä yhdellä millisekunnilla. Nostaminen tapahtuu jälkimmäisessä lausekkeessa: reaktiivista arvoa ja numeerista alkeistyyppiä käsitellään yhteenlaskuoperaattorin avulla, jonka lopputuloksena saadaan uusi reaktiivinen arvo. Flapjaxia käytettäessä on tapana käyttää jälkiliitettä B viitattaessa reaktiivisiin muuttujiin ja jälkiliitettä E viitattaessa tapahtumavoihin. Yllä olevan esimerkin tapauksessa *timerB*-funktion nimi siis kertoo paluuarvon olevan reaktiivinen muuttuja.

Koska nostaminen tapahtuu implisiittisesti, muuttujaa *x* voidaan käsitellä suoraan kuten mitä tahansa JavaScriptin tyyppiä olevaa muuttujaa. Asetettaessa muuttujaa *y* sen arvo päivittyy automaattisesti osaksi riippuvuusverkkoa ja on riippuvainen muuttujan *x* arvosta. Flapjaxin tapaan useat dynaamisen tyypityksen omaavat kielet käyttävät implisiittistä nostamista.

Eksplisiittinen nostaminen

Staattisen tyypityksen omaavat kielet käyttävät usein eksplisiittistä tai manuaalista nostamistapaa. Eksplisiittisesti nostettaessa ei saavuteta implisiittisen nostamisen tavoin reaktiivista läpinäkyvyyttä, vaan reaktiivisten arvojen ja kielen muita tyyppejä olevien arvojen yhdisteleminen vaatii ohjelmoijalta erillistä nostamisen määrittelemistä. Flapjax-ohjelmointikielen lisäksi Flapjax on mahdollista ottaa käyttöön myös JavaScript-kielisenä kirjastona, jota käytettäessä nostaminen täytyy tehdä eksplisiittisesti.

```
// Asetetaan x:n arvoksi sekunnin välein päivittyvä reaktiivinen arvo,
// jonka arvo on aina päivitysajankohta millisekunteina
var x = timerB(1000);

// Asetetaan y:n arvoksi eksplisiittisesti nostettu x,
// jonka arvoon lisätään aina yksi millisekunti.
// Funktion sisällä t on numeerinen, jolloin sitä voidaan käsitellä
// JavaScriptin tavallisilla operaatioilla
var y = liftB(function(t) {
      return t + 1;
}, x);
```

Yllä oleva lähdekoodi on toteutettu Flapjax-kirjastolla. Koodi vastaa toiminnallisuudeltaan täysin aiempaa Flapjax-kielistä esimerkkiä. Käytettäessä kirjastoa yhteenlaskuoperaattorin käyttäminen ei onnistu suoraan reaktiivisen arvon *x* ja

numeerisen vakion välillä, vaan se täytyy tehdä nostamalla. Nostaminen tapahtuu Flapjax-kirjaston *liftB*-funktion avulla, joka luo uuden reaktiivisen arvon, jolla on riippuvuus muihin reaktiivisiin arvoihin. *liftB*-funktiolla luotavalla uudella reaktiivisella arvolla on riippuvuus muuttujaan *x*. *liftB*-funktiolle annetaan ensimmäiseksi parametriksi nimetön funktio, jolla käsitellään jälkimmäisenä parametrina annettua reaktiivista arvoa *x*. Nimetön funktio saa *x*:n arvon automaattisesti parametrissa *t*. Koska *t* on numeerinen, sen arvoa voidaan kasvattaa yhdellä millisekunnilla käyttäen JavaScriptin tavallista yhteenlaskuoperaattoria. Kasvatettu arvo palautetaan, koska *liftB* odottaa ensimmäisen parametrin palauttavan arvon, joka sitten asetetaan uuden reaktiivisen arvon arvoksi.

Koodissa *y*:llä on riippuvuus ainoastaan reaktiiviseen muuttujaan *x*, mutta riippuvuuksia voidaan antaa *liftB*-funktion parametreina useampiakin. Eksplisiittinen nostaminen lisää muuttujien väliset riippuvuudet aina riippuvuusverkkoon.

Manuaalinen nostaminen

Manuaalinen nostaminen on käytössä kielissä, joissa ei ole reaktiivisia arvoja ensimmäisen luokan kansalaisina. Manuaalisessa nostamisessa muuttujien välinen riippuvuus on olemassa, mutta voimassa oleva arvo täytyy pyytää erikseen tai siitä voidaan ilmoittaa erikseen. Manuaalisen nostamisen toteutustapa saattaa vaihdella kielestä riippuen melko paljonkin. Esimerkiksi Reactive Extensions ilmoittaa uudelleen lasketuista arvoista aina takaisinkutsuina.

```
// Luodaan tapahtumavuo, joka lähettää tapahtumia sekunnin välein
var source = Rx.Observable.interval(1000);

// Luodaan edellisestä uusi tapahtumavuo, jonka yksittäisten
// tapahtumien arvo on suoritushetki lisättynä yhdellä millisekunnilla
var added = source.map(function(x) {
      // Otetaan suoritushetki millisekunteina
      var time = (new Date()).getTime();

      // Lisätään arvoon yksi millisekunti
      return time + 1;
});

// Tilataan tapahtumavuo ja käsitellään sitä takaisinkutsun avulla
var subscription = added.subscribe(function(t) {
      // Tämä funktio suoritetaan sekunnin välein ja t:n arvo
      // vastaa aiemmin määritellyn map-metodin palauttamaa
      // t:tä voidaan käsitellä tässä takaisinkutsussa vastaavasti
      // kuin aiemmissa esimerkeissä
});
```

Yllä oleva lähdekoodi on toteutettu RxJS-kirjastolla, ja se simuloi jossain määrin kahta aiempaa koodiesimerkkiä. Täysin vastaavaa esimerkkiä RxJS:llä ei ole mahdollista toteuttaa, koska kirjasto ei tue reaktiivisia arvoja vaan reaktiivisuus toteutetaan RxJS:lla aina tarkkailemalla muutoksia tapahtumavoissa. Koodissa luodaan ensin tapahtumavuo *source*, joka lähettää yksittäisiä tapahtumia sekunnin välein. Tämän jälkeen muuttujaan *added* asetetaan edellisestä luotu tapahtumavuo, jossa otetaan *Date*–olion avulla tämänhetkinen aika ja lisätään siihen yksi millisekunti. Koodissa joudutaan ottamaan suoritusaika erikseen, koska RxJS:n *interval*-metodilla luotu tapahtumavuo ei sisällä suoritusaikaa, toisin kuin Flapjaxin *timerB*-funktiota käytettäessä. Uudessa tapahtumavuossa ilmeneville tapahtumille rekisteröidään tarkkailija *subscribe*-metodilla. Tapahtumien käsittelijäksi määritellään nimetön funktio, joka saa automaattisesti arvon ajasta muuttujassa *t*. Muuttujaa *t* voidaan käsitellä nimettömän funktion sisällä vastaavasti kuin eksplisiittisesti nostetussa esimerkissä.

Koodissa nostaminen tapahtuu *map*-metodin avulla, joka päivittää myös riippuvuusverkkoa. Toimintatapa vastaa eksplisiittistä nostamista, mutta lopputulokseksi saadun reaktiivisen muuttujan käsittely tapahtuu eri tavalla, koska Reactive Extensions mahdollistaa reaktiivisuuden ainoastaan tietovoiden kuuntelemisen kautta sen sijaan, että arvoa voitaisiin kysyä suoraan.

3.3 Laskentamalli ja häiriöt (glitch)

Riippuvuusgraafin läpikäymiseen on kaksi vaihtoehtoa: vetäminen (pull) ja työntäminen (push). Pull-mallissa tarvittava arvo ”vedetään” riippuvuuden kohteesta. Aiemmin esitetyn kuvan 3.1 tapauksessa tarvittaessa muuttujan *d* arvoa, vedettäisiin ensin arvo *c*, jonka jälkeen vedettäisiin arvot *b* ja *a*. Ensimmäiset FRP-kielet, kuten Fran, käyttivät verkon läpikäymiseen pull-mallia. Pull-malli on push-mallia suoraviivaisempi toteuttaa, ja sen etuna on, että arvoja lasketaan ja päivitetään ainoastaan silloin, kun se on ehdottomasti pakollista. Pull-mallissa suurimpana ongelmana on mahdollinen viive tapahtuman ja vastauksen välillä, mikäli vastauksen muodostaminen vaatii pitkän ketjun riippuvuusgraafin solmujen läpikäyntejä. Tästä saattaa olla seurauksena tila–aika -vuotoja (space-time leaks), jolloin reaktiiviset muuttujat saattavat saada vanhentuneita arvoja.

Push-mallissa arvoja päivitetään verkkoon aina, kun uutta tietoa valmistuu. Jatkuvalla päivittymisellä vältetään tila–aika -vuodot, koska kaikkea ei lasketa kerralla. Varjopuolena on, että Push-malli vaatii tehokasta toteutusta turhan

laskennan välttämiseksi. Sittemmin push-malli on vakiintunut reaktiivisten kielten yleisimmäksi laskentamalliksi, ja esimerkiksi Scala.React ja Flapjax käyttävät tätä mallia.

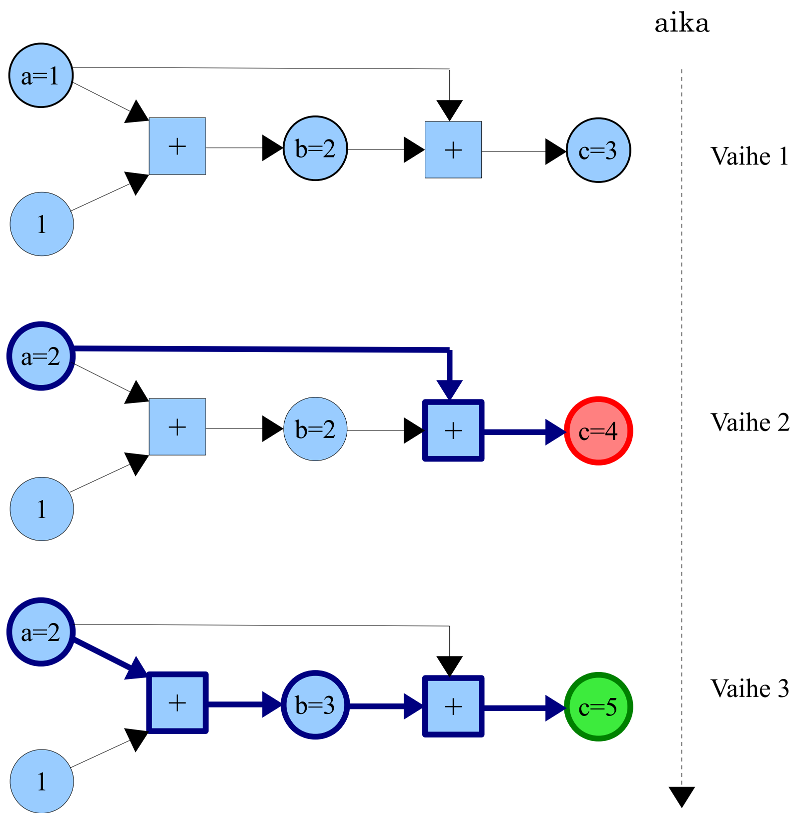
Eräs keskeisimmistä ongelmista reaktiivisten ohjelmointikielien ja kirjastojen toteuttamisessa on myös häiriöiden (glitch) estäminen. Häiriö tapahtuu, kun reaktiivisen muuttujan arvoa ollaan laskemassa, eivätkä sen kaikki riippuvuudet ole vielä päivittyneet riippuvuusverkossa. Tällöin lopputulokseksi saadaan virheellinen arvo, koska laskenta perustuu osittain vanhaan tietoon. Häiriöt ovat ainoastaan push-mallin ongelma.

```
a = 1
b = a + 1
c = a + b
```

Pseudokoodina toteutetussa yllä olevassa esimerkissä muuttujalla c on suora riippuvuus muuttujaan a. Lisäksi muuttujalla c on riippuvuus muuttujaan a myös muuttujan b kautta. Tällöin verkon läpikäyntijärjestys vaikuttaa muuttujan c arvoon.

Kuvassa 3.2 on havainnollistettu verkon läpikäyntiä. Kaaviossa verkon solmujen päivittyminen on esitetty kolmessa vaiheessa ylhäältä alas. Ensimmäisessä vaiheessa verkko on alkuperäisessä, yllä olevan koodin mukaisessa tilassa. Toisessa vaiheessa muuttujan a arvoksi asetetaan *2*, ja verkkoa lähdetään päivittämään työntämällä päivitetty arvo lausekkeen $c = a + b$ läpi. Työntäminen on esitetty kaaviossa paksulla sinisellä viivalla. Lausekkeen arvo lasketaan b:n vanhalla arvolla, josta johtuen lopputulokseksi saatu c:n arvo on väärä. Kolmannessa vaiheessa a:n arvo työnnetään lausekkeen $b = a + 1$ läpi, jolloin b päivittyy oikeaksi arvoksi *3*. Tämän jälkeen työntämistä jatketaan lausekkeeseen $c = a + b$, jolloin c:n arvo on jälleen oikea, eli 5. Mikäli verkkoa olisi lähdetty käymään läpi vaiheesta 3, häiriötä ei olisi aiheutunut.

Useimmat reaktiiviset kielet ratkaisevat tämän ongelman järjestämällä verkon riippuvuuksien mukaan eli topologisesti. Läpikäytäessä topologisesti järjestettyä verkkoa riippuvuudet lasketaan aina ennen käsiteltävää solmua, jolloin häiriöitä ei synny.



Kuva 3.2: Esimerkki häiriön tapahtumisesta

4 Web-ohjelmointi

Verkkosivut ovat viimeisen kymmenen vuoden aikana kehittyneet staattisista dokumenteista natiivien sovellusten kilpailijoiksi [Tai11]. Tämä puolestaan asettaa kasvavia vaatimuksia ohjelmointiympäristölle pyrittäessä hallitsemaan monimutkaistuvia palveluita.

Web-ohjelmointi on usein jaettu selainpuolen (frontend) ja palvelinpuolen (backend) vastuualueisiin. Verkkosivujen alkuperäinen toimintatapa on ollut muodostaa palvelinpuolella valmiita dokumentteja vastauksiksi verkkoselaimen lähettämiin palvelupyyntöihin. Tässä toimintatavassa selainpuolen vastuulle on jäänyt lähinnä sivujen sisällön esittäminen HTML-merkkauskielen ja CSS-tyylitiedostojen avulla. Sisällön päivittäminen tapahtuu lataamalla aina kokonainen, uusi sivu esimerkiksi linkin klikkaamisen seurauksena. Tässä tekstissä termillä perinteinen verkkopalvelu viitataan juuri tällaiseen toteutustapaan.

2000-luvun aikana selainpuolen rooli on muuttunut merkittävämmäksi: kokonaiset verkkopalvelut saattavat nykyään koostua vain muutamasta, erikseen ladattavasta sivusta, joita käsitellään JavaScriptin avulla selaimessa. Erillisiä kokonaisia sivulatauksia ei tarvitse tehdä reagoitaessa käyttäjän syötteisiin, koska JavaScriptilla on mahdollista muokata sivun sisältöä ja tehdä erillisiä asynkronisia palvelupyyntöjä taustajärjestelmiin tiedon hakemista ja tallentamista varten. Tässä tekstissä termillä moderni verkkopalvelu tarkoitetaan tällaista toteutustapaa.

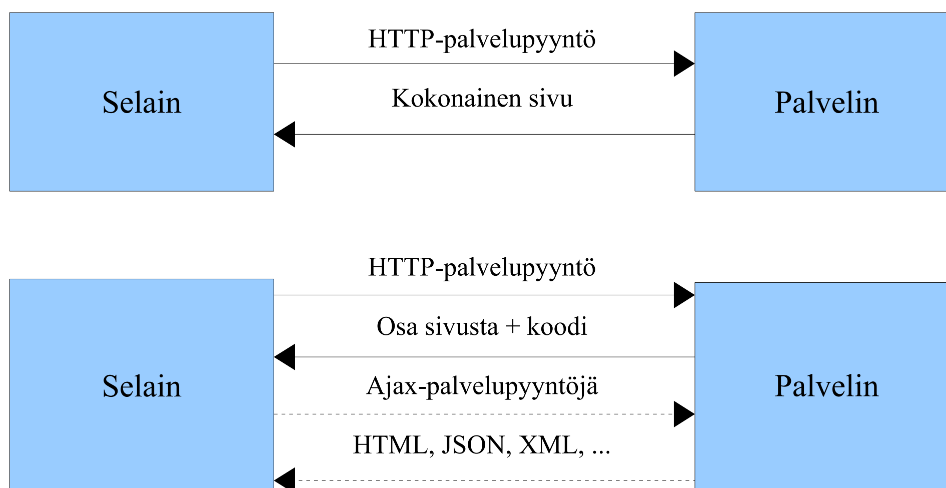
Tässä luvussa käsitellään selainpuolen ohjelmointiin liittyviä tekniikoita keskittyen erityisesti JavaScript-ohjelmointikieleen, koska se on tärkein yksittäinen tekniikka modernien verkkopalveluiden selainpuolen ohjelmoinnissa. Luvussa 4.1 esitellään verkkoselainta ohjelmointiympäristönä yleisemmällä tasolla. Luvussa 4.2 käsitellään tarkemmin JavaScript-ohjelmointikieltä huomioiden etenkin reaktiivisessa ohjelmoinnissa hyödynnettävät ominaisuudet. Luvussa 4.3 esitellään viime vuosina suosiota saaneita, erillisillä kääntäjillä JavaScript-lähdekoodiksi käännettäviä ohjelmointikieliä.

4.1 Verkkoselain ohjelmointiympäristönä

Verkkopalveluiden selainpuolen ohjelmointiympäristö koostuu kolmesta peruselementistä, jotka ovat HTML (hypertext markup language), CSS (cascading style sheets) ja JavaScript. HTML-merkkauskielellä määritellään sivun rakenne ja

sisältö. Rakenne muodostuu sivun sisältämistä elementeistä ja näiden välisistä suhteista: sivu koostuu esimerkiksi otsikoista, navigaatioelementeistä ja tekstikappaleista. Nämä elementit voivat pitää sisällään muita elementtejä. Sivun sisältöä ovat esimerkiksi teksti ja kuvat. CSS-tyylitiedostot puolestaan määrittelevät, miten rakenteinen sisältö esitetään verkkoselaimessa. Käytettävissä olevia esitystapoja ovat esimerkiksi kirjasintyylit, marginaalit, värit, jne. HTML-dokumentti on staattista sisältöä: kerran latauduttuaan sivun sisältö ei muutu.

Koska HTML-dokumentti on staattista sisältöä, samalla verkkosivulla tapahtuva interaktiivinen toiminnallisuus toteutetaan JavaScriptilla tai jollain selaimeen asennettavalla liitännäisellä. Toiminnallisuuden toteuttava koodi suoritetaan aina selaimessa. JavaScriptin tapauksessa suorittamisesta huolehtii selain ja liitännäisen tapauksessa selaimeen asennettu liitännäinen. JavaScriptin etuna liitännäisiin verrattuna on se, että käytännössä kaikki selaimet tukevat sitä suoraan. Erikseen asennettavia liitännäisiä ovat esimerkiksi Adobe Flash, Microsoft SilverLight ja Java Applet -tekniikat, joita käytetään nykyään yhä vähenevässä määrin. Liitännäisten asentamisesta aiheutuvan vaivan lisäksi käyttäjän ei ole välttämättä mahdollista asentaa liitännäistä lainkaan esimerkiksi ilman ylläpitäjän käyttöoikeuksia. Lisäksi liitännäisten tuki mobiililaitteilla on ollut hyvin vaihtelevaa. Monet pitävät myös ongelmallisena riippuvuutta kolmannen osapuolien suljetuista ohjelmistoista, kuten Flashista tai SilverLightista. Näistä syistä suuntaus on ollut viime vuosina vahvasti käyttää JavaScriptia ja muita avoimia standardeja.



Kuva 4.1: Perinteinen ja moderni verkkopalvelu

Kuvassa 4.1 on esitetty perinteisen verkkopalvelun ja modernin verkkopalvelun toimintatavat. Perinteisessä verkkopalvelussa kokonainen verkkosivu ja sen sisältämät elementit ladataan aina erikseen uudella palvelupyynnöllä. Moderni verkkopalvelu taas lataa sivun tarvitseman materiaalin vastaavasti kuin edellinen, mutta tämän jälkeen staattista sisältöä muokataan JavaScriptin avulla ja taustajärjestelmiin voidaan olla yhteydessä Ajax-palvelupyynnöillä ilman, että kokonaisia sivulatauksia tarvitsee suorittaa. Ajax-palvelupyyntöjä esitellään tarkemmin aliluvussa Ajax ja XMLHttpRequest.

Web-sovellusten selainpuolen ohjelmointiin liittyvä tärkeä huomio on myös se, että Ajax-palvelupyyntöjä lukuun ottamatta kaikki koodi suoritetaan selaimessa yhdessä säikeessä. HTML5:n mahdollistama WebWorker-rajapinta on tosin viime vuosina tarjonnut mahdollisuuden monisäieohjelmointiin. Rajapintaa ei käsitellä tässä tutkielmassa tarkemmin, mutta sen hyödyntäminen yleistynee tulevaisuudessa, kun verkkoselaimet alkavat tukea rajapintaa paremmin.

4.2 JavaScript

Koska käytännössä kaikki nykyiset selaimet tukevat JavaScriptia, se on muodostunut viime vuosina tärkeimmäksi verkkosovellusten selainpuolen ohjelmointikieleksi. Myös kaikki tässä tutkielmassa tarkemmin käsiteltävät kirjastot ovat toteutettu JavaScriptilla. JavaScriptin ominaispiirteiden ymmärtäminen on oleellista myös reaktiivisen web-ohjelmoinnin ymmärtämiselle, joten kieltä käsitellään tässä luvussa hieman tarkemmin.

JavaScript-ohjelmointikielen suunnitteli Netscape-yrityksellä työskentelevä Brendan Eich. Kielen ensimmäinen versio julkaistiin vuonna 1995 nimellä Mocha. Nimi vaihdettiin vielä samana vuonna ensin LiveScriptiksi ja tämän jälkeen JavaScriptiksi. JavaScript standardoitiin vuonna 1997 kansainvälisen Ecma International -standardointiorganisaation toimesta, jonka johdosta virallinen standardi kulkee nimellä ECMAScript. JavaScript on ECMAScript-standardin yksi implementaatio [W3C14]. JavaScript kärsi pitkään maineesta ”amatöörien ohjelmointikielenä”, mutta verkkosovellusten selainosuuden monimutkaistuessa ja osaavien ohjelmoijien käyttäessä sitä yhä enenevässä määrin JavaScriptia on alettu pitää varteenotettavana kielenä [Cro01].

JavaScriptissa on tunnetusti joitain epäonnistuneita suunnitteluratkaisuja, mutta myös paljon hyviä ominaisuuksia. Douglas Crockford, joka on ollut pitkään aktiivisena

JavaScriptin kehittäjänä, on kirjoittanut erinomaisen yhteenvedon JavaScriptin hyvistä ja huonoista ominaisuuksista kirjaansa JavaScript: The Good Parts [Cro08]. Huonoista ominaisuuksista aiheutuvia potentiaalisia virheitä on mahdollista minimoida käyttämällä erilaisia lint-työkaluja. Lint-työkalut ovat erillisiä itsenäisiä ohjelmia tai ohjelmointityökaluihin asennettavia liitännäisiä, jotka analysoivat lähdekoodia ja ilmoittavat ohjelmoijalle huonoista ohjelmointikäytännöistä. Eräs suosittu tällainen työkalu JavaScriptille on Crockfordin toteuttama JSLint [Cro02]. Lint-työkalujen lisäksi viime aikoina ovat yleistyneet myös JavaScript-lähdekoodiksi käännettävät ohjelmointikielet, kuten CoffeeScript [Ash14]. Käännettävillä ohjelmointikielillä pyritään piilottamaan JavaScriptin huonoja tai ongelmallisia ominaisuuksia sekä toisaalta tuomaan käyttöön uusia ja hyödyllisiä ominaisuuksia. Tällaisia kieliä esitellään hieman tarkemmin luvussa 4.3. Viime vuosina JavaScript on kasvattanut suosiotaan myös palvelinpuolen ohjelmoinnissa. Suurimpana suosion edistäjänä on ollut Node.js-ohjelmointialusta [Joy14].

Syntaksi ja keskeiset ominaisuudet

JavaScriptin syntaksi muistuttaa melko paljon C-kielen syntaksia. JavaScriptista löytyvät C:n tyyliset kontrollirakenteet, kuten if, for ja while. C:n tapaan myös JavaScript määrittelee erilliset lausekkeet (expression) ja lauseet (statement). Lausekkeet voidaan C:n tapaan erottaa toisistaan puolipisteellä, mutta JavaScriptissa tämä voidaan tehdä myös riviä vaihtamalla. Monien muiden skriptikielten tapaan myös JavaScript on dynaamisesti tyypitetty, ja siinä on automaattinen roskienkeruu. JavaScript tukee myös olio-ohjelmointia. Muihin yleisesti käytettyihin kieliin, kuten Javaan tai C++:aan verrattuna JavaScriptin poikkeavampia ominaisuuksia ovat prototyyppipohjaisuus (prototype-based programming) ja funktiotason näkyvyys (function-level scope).

Funktiotason näkyvyys tarkoittaa sitä, että tavallisten lohkotason elementtien sijaan ainoastaan funktiot vaikuttavat näkyvyyteen. Tämä aiheuttaa monesti ongelmia tottumattomalle ohjelmoijalle, koska useimmat yleiset ohjelmointikielet, kuten Java ja C tai C++ tukevat lohkotason näkyvyyttä.

```
if (true) {
      var a = true;
}
// a:n arvo on tässä kohtaa suoritusta true,
// koska if-lohko ei vaikuta näkyvyyteen

function() {
```

```
      var b = true;
}
// b:n arvo on tässä kohtaa suoritusta undefined,
// koska sitä ympäröivä funktio rajaa näkyvyyden
```

Yllä olevassa lähdekoodissa on esimerkki JavaScriptin näkyvyysmäärittelyistä. Koodissa muuttujan *a* arvo on if-lohkon jälkeen *true*, koska if-lauseke muodostaa ”tavallisen” lohkon, joka ei vaikuta näkyvyyteen. Muuttujan *b* arvoa ei ole puolestaan määritelty funktion jälkeen, eli se on JavaScriptin tyyppiä *undefined*, koska sitä ympäröivä funktio rajoittaa näkyvyyden funktion sisään. JavaScript ei tue muuttujien ja funktioiden näkyvyysmäärittelyä esimerkiksi monista kielistä tuttujen public ja private -avainsanojen avulla, joten kapseloinnin aikaansaamiseksi hyödynnetään funktioiden vaikutusta näkyvyyteen. Esimerkiksi myöhemmin tässä luvussa käsiteltävä moduuli-suunnittelumalli perustuu näkyvyyden rajoittamiseen funktioiden avulla.

Olio-ohjelmointi

JavaScriptissa olio on määritelty kokoelmana ominaisuuksia (property), jotka ovat olioon liittyviä muuttujia. Ominaisuudet on määritelty avain-arvo-pareina, joista avain kertoo ominaisuuden nimen ja arvo sisältää ominaisuuden konkreettisen arvon, kuten tavallistenkin muuttujien tapauksessa. Ominaisuudet voivat olla mitä tahansa JavaScriptin tyyppiä, kuten merkkijonoja, funktioita tai taulukoita, kuten alla olevassa esimerkissä. Uusia olioita voidaan luoda JavaScriptissa usealla tavalla, joista esimerkissä käytetty aaltosulkunotaatio on yksi yleisimmistä.

```
// Luodaan uusi olio aaltosulkunotaatiolla
var exampleObject = {
      value: 'an example string',
      getString: function() { return 'hello'; },
      getArray: function() { return [ 'h', 'e', 'l', 'l', 'o' ]; }
};

// Olion ominaisuuksiin päästään käsiksi niiden nimillä
exampleObject.value;
exampleObject.getString();
exampleObject.getArray();
```

JavaScriptin oliomalli perustuu perinteisemmän luokkamallin sijaan prototyyppimalliin (prototype-based programming). JavaScriptin prototyyppimallissa ei ole lainkaan luokkia, joista luodaan olioita, vaan käytössä on ainoastaan olioita. Uusia olioita ei luoda luokkien pohjalta vaan tyhjästä tai jo olemassa olevien olioiden

pohjalta. Luokkamallin käyttämät ylä- ja aliluokat ja perintä soveltuvat hyvin uudelleenkäytettävän koodin luomiseen. Vaikka JavaScriptissa ei ole käytössä luokkia, niin olioiden välinen perintä ja uudelleenkäytettävän koodin tuottaminen onnistuu prototyyppipohjaisen perinnän (prototypal inheritance) avulla. Prototyyppimallissa jokaisella oliolla on sisäinen prototyyppiviite johonkin toiseen olioon. Nämä viitteet muodostavat prototyyppiketjuja, jotka toimivat siten, että kutsuttaessa olion ominaisuutta etsimistä jatketaan prototyyppiketjussa eteenpäin, mikäli ominaisuutta ei löydy kyseisestä oliosta.

```
// Luodaan olio, jolla on kaksi metodia
var a = {
      getValue: function() { return 'x'; },
      getAnotherValue: function() { return 'y'; }
};

// Luodaan olio, jolla on samanniminen metodi kuin oliolla a
var b = {
      getValue: function() { return 'z'; }
};

// Määritellään olio a b:n prototyyppiolioksi
b.prototype = a;

// Funktio palauttaa arvon 'z', koska kutsuttava funktio on määritelty
// samassa oliossa
b.getValue();

// Funktio palauttaa arvon 'y', koska kutsuttava funktio löytyy a:sta,
// joka on b:n prototyyppiolio
b.getAnotherValue();
```

Yllä olevassa esimerkissä luodaan kaksi oliota, joista ensimmäinen asetetaan jälkimmäisen prototyyppiolioksi. Koska *a* asetetaan *b*:n prototyyppiolioksi, *b*:llä on käytössä myös kaikki *a*:n ominaisuudet. Kutsuttaessa *b*:n metodia *getValue()*, käytetään olion omaa metodia eikä *a*:n vastaavaa, koska kutsuissa käytetään aina prototyyppiketjun lähintä sopivaa metodia. g*etAnotherValue*:n tapauksessa metodia ei löydy oliosta itsestään, joten sitä lähdetään etsimään prototyyppioliosta. Koska prototyyppiolioksi on määritelty *a*, josta kyseinen metodi löytyy, suoritetaan se.

Prototyyppimalli on erittäin ilmaisuvoimainen, ja sillä pystytään simuloimaan helposti myös luokkamallia [Cro08, s. 46]. Tästä voidaan hyötyä esimerkiksi tilanteissa, joissa prototyyppimalli ei ole tuttu ohjelmoijalle ja mallintaminen tuntuu luontevammalta luokkamallin avulla. Myös jotkut JavaScriptiksi käännettävät ohjelmointikielet, kuten CoffeeScript, mahdollistavat luokkapohjaisen olio-ohjelmoinnin.

JavaScriptin olioita hyödynnetään myös nimiavaruuksien määrittelyssä, koska

JavaScriptissa ei ole suoraa tukea nimiavaruuksien (namespace) määrittelylle, kuten esimerkiksi C++:ssa *namespace*-avainsanan avulla. Globaalin nimiavaruuden sotkeutumisen välttämiseksi normaalina käytäntönä on luoda suuremmalle ohjelmakokonaisuudelle, kuten kokonaiselle kirjastolle tai komponentille, globaaliin nimiavaruuteen yksi olio, jonka alle sijoitetaan kaikki ohjelman muuttujat, funktiot ja alikomponentit. Tästä toimintatavasta käytetään usein nimeä nimiavaruussuunnittelumalli (namespace pattern) [Ste10, s. 87]. Esimerkiksi luvussa 5.2 käsiteltävä RxJS-kirjasto määrittelee globaaliin nimiavaruuteen olion Rx, jonka alla kaikki kirjaston muut komponentit sijaitsevat. Luvussa 5.1 käsiteltävä Flapjax-kirjasto puolestaan määrittelee globaaliin avaruuteen olion F.

Funktionaalinen ohjelmointi

JavaScript ei ole varsinaisesti funktionaalinen ohjelmointikieli, mutta se tarjoaa joitain funktionaalisista kielistä tuttuja ominaisuuksia mahdollistaen osittaisen funktionaalisen ohjelmoinnin. JavaScriptissa funktiot ovat ensimmäisen luokan kansalaisia, eli niitä voidaan käsitellä samoin kuin muitakin tietotyyppejä. Tästä johtuen JavaScript mahdollistaa korkeamman asteen funktiot (higher order functions), sulkeumat (closures) sekä nimettömät ja sisäkkäiset funktiot. Näitä ominaisuuksia hyödynnetään aktiivisesti web-ohjelmoinnissa ja etenkin reaktiivisessa web-ohjelmoinnissa.

Korkeamman asteen funktioilla tarkoitetaan sitä, että funktioita on mahdollista käyttää funktioiden kutsuparametreina ja paluuarvoina. Asynkronisessa JavaScript-ohjelmoinnissa funktioita käytetään yleisesti kutsuparametreina takaisinkutsujen määrittelemisessä.

```
// Ladataan tietoa Ajax-palvelupyynnöllä ja annetaan takaisinkutsu-
// parametriksi nimetön funktio
$.get('/url/to/backend/', function(data) {
      // Kun asynkroninen palvelupyyntö on valmistunut, muuttujasta
      // data löytyy palvelupyynnön palauttama tieto
});
```

Yllä on esimerkki funktiosta kutsuparametrina. *$.get* on jQuery-kirjaston [Jqu14] metodi, jolla lähetetään HTTP GET -muotoinen Ajax-palvelupyyntö. Metodin jälkimmäiseksi parametriksi annetaan nimetön funktio, jota kutsutaan automaattisesti palvelupyynnön valmistuttua. Kutsuttaessa nimetöntä funktiota se saa automaattisesti palvelupyynnön palauttaman tiedon *data*-parametrina.

Ajax ja XMLHttpRequest

Ajax eli Asynchronous JavaScript and XML on eräs keskeisimmistä tekniikoista, joihin modernit verkkopalvelut perustuvat. Määritelmän ja termin esitteli alun perin Jesse James Garrett artikkelissaan Ajax: A New Approach to Web Applications [Gar05]. Garrettin määritelmässä Ajax-tekniikan ydin on JavaScriptilla tapahtuva verkkosivun sisällön dynaaminen muokkaaminen sekä tiedon siirtäminen selaimen ja palvelimen välillä XMLHttpRequest-oliota hyödyntämällä.

XMLHttpRequest mahdollistaa asynkronisen kommunikaation selaimen ja palvelimen välillä ilman kokonaisen verkkosivun uudelleenlatausta. Rajapinnan kehitti alun perin Microsoft, ja sittemmin myös muut selainvalmistajat ovat lisänneet tuen sille. Rajapinta on parhaillaan W3C:n standardoitavana. XMLHttpRequest tukee GET ja POST -tyyppisiä HTTP ja HTTPS-palvelupyyntöjä. HTML5-standardin mukana ovat tulleet myös WebSocketit, jotka saattavat tulevaisuudessa vallata sijaa XMLHttpRequestilta, koska WebSocketit mahdollistavat myös datan työntämisen selaimelle palvelimelta.

Moduuli ja laajennettu moduuli -suunnittelumallit

Moduuli ja laajennettu moduuli -suunnittelumalleja on hyödynnetty paljon tätä tutkielmaa varten toteutetussa sovelluksessa, joten niitä on esitelty hieman tarkemmin. Moduuli-suunnittelumalli (module pattern) [Ste10, s. 97] on yksi yleisimmistä JavaScript-ohjelmoinnissa käytettävistä kapselointitavoista. Moduuli määrittelee julkisen rajapinnan ja kapseloi sisälleen toteutuksen. Toteutuksessa voidaan hyödyntää muuttujia ja funktioita, jotka eivät näy moduulin ulkopuolelle. Moduuli-suunnittelumallia käytettäessä moduulista muodostuu ainoastaan yksi instanssi globaaliin nimiavaruuteen, joten se muistuttaa jossain määrin olio-ohjelmoinnista tunnettua ainokainen-suunnittelumallia (singleton pattern) [Gam94, s. 127].

Moduulin määrittely on mahdollista tehdä ainoastaan yhdessä tiedostossa. Joskus on kuitenkin hyödyllistä jakaa suurikokoisen moduulin toteutus useampaan tiedostoon. Tämä onnistuu laajennettu moduuli -suunnittelumallin (augmented module pattern) avulla. Toteutuksen useampaan tiedostoon jakamisen lisäksi moduulin laajentamisesta on hyötyä kirjoitettaessa uudelleenkäytettävää koodia. Useille moduuleille yhteinen koodi voidaan toteuttaa alkuperäiseen moduuliin, jota laajennetaan uusilla moduuleilla. Laajentaminen muistuttaa hieman olio-ohjelmoinnin yliluokka–aliluokka -suhdetta. Perittyihin luokkiin verrattuna laajennettu moduuli-

suunnittelumallin heikkoutena on se, että laajentava moduuli ei pääse suoraan käsiksi alkuperäisen moduulin yksityisiin muuttujiin ja funktioihin. Yksityisten ominaisuuksien käsittely joudutaan tekemään moduulin julkisen rajapinnan kautta vastaavasti kuin käytettäessä moduulia normaalistikin.

Laajennettu moduuli -suunnittelumalli ei ole yhtä yleisesti tunnustettu suunnittelumalli kuin esimerkiksi tavallinen moduuli tai nimiavaruus-suunnittelumallit, mutta siitä on mainintoja useissa WWW-lähteissä [Che10].

```
// Määritellään yksinkertainen moduuli
var Module = (function() {
	// Muuttujan näkyvyys rajautuu moduuliin
	var private = 1;

	// Palautetaan olio, joka sisältää moduulin julkisen rajapinnan
	return {
		set: function(value) { private = value; },
		get: function() { return private; },
		increment: function() { private++; }
	};
})();

// Laajennetaan moduulia uusilla metodeilla
(function(module) {
	// Funktion näkyvyys rajautuu laajentavaan moduuliin,
	// mutta siihen lisätään myöhemmin viite julkiseen rajapintaan
	function doSubtract() {
		module.set(module.get() - 1);
	}

	// Lisätään alkuperäiseen moduuliin uusi metodi
	module.subtract = doSubtract;
})(Module);
```

Yllä olevassa listauksessa luodaan *Module*-niminen moduuli, joka tarjoaa kolme metodia yksityisen *private*-muuttujan käsittelyyn. Moduulin määrittely tapahtuu luomalla funktio, joka suoritetaan välittömästi (immediately-invoked function expression). Funktio sisältää moduulin toiminnallisuuden ja palauttaa julkisen rajapinnan oliona, jolloin sisällön näkyvyys rajautuu ainoastaan palautettuun olioon. Moduulin määrittelyn jälkeen hyödynnetään laajennettu moduuli -suunnittelumallia ja lisätään alkuperäiseen moduuliin *subtract*-metodi, joka vähentää alkuperäisessä moduulissa olevan *private*-muuttujan arvoa yhdellä. Koska laajennetusta moduulista ei ole suoraa pääsyä alkuperäisen moduulin yksityisiin muuttujiin, toiminnallisuus on toteutettu käyttämällä moduulin *set* ja *get* -metodeja. Lopullinen *Module*-moduulin rajapinta koostuu siis kaikista neljästä metodista. Laajennetun moduulin määrittely tapahtuu kuten alkuperäisenkin, mutta lisäksi välittömästi suoritettavalle funktiolle

annetaan kutsuparametriksi alkuperäinen moduuli, jolloin se on käytettävissä funktion sisällä.

4.3 JavaScriptiksi käännettävät kielet

JavaScriptin monista huonoista puolista johtuen [Cro01] viime vuosina sovelluskehityksessä on ollut kasvavana trendinä JavaScriptiksi käännettävien ohjelmointikielten käyttäminen. Näistä kielistä erityisen suosittu on CoffeeScript, jonka tarkoituksena piilottaa JavaScriptin huonoja puolia ja tuoda esiin hyviä puolia. CoffeeScriptin syntaksi eroaa melko paljon JavaScriptin syntaksista, ja se on saanut vaikutteista muun muassa Ruby ja Python -ohjelmointikielistä. Tavallista JavaScriptia on myös mahdollista kirjoittaa suoraan CoffeeScript-lähdekoodin sekaan. Käännetyn JavaScript-koodin suorituskyvyn luvataan olevan vähintään samaa luokkaa tai parempaa kuin suoraan JavaScriptina kirjoitetun [Ash14].

CoffeeScriptin on luonut Jeremy Ashkenas, ja sen ensimmäinen versio julkaistiin vuonna 2009. Se on ollut erittäin suosittu projekti GitHub-palvelussa julkaisustaan lähtien. CoffeeScriptin lisäksi kiinnostavaa on muiden tyypitettyjen JavaScriptiksi käännettävien kielien yleistyminen. Eräs tunnettu esimerkki tällaisista on Googlen vuonna 2011 julkaisema Dart-ohjelmointikieli [Goo14a].

JavaScriptiksi käännettävien ohjelmointikielten yhtenä haasteena on suoritusaikana ilmenevien virheiden jäljittäminen (debugging), koska virheet syntyvät kääntäjän tuottamasta koodista, eivätkä suoraan ohjelmoijan kirjoittamasta lähdekoodista. Ainakin CoffeeScriptin tapauksessa tilanne on parantumassa: kielen uusimmissa versioissa on mukana source maps -ominaisuus, jonka avulla pystytään jäljittämään suoritusaikana ilmenevien virheiden sijainti alkuperäisessä CoffeeScript-lähdekoodissa.

5 Reaktiivinen web-ohjelmointi

Modernien verkkosovellusten kasvava monimutkaisuus ja asynkronisuus aiheuttaa useita haasteita: sovellukset kommunikoivat taustajärjestelmien – kuten tietokantapalvelinten – kanssa asynkronisesti. Lisäksi verkkosovellusten selainpuolen koodissa on usein hyvin paljon käyttäjän syötteisiin reagoimista. Perinteinen ratkaisu asynkronisten tapahtumien ja käyttäjän syötteiden käsittelyyn on ollut tarkkailija-suunnittelumalli (observer pattern), jossa tarkkailijat (observer) rekisteröityvät tarkkailtavalle (subject). Kun tarkkailtavassa tapahtuu jotain tarkkailijoiden kannalta mielenkiintoista, tarkkailtava ilmoittaa tästä kaikille rekisteröityneille tarkkailijoille kutsumalla jotain niiden metodeista [Gam94, s. 293].

Kokonaista tarkkailija-suunnittelumallia hieman yksinkertaisempi, usein JavaScript-ohjelmoinnissa käytetty tapa on pelkkien parametrina annettujen nimettömien funktioiden kutsuminen asynkronisen operaation valmistuttua. Takaisinkutsut (callback) ja niitä käyttävät suunnittelumallit, kuten tarkkailija, ovat ongelmallisia etenkin silloin, kun nämä ovat riippuvaisia muista takaisinkutsuista. Verkko-sovellusten toteuttamisessa esiintyy usein tilanteita, joissa joudutaan ketjuttamaan toisistaan riippuvia takaisinkutsuja. Koodista tulee tällöin vaikeasti seurattavaa. Alla on esimerkki kuvitteellisen chat-sovelluksen selainosuuden toimintalogiikasta, jossa käyttäjä rekisteröidään taustajärjestelmään kuuluvalle palvelimelle, liitytään vapaaseen keskusteluhuoneeseen ja lähetetään automaattinen viesti kaikille huoneessa olijoille. Esimerkki on Boixin, De Meuterin ja Kambonan tutkimuksesta An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications [KBD13].

```
registerToChatOnServer(username, function(rooms) {
      joinAvailableRoom(rooms, function(roomname) {
            sendChatToAll(roomname, msg, function(reply) {
                  showChatReply(reply);
            })
      })
});
```

Ensimmäisenä ongelmana koodissa on sisäkkäiset takaisinkutsut, jotka vaikeuttavat lukemista. Ongelmasta käytetään toisinaan nimityksiä ”callback hell” ja ”pyramid of doom” [KBD13]. Lisäksi useat takaisinkutsut hajauttavat koodin useaan eri paikkaan vaikeuttaen kokonaisuuden hahmottamista. Kokonaisuuden hajautumisen yhteydessä puhutaan myös ”asynkronisesta spagetista” [Mic14a].

Kuten Boixin, De Meuterin ja Kambonan tutkimuksessa todetaan, reaktiivisella ohjelmoinnilla pystytään ratkaisemaan ainakin joitain näistä ongelmista. JavaScriptin ollessa luonteva valinta yleiseen web-ohjelmointiin, se on myös luonteva valinta reaktiiviseen web-ohjelmointiin. Tapoja toteuttaa reaktiivista ohjelmointia selainympäristössä on käytännössä kaksi: voidaan käyttää itsenäistä ohjelmointikieltä, joka käännetään JavaScriptiksi erillisellä kääntäjällä, tai ottaa käyttöön JavaScriptilla toteutettu kirjasto. RxJS noudattaa jälkimmäistä mallia ja Flapjaxia on mahdollista käyttää kummalla tahansa tavalla. Tällä hetkellä eräitä suosituimmista vaihtoehdoista reaktiiviseen web-ohjelmointiin tuntuvat olevan RxJS ja bacon.js. Sanahaku GitHub-hostingpalveluun sanalla "RxJS" palauttaa 71 koodivarastoa (repository). Vastaava haku sanalla "bacon.js" palauttaa 58 ja "Flapjax" 15.

5.1 Flapjax

Flapjax oli ensimmäinen selainympäristöön suunnattu funktionaalinen reaktiivinen ohjelmointikieli [Mey09]. Flapjax on tekijöidensä mukaan luokiteltu FRP-kieleksi, joten sitä käsitellään tässä tekstissä myös reaktiivisena ohjelmointikielenä. Koska Flapjax on JavaScriptin laajennos, sillä on myös kaikki JavaScriptin ominaisuudet. Etenkin funktionaalisen ohjelmoinnin mahdollistavat ominaisuudet ovat erittäin hyödyllisiä reaktiivisella paradigmalla ohjelmoitaessa huomioiden reaktiivisten kielten ja funktionaalisen ohjelmoinnin yhteyden. Flapjaxin kehitys tuntuu olleen jo jonkin aikaa pysähtynyt, ja viimeisin versio (2.1) on vuodelta 2009. Flapjaxista löytyy kuitenkin akateemista tutkimusta, ja se on ollut tärkein uranuurtaja reaktiivisen selainohjelmoinnin saralla, minkä takia sitä käsitellään tarkemmin tässä tekstissä. Lisäksi Flapjax vaikuttaa olevan tällä hetkellä ainoa selainohjelmointiin suunniteltu kieli, joka mahdollistaa läpinäkyvän reaktiivisuuden.

Flapjaxia on mahdollista käyttää JavaScript-kirjastona tai omana ohjelmointikielenä, joka käännetään Flapjax-lähdekoodista JavaScript-lähdekoodiksi. Ohjelmointikielen käyttämisellä kirjastoon verrattuna saavutetaan joitain etuja, kuten automaattinen nostaminen ja Flapjaxin kirjoittaminen HTML:n sekaan (inline Flapjax). Koska Flapjax-ohjelmointikieli toimii JavaScriptin laajennoksena, näiden välinen yhteistoiminta on saumatonta. Koska reaktiivisuus on tällöin toteutettu ohjelmointikielitasolla, saavutetaan myös läpinäkyvästä reaktiivisuudesta johtuva "reaktiivisempi tuntuma", kuten nostamista käsittelevässä luvussa 3.2 esiteltiin.

Inline Flapjax -lähdekoodi lisätään {! !} -merkkien väliin HTML-lähdekoodin sekaan. Flapjax-kääntäjä kääntää merkkien välissä olevan koodin JavaScriptiksi.

```
<input id="name"
      style={! { borderColor: validColor(ccNumValid) } !}
      disabled={! !ccNumValid !}/>

function validColor(valid) {
      return valid ? 'aqua' : 'cyan'; }

var ccNumField = $B('ccNum');
var ccNumValid = validCC(ccNumField);
```

Yllä olevassa koodissa, joka on otettu Flapjaxin esittelevästä tutkimuksesta, määritellään HTML-tekstikenttä käyttäjän nimeä varten, jonka on tarkoitus olla aktiivinen ainoastaan, jos käyttäjä on syöttänyt toiseen kenttään kelvollisen luottokorttinumeron. Lisäksi kentälle on määritelty reunukset, joiden väri myös riippuu luottokortti-numeron oikeellisuudesta. Tekstikenttä-elementissä *ccNumValid* on reaktiivinen arvo, joka on tosi, mikäli muualle käyttöliittymän syötetty luottokorttinumero on oikea. *validColor* on puolestaan funktio, joka boolean-tyyppisen parametrin arvosta riippuen palauttaa värin merkkijonona. Tekstikenttä-elementin jälkeen listauksessa on tavallisena Flapjax-koodina määritelty *validColor*-funktio ja kaksi muuttujaa. *ccNumField* asetetaan reaktiiviseksi arvoksi, joka vastaa muualla HTML-dokumentissa määritellyn, luottokorttinumerolle tarkoitetun tekstikentän arvoa. *validCC* on muualla lähdekoodissa määritelty Flapjax-kielinen funktio, joka tarkistaa luottokorttinumeron oikeellisuuden. Kun luottokorttikentän arvo muuttuu, *ccNumValid*-muuttujan arvo lasketaan automaattisesti uudelleen ja nimi-kenttä päivitetään.

Sama toiminnallisuus saataisiin aikaan ilman inline-tyyppistä toteutusta määrittelemällä *ccNumValid*-muuttujan asettamisen yhteyteen koodi, joka asettaa nimi-kentän reunukset ja aktiivisuuden.

Vaikka Flapjax-kielisen lähdekoodin kirjoittamisessa on paljon etuja verrattuna Flapjax-kirjaston käyttämiseen, varjopuolena on kääntämisen vaatima ylimääräinen välivaihe kehitysprosessiin. Välivaihe ei kuitenkaan välttämättä ole ongelma: viime vuosina selainohjelmoinnissa on ollut kasvavana trendinä automaattisten build-työkalujen, kuten Gruntin [Gru14] käyttö. Tällaiset työkalut vastaavat pitkälti Unix-maailmasta tuttua Make-työkalua. Build-työkaluilla Flapjaxin tai jonkin muun kielen kääntäminen JavaScriptiksi voidaan tehdä automaattisesti osana build-prosessia. Kuten aiemmin mainittiin, selainympäristössä on alettu suosia enenevässä määrin JavaScriptiksi käännettäviä ohjelmointikieliä kuten CoffeeScriptia [Ash14] ja Dartia [Goo14a], mikä saattaisi kertoa käännettävien kielten hyötyjen olevan ehdottomasti

haittoja suurempia.

5.2 Reactive Extensions for JavaScript (RxJS)

Reactive Extensions for JavaScript (RxJS) [Mic14c] on Microsoftin kehittämä reaktiivisen JavaScript-ohjelmoinnin mahdollistava kirjasto, joka pohjautuu Microsoftin Reactive Extensions -sovelluskehykseen. Sovelluskehyksestä löytyy myös Microsoftin julkaisemat .NET ja Windows Phone -versiot C#-ohjelmointikielelle. Rx:ää on käytetty laajalti myös kaupallisessa ohjelmistotuotannossa, ja siitä löytyy kolmansien osapuolien aktiivisessa kehityksessä olevat RxJava- ja RxScala-versiot. Esimerkiksi videopalvelu Netflix käyttää verkkopalvelunsa käyttöliittymäkoodissa RxJS:ää. RxJS eroaa FRP-tyyppisistä kielistä – kuten Flapjaxista – siinä, että siitä puuttuu kokonaan reaktiiviset arvot. RxJS on edelleen aktiivisen kehitystyön alla ja uusia versioita julkaistaan tasaisesti. Edellisessä luvussa esitetty Flapjax-kielinen koodi voitaisiin toteuttaa RxJS:llä esimerkiksi seuraavasti:

```
var source = Rx.Observable.fromEvent(
      document.getElementById('ccNum'), 'change');

var subscription = source.subscribe(function(e) {
      var ccNumValid = validCC(e.value);
      var nameElem = document.getElementById('name');
      nameElem.style.borderColor = validColor(ccNumValid);
      nameElem.disabled = !ccNumValid;
});

function validColor(valid) {
      return valid ? 'aqua' : 'cyan';
}
```

Koodissa luodaan ensin *Observable*-tyyppinen muuttuja, joka vastaa reaktiivisen ohjelmoinnin käsitteistössä tapahtumavuota. Vuohon ilmestyy aina tapahtuma, mikäli luottokorttitekstikentän arvo muuttuu. Seuraavaksi lähteen lähettämät tapahtumat tilataan käsiteltäväksi Observablen *subscribe*-metodilla. Metodille annetaan parametriksi nimetön funktio, jonka kutsuparametri on tapahtuman aiheuttanut elementti. Tämän jälkeen tarkastetaan, onko luottokorttinumero oikea, ja päivitetään vastaavasti nimi-kentän reunuksen väriä ja aktivointitietoa.

Koodi vastaa hyvin lähelle Flapjax-kirjaston avulla toteutettua ei-inline-tyyppistä versiota, jossa käytettäisiin ainoastaan tapahtumavoita. Boix, De Meuter ja Kambona ovat tutkimuksessaan tulleet siihen johtopäätökseen, että selainohjelmoinnissa tapahtumavuot olivat yleisesti reaktiivisia arvoja käyttökelpoisempia. Koska RxJS on

edelleen aktiivisen kehityksen alla, siitä löytyy myös huomattavasti monipuolisemmat työkalut tapahtumavoiden käsittelyyn kuin Flapjaxista.

5.3 Vaihtoehto reaktiiviselle ohjelmoinnille: lupaukset (promise)

Boix, De Meuter ja Kambona esittävät tutkimuksessaan reaktiiviselle ohjelmoinnille yhdeksi vaihtoehdoksi lupauksia (promise) [KBD13]. Käsite esiteltiin ensimmäisen kerran jo 1970-luvulla Friedmanin ja Wisen toimesta [FrW76]. Lupaus on olio, joka toimii edustajana (proxy) asynkroniselle laskennalle, jonka valmistumisajankohtaa tai suorituksen onnistumista ei vielä tiedetä. Web-ohjelmoinnissa hyvä sovelluskohde lupauksille on asynkronisten taustajärjestelmäkutsujen yhteydessä. Lupauksien avulla asynkronisia kutsuja on mahdollista yhdistää ja niiden suoritusjärjestyksistä saadaan selkeämpiä kuin perinteisillä takaisinkutsuilla.

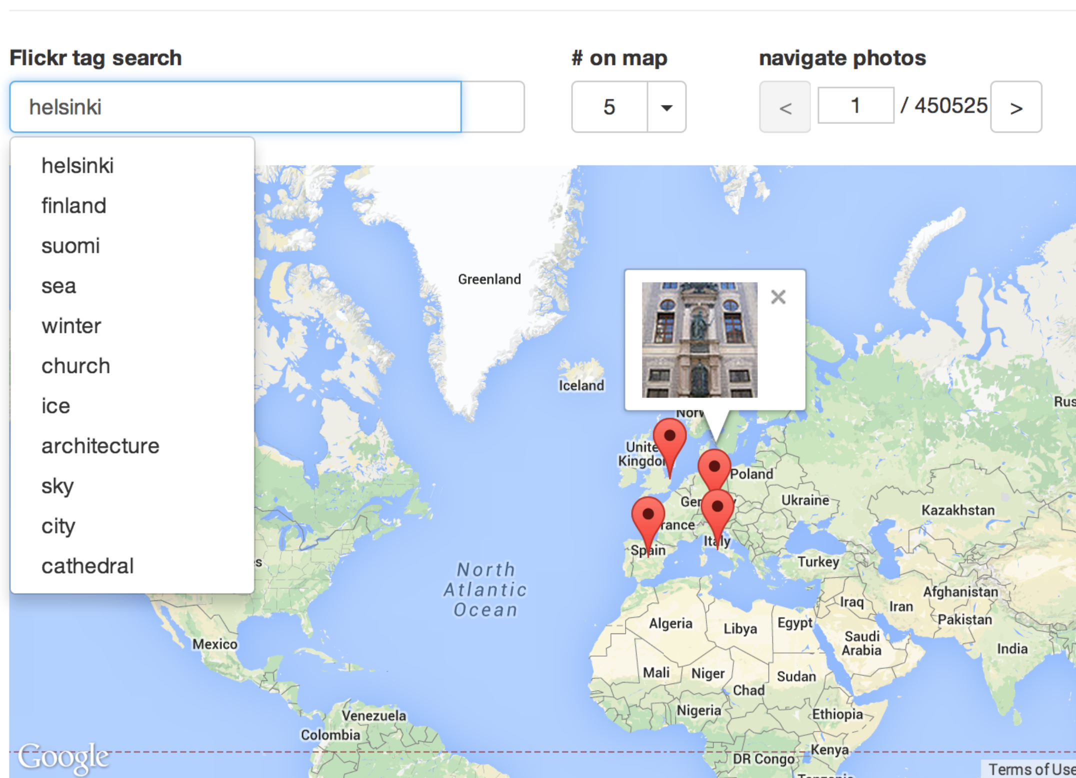
```
Q.fcall(registerToChatOnServer)
      .then(joinAvailableRoom)
      .then(sendChat)
      .then(function(reply) {
            showChatReply(reply)
      }, function(error) {
            // Catch error from all async operations
      })
      .done();
```

Yllä oleva koodilistaus on lupauksia käyttävä toteutus, joka vastaa tämän luvun alussa esitettyä sisäkkäisten takaisinkutsujen johdosta epäselvää koodiesimerkkiä. Koodi on toteutettu Q-kirjastolla [Kow14]. Q-kirjaston avulla suoritusjärjestyksen vaativat operaatiot voidaan ketjuttaa luontevasti ilman, että ohjelmoijan tarvitsee seurata, missä kohtaa sisäkkäisten takaisinkutsujen ketjua ollaan menossa. Lisäksi mahdollisten virhetilanteiden käsittely onnistuu siististi ja keskitetysti. Takaisinkutsuilla toteutetussa alkuperäisessä listauksessa virhetilanteita ei ollut edes otettu huomioon. Myös jQuery-kirjastosta löytyy tuki lupauksille Deferred-olion muodossa. Lupauksia voidaan käyttää esimerkiksi tilanteissa, joissa asynkroniset kutsut monimutkaistavat koodin luettavuutta ja jonkin reaktiivisen ohjelmointikielen käyttöönotto tuntuu liian suurelta paradigmamuutokselta.

6 Esimerkkisovellus: Flickr & Google Maps Mashup

Tätä tutkielmaa varten suunniteltiin web-sovellus, josta tehtiin kolme erilaista toteutusta: kaksi reaktiivista ja yksi ei-reaktiivinen versio. Sovelluksen versioiden toteutus vaihtelee keskenään paljon, mutta niiden toiminnallisuus on täysin sama. Sovellus on luvussa 4 esitellyn modernin verkkopalvelun arkkitehtuurin mukainen. Se mahdollistaa kuvien etsimisen ja hakemisen Flickr-kuvapalvelusta [Fli14], ja kuvien näyttämisen sijaintitiedon (geolocation) perusteella Google Maps -kartalla [Goo14b]. Haettavia kuvia on lisäksi mahdollista selata erillisten navigaatioelementtien avulla. Sovelluksesta on haluttu tehdä riittävän yksinkertainen, jotta tämän työn puitteissa on ollut mahdollista toteuttaa versioita useammalla eri tekniikalla, mutta toisaalta riittävän yksityiskohtainen, jotta tutkimustulokset olisivat yleistettävissä laajempiin tuotantosovelluksiin. Suunnittelussa on huomioitu myös käyttäjäkokemus: usein yksinkertaisiinkin sovelluksiin tulee helposti paljon monimutkaisuutta pienistä käyttöliittymän yksityiskohdista, kuten virheilmoituksista ja käynnissä olevien palvelupyyntöjen indikoinnista käyttäjälle. Useimmissa reaktiivista ohjelmointia hyödyntävissä esimerkeissä ja oppimateriaaleissa [Mic14d, Fla14] tätä näkökulmaa ei ole otettu kovin hyvin huomioon.

Flickr & Google Maps Mashup esittelee melko hyvin aiemmin tutkielmassa käsiteltyjä reaktiivisen ohjelmoinnin potentiaalisia käyttökohteita selainpuolella: sovelluksessa käytetään paljon asynkronisia Ajax-palvelupyyntöjä ja hyödynnetään monipuolisesti käyttöliittymän avulla tapahtuvaa käyttäjän toimiin reagointia. Tässä luvussa esitellään tutkielmaa varten toteutetun sovelluksen toimintaa, arkkitehtuuria ja toteutusta sekä sille luotuja automaattisia testejä.



Kuva 6.1: Kuvakaappaus esimerkkisovelluksesta

6.1 Toiminta ja toiminnalliset vaatimukset

Flickr & Google Maps API Mashup -sovellus on yhdellä verkkosivulla toimiva yksittäinen näkymä, joka on esitetty kuvassa 6.1. Sovelluksen käyttöliittymä koostuu hakukentästä, jonka avulla käyttäjän on mahdollista hakea tunnisteita – eli tägejä (tag) – Flickr-kuvapalvelusta. Tägit ovat lyhyitä, yhden sanan mittaisia tunnisteita, joita voidaan liittää kuvaan metatiedoksi. Syötettäessä tekstiä hakukenttään, palvelu ehdottaa syötettyyn hakusanaan liittyviä tunnisteita kentän alle avautuvassa pudotusvalikossa. Pudotusvalikko päivittyy automaattisesti sitä mukaa, kun tekstiä syötetään, eikä käyttäjän tarvitse lähettää hakua erikseen.

Valittaessa jokin ehdotetuista tunnisteista tai painettaessa rivivaihtonäppäintä, suoritetaan kuvahaku valitulla tunnisteella merkityistä Flickr-palvelun julkisista kuvista. Haettavilta kuvilta edellytetään myös löytyvän sijaintitieto, jotta ne voidaan sijoittaa Google Maps -kartalle. Kuvahaun valmistuttua kuvia ja niiden sijaintia

ilmaisevat karttapinnit sijoitetaan kartalle kuviin liittyvän sijaintitiedon perusteella. Vietäessä hiiren osoitin pinnin päälle, käyttäjälle näytetään esikatselukuva (thumbnail). Esikatselukuvat ladataan selaimen muistiin ennen pinnien näyttämistä kartalla, jotta kuvat aukeavat välittömästi. Pinnejä näytetään kartalla aina hakukentän oikealla puolella olevaan pudotusvalikkoon valittu määrä. Vaihdettaessa lukumäärää pinnit ladataan kartalle uudelleen ja oikeassa reunassa olevaa navigaatioelementtiä päivitetään. Kuvia haetaan aikajärjestyksessä uusimmasta vanhimpaan ja niiden selaaminen tapahtuu sivun oikeassa reunassa olevan navigaatioelementin avulla. Navigaatioelementin avulla voidaan valita seuraava tai edellinen kokoelma kuvia, tai syöttää kokoelman numero suoraan tekstikenttään. Edellisen ja seuraavan kokoelman painikkeet ovat pois toiminnasta, mikäli ollaan kokoelmien ensimmäisellä tai viimeisellä sivulla. Kun sovellus käynnistetään eikä kuvia ole vielä ladattu, koko navigaatio on pois toiminnasta.

Karttapinnejä klikattaessa käyttäjälle näytetään esikatselukuvaa suurempi versio kuvatiedostosta sekä kuvan metatietoja. Tämä tapahtuu lataamalla ensin metatiedot Flickr-palvelusta, minkä jälkeen suurempi kuvatiedosto esiladataan vastaavasti kuten esikatselukuvat aiemmin. Latauksen aikana käyttäjälle näytetään latauskuvaketta, ja suuren kuvan ja metatietojen sisältävä ponnahdusikkuna avataan vasta lataamisen valmistuttua. Sovelluksessa näytetään animoitu latauskuvake aina, kun verkosta ollaan lataamassa tietoa – tämä pätee myös tunniste-ehdotusten ja kuvahaun tapauksissa. Käyttäjälle myös näytetään aina virheilmoitus, mikäli jokin palvelupyyntö epäonnistui.

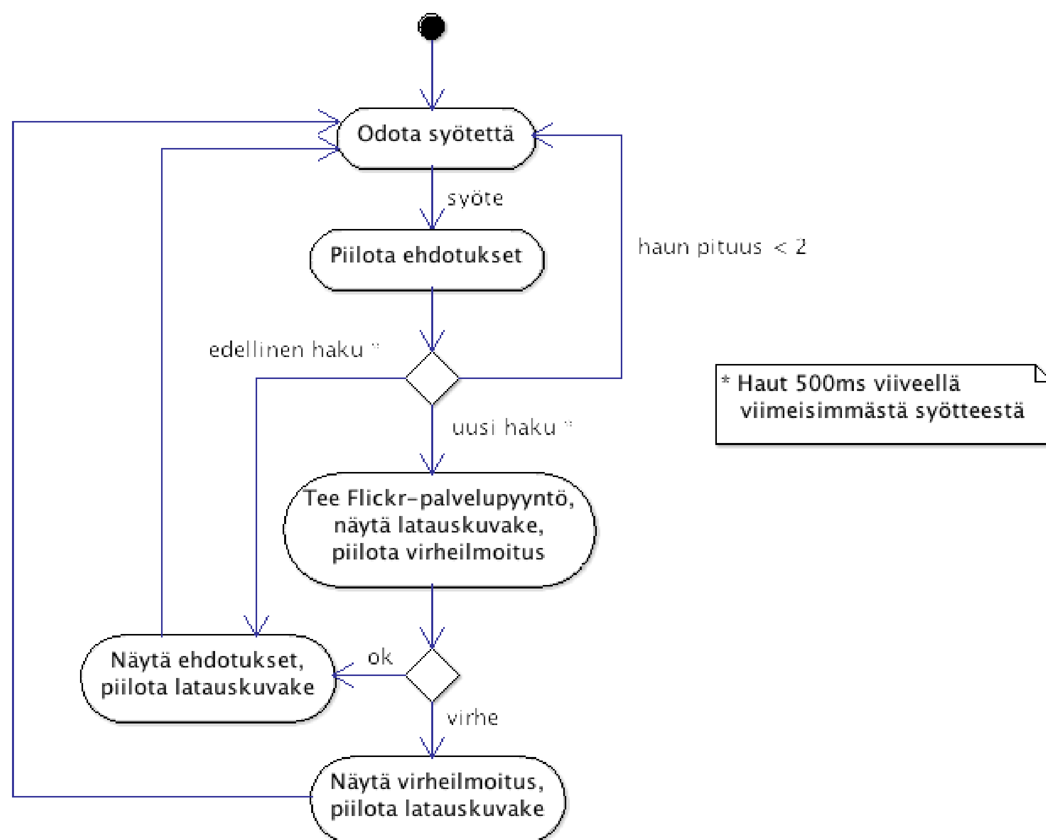
Sovelluksen toimintalogiikka koostuu kolmesta suuremmasta kokonaisuudesta: tunniste-ehdotusten käsittelystä, kuvahausta ja kuvan tarkempien tietojen näyttämisestä. Näitä kokonaisuuksia käsitellään seuraavaksi hieman tarkemmin.

Tunniste-ehdotusten hakeminen

Tunniste-ehdotuksia sisältävä pudotusvalikko päivitetään aina, kun hakukentän sisältö muuttuu. Jotta palvelupyyntöä ei tehtäisi jokaisen syötetyn merkin kohdalla, haku suoritetaan vasta aina puolen sekunnin jälkeen viimeisen merkin painalluksesta. Tällöin käyttäjä ehtii kirjoittamaan haluamansa sanan kokonaan ilman, että kirjoittamisen lomassa lähetetään turhia palvelupyyntöjä. Toisaalta puolen sekunnin viive on niin lyhyt, että syötteen valmistuttua ehdotukset sisältävä pudotusvalikko aukeaa kuitenkin lähes välittömästi. Viiveen lisäksi ehdotushakuja on rajattu hakutekstin pituuden perusteella: tyhjiä tai ainoastaan yhden merkin pituisia

syötteitä ei käsitellä.

Pudotusvalikko avataan aina, kun hakukenttä klikataan aktiiviseksi. Valikko sulkeutuu aina, kun klikataan jossain muualla tai painetaan esc-näppäintä. Jos hakusanaa vastaavat ehdotukset on jo kerran ladattu, uutta palvelupyyntöä ei tehdä, vaan näytetään ehdotukset, jotka saatiin jo edellisellä latauskerralla. Ladattaessa ehdotuksia näytetään hakukentän oikealla puolella animoitu latauskuvake, joka piilotetaan jälleen latauksen valmistuttua. Mikäli ehdotusten lataus epäonnistui, käyttäjälle näytetään virheilmoitus. Tunniste-ehdotusten lataaminen voi epäonnistua käytännössä kahdella tavalla: joko itse palvelupyyntö epäonnistuu tai sitten Flickr-rajapinnasta tuleva vastaus on virheellinen. Kuva 6.2 esittää ehdotushaun tarkan toimintalogiikan toimintokaaviona (activity diagram).

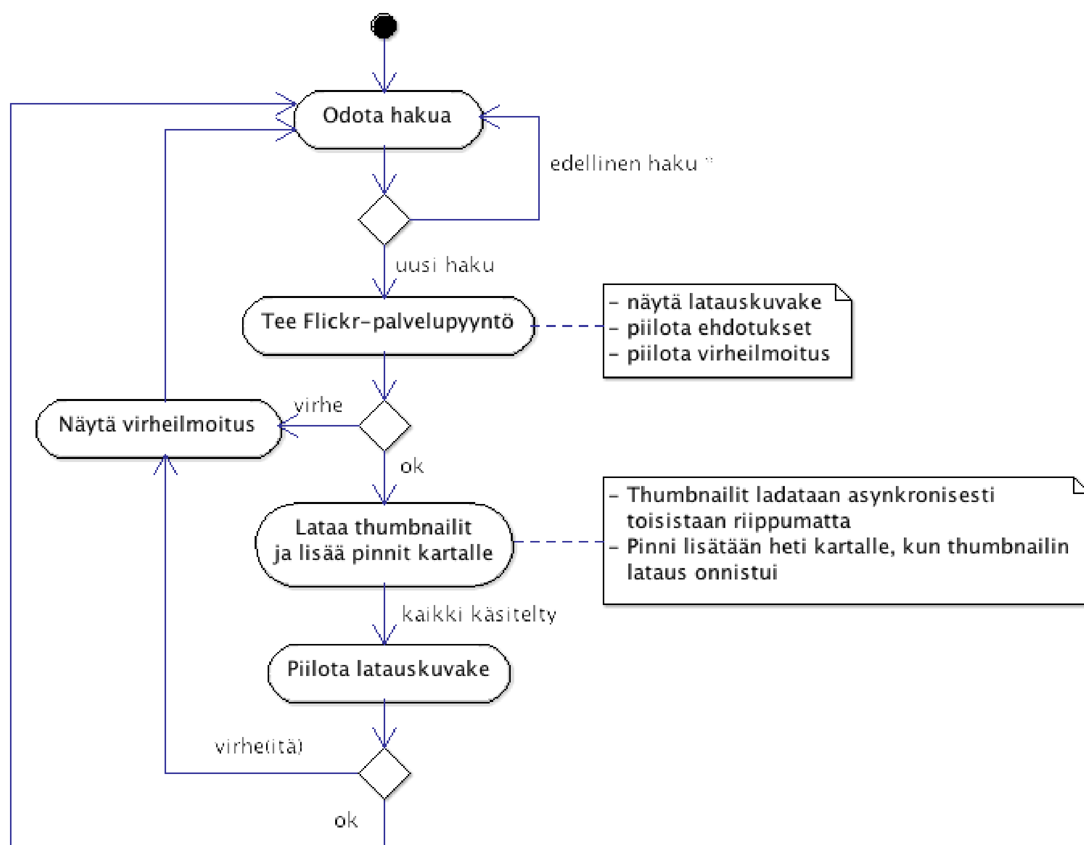


Kuva 6.2: Toimintokaavio ehdotusten hakemisesta

Kuvien hakeminen

Kun tunnistetta klikataan pudotusvalikosta tai hakukentässä painetaan

rivivaihtonäppäintä, suoritetaan kuvahaku Flickr-palvelusta. Kuvahaun tarkka toiminta on esitetty toimintokaaviona kuvassa 6.3. Lähetettäessä palvelupyyntö tunnisteen perusteella, käyttäjälle näytetään latauskuvake kuten ehdotusten tapauksessakin. Mikäli kuvahaku on edellistä hakua vastaava, siihen ei reagoida mitenkään. Onnistunut palvelupyyntö palauttaa kokoelman kuvien tietoja, jolloin päästään käsiksi myös esikatselukuvien URL-osoitteisiin. Kaikki kuvatiedostot ladataan asynkronisesti selaimen muistiin, ja kuvia vastaavat pinnit sijoitetaan kartalle sitä mukaa, kun lataus kyseiselle kuvalle valmistuu. Latauskuvaketta näytetään, kunnes kaikki kuvat ovat käsitelty, minkä jälkeen se piilotetaan. Mikäli joitain kuvista ei saatu ladattua, jätetään myös näitä vastaavat pinnit sijoittamatta kartalle. Mikäli Flickr-palvelupyyntö tai yhdenkin esikatselukuvan lataus epäonnistui, näytetään virheilmoitus.



Kuva 6.3: Toimintokaavio kuvien hakemisesta

Yksittäisen kuvan tiedot

Klikattaessa pinniä sen yläpuolelle avautuu lisätietoikkuna, joka sisältää suuremman

version kuvasta sekä metatietoja, kuten kuvaajan, sijainnin ja kuvatekstin. Metatiedot haetaan erillisellä Flickr-palvelupyynnöllä, joka suoritetaan heti, kun pinniä on klikattu. Kuten esikatselukuvien tapauksessakin, myös suurempi kuvatiedosto esiladataan ja käyttäjälle näytetään latauskuvaketta, kunnes sekä Flickr-palvelupyyntö että kuvatiedosto ovat molemmat latautuneet. Mikäli jommankumman lataamisessa tapahtui virhe, näytetään virheilmoitus. Kuvatiedot ja kuva voidaan sulkea esc-näppäimellä tai viemällä hiiren osoitin jonkin muun pinnin päälle, jolloin lisätietoikkuna sulkeutuu ja kyseisen pinnin esikatseluikkuna avautuu.

6.2 Arkkitehtuuri

Sovellus koostuu yhdestä HTML-sivusta, jota käsitellään JavaScriptilla. Sovelluksen toimintalogiikka on jaettu useampaan JavaScript-komponenttiin, jotka on toteutettu luvussa 4.2 esitellyn moduuli-suunnittelumallin mukaisesti. Jokainen moduuli on sijoitettu omaan tiedostoonsa, joka on nimetty moduulin nimiseksi. Sovelluksen tiedostorakenne löytyy liitteestä 2. Lähes jokaisella komponentilla on yhteistä koodia eri toteutusversioiden välillä. Lähdekoodin turhaa toisteisuutta on vältetty hyödyntämällä luvussa 4.2 esiteltyä laajennettu moduuli -suunnittelumallia. Moduulien täydentäminen esimerkkisovelluksessa on tehty siten, että alkuperäinen moduuli sisältää kaikille versioille yhteisen toiminnallisuuden, jota laajennetaan versiokohtaisesti uusilla ominaisuuksilla. Jotta JavaScriptin globaali nimiavaruus pysyisi mahdollisimman siistinä, moduulit on sijoitettu yhteisen *mashup*-nimisen olion alle luvussa 4.2 esitellyn nimiavaruus-suunnittelumallin mukaisesti.

Kaikki versiot hyödyntävät jQuery-kirjastoa, jota on hyödynnetty sivun HTML-elementtien muokkaamisessa ja Ajax-palvelupyyntöjen lähettämisessä. jQuery on muodostunut viime vuosien aikana lähes standardiksi kaikkien verkkopalvelujen selainpuolen toteutuksissa. jQueryn lisäksi sovelluksessa hyödynnetään Bootstrap 3.0 -sovelluskehystä [OtT14]. Bootstrap on kokoelma työkaluja verkkosovellusten selainpuolen koodin toteuttamiseen. Bootstrapin keskeisimmän osan muodostavat monipuoliset tyylimäärittelyt, joiden avulla on nopeaa toteuttaa tyylikkäitä ja yhdenmukaisia elementtejä, kuten nappeja, ponnahdusikkunoita, otsikoita ja virheilmoituksia. Bootstrap mahdollistaa myös useille päätelaitteille suunnattujen responsiivisten verkkosivujen ja -sovellusten toteuttamisen vaivattomasti. Responsiivisuudella tarkoitetaan tässä yhteydessä sitä, että verkkosivu tai -sovellus huomioi päätelaitteen, jolla sitä käytetään. Tällöin esimerkiksi ulkoasun leveys voidaan sovittaa automaattisesti päätelaitteen resoluution mukaiseksi. Tyylimäärit-

telyjä täydentämään Bootstrapista löytyy myös JavaScript-komponentteja, joilla on mahdollista esimerkiksi animoida valikoita ja toteuttaa ponnahdusikkunoita ja kuvakaruselleja. Lisäksi Bootstrapin responsiiviset ominaisuudet, tyylit ja JavaScript-komponentit on toteutettu siten, että ne toimivat ja näyttävät yhtenäisiltä kaikissa suhteellisen uusissa verkkoselaimissa. Tässä sovelluksessa Bootstrapista on hyödynnetty lähinnä tyylimäärittelyjä, joiden avulla sovelluksesta saatiin hyvin pienellä vaivalla tyylikkään ja modernin näköinen. Koska suurin osa tyylimäärittelyistä tulee Bootstrapista, sovelluksen omassa tyylitiedostossa on ainoastaan muutamia määrittelyitä. Tyylien lisäksi pudotusvalikot hyödyntävät Bootstrapin JavaScript-komponenttia. Bootstrapin responsiivisia ominaisuuksia ei ole hyödynnetty, koska sovelluksen toteuttaminen responsiivisesti ei vaikuta tässä työssä tutkittavaan JavaScript-ohjelmointiin. Mikäli esimerkkisovelluksesta tehtäisiin oikea tuotantosovellus, Bootstrapin responsiivisia ominaisuuksia kannattaisi todennäköisesti ottaa käyttöön.

Flickr ja Maps API -rajapintoja on helppo hyödyntää ulkopuolisissa sovelluksissa, ja niiden käyttäminen on ilmaista ei-kaupallisissa sovelluksissa. Molempien käyttöönotto onnistuu rekisteröitymällä palvelujen sivustoille, josta saa avaimen palvelun käyttöä varten. Flickr API on REST-rajapinta, jolle voidaan lähettää HTTPS-palvelupyyntöjä. Avain lähetetään parametrina jokaisen palvelupyynnön mukana. Palvelupyynnön vastaus on mahdollista saada useammassa formaatissa, kuten XML:nä tai JSON:ina. Esimerkkisovelluksessa käytetään JSON-muotoisia vastauksia. Maps API on useista komponenteista koostuva JavaScript-kirjasto. Maps API:n käyttöönotto tapahtuu lataamalla kirjaston sisältävä JavaScript-koodi sivulle. Lataaminen voidaan tehdä helposti suoraan script-HTML-elementin avulla, jolle annetaan kirjaston osoitteen lisäksi avain GET-parametrina. API:n latauduttua kaikki sen toiminnallisuudet, kuten kartan lataaminen, pinnien ja ponnahdusikkunoiden näyttäminen ovat käytettävissä.

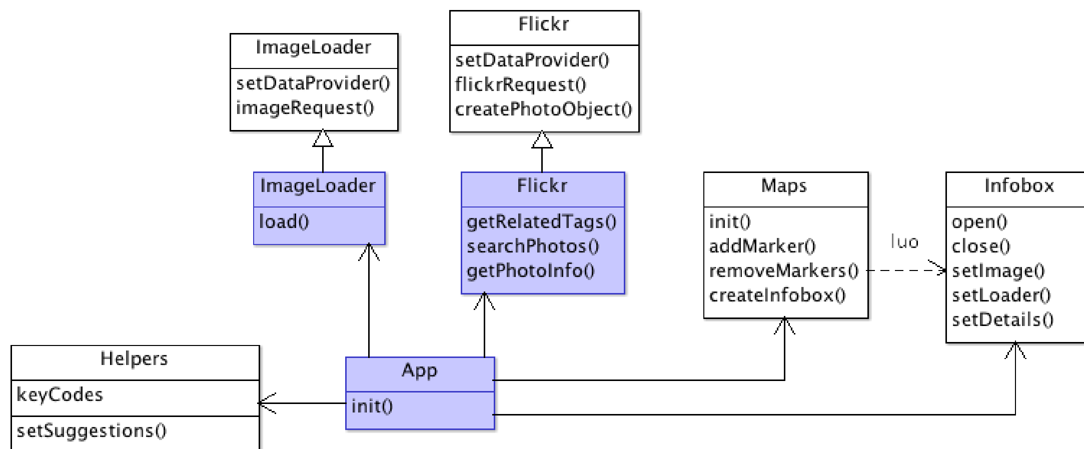
Esimerkkisovelluksesta on tehty erilliset toteutukset pelkällä JavaScriptilla, RxJS-kirjaston avulla ja Flapjax-kielellä. Sovelluksen sisältävälle HTML-sivulle on mahdollista antaa GET-parametrit *app* ja *mock*, joista edellisellä määritetään näytettävä toteutusversio ja jälkimmäisen avulla voidaan korvata osa toiminnallisuudesta mock-olioilla (mock object) – esimerkiksi mock-olioilla varustettu RxJS-versio ladattaisiin osoitteella *index.html?app=rxjs&mock=true*. Mock-oliot helpottavat testauksen toteuttamista, ja niistä on kerrottu lisää luvussa 6.3.

Komponentit

Kaikki sovelluksen versiot koostuvat samoista komponenteista, mutta niiden toteutustapa vaihtelee paljon eri versioiden välillä. Toteutustavan lisäksi myös komponenttien rajapinnat vaihtelevat jonkin verran. Rajapinnoissa keskeisimpänä erona on se, että JavaScript-toteutus käyttää asynkronisten operaatioiden yhteydessä takaisinkutsuja ja RxJS- sekä Flapjax-toteutukset tapahtumavoita. Sovelluksen arkkitehtuuri koostuu moduuli-suunnittelumallin mukaisista komponenteista App, ImageLoader, Flickr, Maps ja Helpers. Moduuleista kaikkia muita paitsi Maps:ia ja Helper:iä on laajennettu versiokohtaisesti aiemmin mainitun laajennettu moduuli -suunnittelumallin mukaisesti.

Kuvassa 6.4 on esitetty sovelluksen moduulit luokkakaavion avulla. Moduulien kuvaamiseen on käytetty luokkakaaviota, koska JavaScript-moduulien kuvaamiseen tarkoitettua työkalua ei kirjoitushetkellä ollut saatavilla. Kaavion tarkoituksena on toimia korkeamman tason arkkitehtuurikuvauksena, joten moduulien metodien kutsu- tai paluuparametreja tai niiden tyyppejä ei ole esitetty. Parametreja ja tyyppejä on mahdollista tarkastella liitteestä 3, josta löytyvät kaikkien moduulien kaikkien versioiden julkiset rajapinnat.

Kaaviossa valkoisella taustalla esitetyt moduulit ovat kaikille versioille yhteisiä, alkuperäisiä moduuleja, ja sinisellä taustalla esitetyt edellisiä täydentäviä, versiokohtaisia moduuleja. Viitteet moduulien välillä eivät ole omistussuhteita, vaan ne ainoastaan esittävät sitä, että moduuli käyttää toista moduulia suoraan sen julkisen rajapinnan kautta. *App*-moduulilla on kuitenkin omistussuhde *Infobox*-komponenttiin, joka kapseloi Maps API:n ponnahdusikkunan toiminnallisuuden. Toisin kuin muut komponentit, Infobox ei ole moduuli, vaan tavallinen JavaScript-olio. Uutta Infobox-oliota ei ole mahdollista luoda suoraan, vaan luonti tapahtuu aina Maps-moduulissa olevan tehdasmetodin (factory method) *createInfobox*-avulla. Tämän toteutustavan avulla koko karttatoiminnallisuus on saatu kapseloitua siististi yhdeksi omaksi rajapinnaksi.



Kuva 6.4: Esimerkkisovelluksen moduulit

App-moduuli on sovelluksen pääkomponentti, joka käyttää muita komponentteja niiden julkisten rajapintojen avulla. Muista komponenteista ei ole viitteitä pääkomponenttiin tai toisiinsa. App-komponentin eri versioilla on niin vähän yhteistä keskenään, että ne on pidetty kokonaan itsenäisinä, eikä niillä ole yhteistä alkuperäistä moduulia, jota laajennettaisiin.

ImageLoader-moduuli tarjoaa yksinkertaisen rajapinnan kuvien asynkroniseen esilataamiseen (preloading) *load*-metodin avulla. ImageLoader:ia hyödynnetään sekä esikatselukuvien että ison kuvan lataamisessa. Metodin kutsu- ja paluuparametreissa on eroja eri versioiden välillä, koska JavaScript-komponentin toteutus perustuu takaisinkutsuihin ja RxJS- sekä Flapjax-toteutukset tapahtumavoihin.

Flickr-moduuli tarjoaa rajapinnan hakusanaan liittyvien tagien hakemiseen, tagiin liittyvien kuvien hakemiseen ja yksittäisen kuvan tarkempien tietojen hakemiseen Flickr-palvelusta. Rajapinta muodostuu *getRelatedTags*, *searchPhotos* ja *getPhotoInfo* -metodeista, jotka ovat kaikki asynkronisia. Tieto pyydetään Flickr-rajapinnasta JSON-muotoisena, jolloin sitä voidaan käsitellä JavaScriptilla suoraan sellaisenaan. Vastaavasti kuin *ImageLoader*-moduulin tapauksessa, kutsu- ja paluuparametrit vaihtelevat.

Maps-moduuli tarjoaa yksinkertaisen rajapinnan Maps API:n käsittelyyn. Lisäksi se ylläpitää kartan tilaa. Komponentin toteutus on kaikille versioille yhteinen, koska sen tarkoituksena on ainoastaan kapseloida palvelun tarvitsemat Google Maps -rajapinnan osat siististi. Komponentin *addMarker*-metodin kutsuparametrien kaksi takaisin-kutsua oltaisiin voitu korvata tapahtumavoilla *ImageLoader*:in ja *Flickr*:in tapaan,

mutta muutos olisi ollut niin pieni komponentin muuhun koodimäärään verrattuna, että se jätettiin siksi tekemättä.

Helpers-moduuliin on koottu kaikille versioille yhteisiä sekalaisia toiminnallisuuksia. Moduulista löytyy sovelluksessa käytettävien erikoismerkkien näppäimistökoodit ja metodi, joka luo HTML:n tunniste-ehdotusten pudotusvalikkoa varten.

6.3 Toiminnalliset testit

Esimerkkisovellukselle on asetettu tarkat toiminnalliset vaatimukset, jotta eri tekniikoin toteutetut versiot ovat keskenään vertailukelpoisia. Koska sovellus sisältää paljon asynkronisia palvelupyyntöjä Flickr ja Google Maps -rajapintoihin, näiden suoritusjärjestys ja mahdollisten virheilmoitusten näyttämisen ajankohta tulee olla tarkasti määritelty. Toiminnallisten vaatimusten toteutumisen varmistamiseksi sovellukselle on luotu kattavat automaattiset toiminnalliset testit. Toiminnalliset testit testaavat sovelluksen toimintaa kokonaisuutena, joten niitä voidaan ajatella myös kokonaisvaltaisina järjestelmätesteinä (end-to-end testing, system testing).

Esimerkkisovelluksen toiminnalliset testit ovat jaettu viiteen erilliseen testijoukkoon (test suite). Testijoukot testaavat sovelluksen alkutilannetta, tunnistehakua, kuvahakua, kuvien navigointia ja ponnahdusikkunoita. Testijoukoissa on yhteensä 78 testiä, ja niiden lähdekoodi koostuu yhteensä 592 rivistä JavaScriptiä kommentit mukaan lukien. Testien suorittaminen testilaitteistolla (Apple MacBook Pro, 2.3 Ghz Intel Core i7, 8 GB 1600 Mhz DDR3) kesti keskimäärin vähän alle minuutin. Tuloste testien suorittamisesta löytyy liitteestä 1, josta on nähtävissä myös yksittäiset testitapaukset.

Testityökalut

Testit ovat toteutettu PhantomJS:aa [Hid14] hyödyntävällä CasperJS-testaustyökalulla [Per14]. PhantomJS on WebKit-selainmoottorin päälle rakennettu käyttöliittymätön verkkoselain (headless browser), jota voidaan ohjata JavaScript-rajapinnan avulla. CasperJS puolestaan laajentaa tätä rajapintaa tarjoamalla lisätyökaluja testaukseen. WebKit-selainmoottorin tunnetuin käyttökohde on Applen Safari-verkkoselain [App14]. Testien ajamisen etuna käyttöliittymättömällä verkkoselaimella on ennen kaikkea niiden suorittamisen nopeus ja se, että testit on helppo suorittaa komentoriviltä. Testiohjelmaan on myös toteutettu testattavan version valinta komentoriviparametrien avulla – esimerkiksi RxJS-version testit

saadaan suoritettua komennolla *casperjs test test.js –app=rxjs*.

```
// Aloitetaan testijoukko
casper.test.begin('Testing navigation', function(test) {

  // Ladataan sovelluksen sisältävä verkkosivu
  casper.start(page, function() {

    // Syötetään hakukenttään hakusana ja odotetaan pudotusvalikkoa
    this.sendKeys('#search', 'helsinki', { keepFocus: true });
    this.waitUntilVisible('#search-dropdown-menu');

    // Klikataan pudotusvalikkoa ja odotetaan, kunnes kaikki pinnit
    // ovat lisätty kartalle
    this.then(function() {
      this.click('#search-dropdown-menu li:nth-child(8) a');
      waitWhileLoadingAll(function() {

        // Suoritetaan testejä navigaation tilalle
        test.assertExists('#prev-photos[disabled="disabled"]',
          'Previous button is disabled on first page');
        test.assertNotExists('#next-photos[disabled="disabled"]',
          'Next button is enabled on first page');
        test.assertNotExists('#current-page[disabled="disabled"]',
          'Page field is enabled on first page');
        test.assertFieldCSS('#current-page', '1',
          'First page is selected');
      });
    });
```

Yllä oleva koodilistaus sisältää navigaatiota testaavan testijoukon alustuksen ja neljä ensimmäistä testiä. *Casper.start*-komento lataa *page*-muuttujaan määritellyn verkkosivun, joka on tässä tapauksessa *index.html*, jossa sovellus sijaitsee. Tämän jälkeen hakukenttään sijoitetaan hakusana *helsinki* ja jäädään odottamaan, että hakusanaa vastaavista ehdotuksista koostuva pudotusvalikko on näkyvillä. Valikkoa joudutaan odottamaan, koska tunniste-ehdotukset haetaan aina asynkronisesti 500 millisekunnin viiveellä viimeisen merkin syöttämisestä. Asynkronisista odotustilanteista, kuten *waitUntilVisible*-käskystä, voidaan jatkaa eteenpäin testiohjelmassa CasperJS:n käskyn *then* avulla. Then-kutsulle parametriksi annettava funktio suoritetaan vasta, kun sitä edeltävä asynkroninen funktio on valmistunut. Kun pudotusvalikko on näkyvillä, klikataan valikon kahdeksatta linkkiä. Klikkaaminen tapahtuu simuloimalla hiiren klikkausta *click*-metodin avulla. Tässä tapauksessa klikattava kohde on määritelty CSS3-standardin mukaisella syntaksilla. CSS3:n sijaan voitaisiin myös käyttää XPath-syntaksia tai antaa klikattavan kohdan pikselikoordinaatit. Kaikkia kolmea tapaa on käytetty sovelluksen testeissä. Klikkauksen jälkeen *waitWhileLoadingAll*-funktiolla odotetaan, että kaikki pinnit ovat

sijoitettu kartalle. Kun pinnit ovat kartalla, suoritetaan yksittäisiä testejä *assert*-metodeilla, joita CasperJS:stä löytyy hyvin monipuolinen valikoima. Esimerkiksi yllä olevassa koodissa navigaation tilaa testataan HTML-elementtien attribuuttien avulla. w*aitWhileLoadingAll*-metodi on toteutettu erikseen juuri tämän sovelluksen testejä varten. Sen lisäksi testikoodissa on määritelty monia muita omia apufunktioita. Yllä olevaan koodiin on lisätty kommentteja – alkuperäisten testien kommenttitiheys ei ole näin suuri.

CasperJS:aa tunnetumpi vaihtoehto toiminnalliseen selaintestaukseen on Selenium-työkalu [Sel14], joka on verkkoselainliitännäinen (plugin). Seleniumilla on mahdollista ohjata selaimen toimintaa erillisellä skriptikielellä, ja se on saatavilla useimpiin suosittuihin selaimiin. Skriptejä voi toteuttaa Seleniumilla samaan tyyliin kuin CasperJS:lläkin, mutta testit suoritetaan käyttöliittymättömän selaimen sijaan oikeassa verkkoselaimessa. Seleniumin etuna on se, että testit voidaan suorittaa usealla eri verkkoselaimella ja näin varmistua koodin toimiminen kaikilla selaimilla. Seleniumin varjopuolena taas on se, että kokonaisen verkkoselaimen täytyy olla käynnissä, jolloin testien suorittaminen on hitaampaa ja yhtä komentorivikäskyä hankalampaa. Koska tässä tapauksessa tarkoituksena oli testata ainoastaan sovelluksen eri versioiden yhdenmukaisuutta yhdellä selaimella, työkaluksi valittiin CasperJS, joka mahdollisti testien ajamisen hyvin nopeasti ja näppärästi komentoriviltä.

Mock-oliot

Jotta testit olisivat helposti toistettavissa, ulkoisia rajapintoja kutsuttaessa samojen kutsuparametrien tulee tuottaa aina sama vastaus jokaisella suorituskerralla. Tämä on haastavaa etenkin Flickr-rajapinnan tapauksessa, koska palvelupyyntöjen vastauksien sisältö on alati muuttuvaa: tunnisteiden suosittuus vaihtelee ja kuvia tulee lisää tai poistuu. Tästä johtuen testejä suoritettaessa Flickr- ja ImageLoader-komponentit ovat korvattu mock-olioilla (mock object). Mock-oliot ovat alkuperäisen rajapinnan toteuttavia olioita, joiden alkuperäinen toiminnallisuus on korvattu jollain muulla. Mock-olio -nimi viittaa tarkalleen ottaen olio-ohjelmoinnin olioihin, joten tässä tapauksessa olisi ehkä oikeampaa puhua mock-moduuleista. Mock-olio -termiä käytetään kuitenkin tässä yhteydessä tarkoittamaan mock-olioita moduuleille toteutettuna.

Flickr- ja ImageLoader-mock-oliot toteuttavat alkuperäiset rajapinnat, mutta toteutus on muutettu sellaiseksi, että ne palauttavat aina vakiomuotoista tietoa, jolloin

metodien suoritus on aina sama suorituskerrasta riippumatta. Mock-olioiden etuna on myös se, että palauttamalla moduuleissa määriteltyä vakiomuotoista tietoa palvelupyyntöjen sijaan, metodikutsujen nopeus on lähes vakio. Flickr- ja ImageLoader-mock-oliot lisäksi laajentavat alkuperäisiä rajapintoja laajennettu moduuli -suunnittelumallin avulla. ImageLoader mock-olio laajentaa rajapintaa *isLoading*-, *failNext*- ja *delayNext*-metodeilla. *isLoading*-metodin avulla on mahdollista tutkia, onko moduuli parhaillaan lataamassa kuvaa. *failNext*-metodi puolestaan asettaa moduulin tilan sellaiseksi, että seuraava *load*-metodilla tapahtuva latauskutsu epäonnistuu. Tämän avulla virhetilanteiden testaaminen on helppoa. *delayNext*-metodilla voidaan määrittää, kuinka kauan seuraava latauskutsu tulee kestämään. *delayNext*-metodista on hyötyä testattaessa esimerkiksi useamman asynkronisen latauksen valmistumista. Flickr mock-olio laajentaa alkuperäistä Flickr-rajapintaa ImageLoaderia vastaavilla *isLoading* ja *failNext* -metodeilla.

Moduulien Ajax-palvelupyynnöt ja asynkroniset kuvalatauspyynnöt ovat korvattu riippuvuusinjektio-suunnittelumallin (dependency injection) avulla. Riippuvuus-injektio toimii siten, että joitain osia komponentin toiminnallisuudesta on mahdollista asettaa moduulin ulkopuolella. ImageLoader ja Flickr -moduulien tapauksessa tämä onnistuu alkuperäisen moduulin *setDataProvider*-metodin avulla. Metodilla on mahdollista määritellä, miten moduuli hakee tietonsa. Oletustoteutus on käyttää Ajax-palvelupyyntöä Flickr-tiedon hakemiseen tai asynkronista palvelupyyntöä kuvan lataamiseen. Mock-oliot korvaavat oletustoteutuksen sellaisiksi, että palvelupyyntö palauttaa aina vakiovastauksen kutsuparametreista riippuen. Korvaava toteutus myös huolehtii kutsujen viivästämisestä ja epäonnistumisesta, mikäli *delayNext* tai *failNext* -metodeja on kutsuttu.

Jos Maps API:lle olisi Flickr ja ImageLoader -moduulien tapaan toteutettu mock-olio, olisivat kaikki testit mahdollista suorittaa paikallisessa testiympäristössä vaikka kokonaan ilman internet-yhteyttä. Tämä olisi myös nopeuttanut testejä paljon, koska Google Maps tekee lukuisia asynkronisia palvelupyyntöjä etenkin kartan muodostavia kuvia ladattaessa. Google Mapsin korvaaminen mock-oliolla olisi kuitenkin ollut erittäin hankalaa, koska kartta on erittäin monimutkainen käyttöliittymäelementti. Karttapalvelun toimintatapa on myös aina sama suorituskerrasta riippumatta, jolloin testaaminen onnistuu hyvin ilman mock-oliotakin: lähetettäessä sijaintikoordinaatit palvelupyynnön mukana, voidaan olettaa, että pinnit näytetään kartalla aina samassa kohtaa. Myös testien suorittaminen oli kohtuullisen nopeaa ilman Maps-mock-olion toteuttamista, joten se jätettiin tekemättä.

7 Sovelluksen analyysi ja tulokset

Reaktiivisen ohjelmoinnin vaikutusta lähdekoodiin tutkitaan tässä tutkielmassa kahdella tavalla. Ensimmäinen tapa on esimerkkisovelluksen lähdekoodin staattinen analyysi. Toinen tapa on muutosskenaariot, joilla tutkitaan, miten jonkin toiminnallisuuden lisääminen tai muuttaminen tapahtuu sovelluksen jokaisen version kohdalla. Molemmilla tavoilla saatujen tulosten analysoinnissa tarkastellaan myös niihin vaikuttaneita syitä lähdekooditasolla, jolloin pystytään erittelemään kielten yksittäisten ominaisuuksien vaikutusta tuloksiin. Tutkimuksesta saatujen tulosten toivotaan myös antavan ennusteita siitä, miten reaktiivisen ohjelmoinnin soveltaminen vaikuttaa tämän tutkielman sovellusta suurempiin ohjelmistoihin. Lähdekoodin staattisella analyysilla mitataan reaktiivisen ohjelmoinnin vaikutusta lähdekoodin kokoon ja kompleksisuuteen. Hyödyntämällä staattisen analyysin tuloksia yhdessä muutosskenaarioista saatujen tulosten kanssa pyritään ensisijaisesti muodostamaan johtopäätös siitä, mitkä ovat reaktiivisen toteutuksen edut verrattuna ei-reaktiiviseen toteutukseen. Tulosten avulla pyritään myös saamaan tietoa siitä, mikä ovat FRP-kielten – kuten Flapjaxin – reaktiivisten arvojen lisähyöty verrattuna pelkkiin RxJS:n määrittelemiin tapahtumavoihin, sekä selvittää mitkä ovat reaktiivisen ohjelmointikielen edut reaktiiviseen kirjastoon verrattuna.

Toteutettaessa Flapjax-kielistä versiota havaittiin, että Flapjax-kääntäjällä ei ole mahdollista kääntää pelkästä Flapjax-koodista koostuvia tiedostoja, vaan käännettävien dokumenttien tulee olla HTML-dokumentteja, jotka sisältävät HTML-koodin, Flapjax- ja JavaScript-koodin, sekä mahdollisen inline Flapjax -koodin. Tämä aiheutti ongelmia Flapjax-version sovittamisessa esimerkkisovelluksen yleiseen arkkitehtuuriin, jossa HTML ja JavaScript sijaitsevat erillisissä omissa tiedostoissaan. Tutkimuksessa on haluttu myös pitää tarkka fokus pelkässä reaktiivisessa JavaScript-ohjelmoinnissa, joten inline Flapjaxia ei ole käytetty. Tästä johtuen Flapjax-kääntäjän vastuulle olisi jäänyt ainoastaan automaattinen nostaminen. Edellisten syiden takia Flapjax-versio on toteutettu kielen kirjastoversiolla, mutta analyysissa on tuotu erikseen esiin tilanteet, joissa oltaisiin voitu hyödyntää automaattista nostamista.

Tämä luku jakautuu kahteen osaan: metodiikkaan ja analyysiin. Kahdessa ensimmäisessä aliluvussa määritellään mitattavat staattiset ominaisuudet ja muutosskenaariot. Kahdessa jälkimmäisessä aliluvussa esitetään mittausten ja muutosskenaarioiden tulokset ja lopuksi tehdään johtopäätökset tuloksista.

7.1 Mitattavat attribuutit

Lähdekoodin laadun ja ominaisuuksien mittaamisen pohjana on käytetty Fentonin ja Pfleegerin kirjaa Software Metrics: A Rigorous and Practical Approach [Fep98]. Kirjassa esitellyn mittauskehyksen [Fep98, luku 3] mukaisesti mittaaminen on tässä työssä rajattu ohjelmiston sisäiseen laatuun. Sisäisellä laadulla tarkoitetaan ominaisuuksia, joita voidaan mitata lähdekoodista ilman, että ohjelmistoa tarvitsee suorittaa tai ottaa huomioon suoritusympäristöä. Yleisimpiä staattisella analyysilla mitattavia arvoja ovat koodin kokoa ja monimutkaisuutta mittaavat koodirivien lukumäärä ja McCaben syklomaattinen kompleksisuus (CC) [Mcc76], komponenttien välistä riippuvuutta mittaavat koheesio (cohesion) ja kytkeytyminen (coupling) sekä erilaiset oliopohjaisia ohjelmistoja varten suunnitellut mittarit. Sisäisillä mittareilla on mahdollista määrittää ohjelmiston laatua sekä tehdä ennusteita, miten ominaisuudet tulevat vaikuttamaan eri kokoisiin ohjelmistoihin sekä esimerkiksi ohjelmistoja toteuttavien työryhmien työtehoon.

Staattisista mittareista tässä tutkielmassa käytetään koodirivien fyysistä (LOC) ja loogista (LLOC) lukumäärää, syklomaattista kompleksisuutta sekä lisäksi osaa Halsteadin mittareista [Hal77], joilla saadaan toisenlainen näkökulma kokoon ja monimutkaisuuteen. Koheesion ja kytkeytymisen mittaaminen on jätetty tukimuksen ulkopuolelle, koska tutkittavan sovelluksen komponenttimäärä on suhteellisen pieni, näiden väliset yhteydet yksinkertaisia ja rajapintojen erot versioiden välillä hyvin pieniä. Koska JavaScript ei tue luokkapohjaista perintää, useat olio-ohjelmoinnin mittareista eivät ole sellaisenaan käyttökelpoisia. Yleisiä oliomittareita, kuten perintähierarkian syvyyttä ja painotettua metodien määrää luokassa (weighted methods per class), voitaisiin soveltaa esimerkiksi laskemalla metodien määrää moduulissa ja mittaamalla laajennettu moduuli -suunnittelumallin avulla täydennetyn moduulihierarkian pituutta. Koska tutkittavan esimerkkisovelluksen julkisten rajapintojen metodimäärä ja moduulihierarkia on kuitenkin identtinen eri versioiden välillä, ei mittareilla oltaisi saatu järkevää tutkimusaineistoa. Tämän takia sovellettuja oliometriikoita ei ole käytetty.

Tutkimuksessa tehtyjen mittausten suorittamiseen on käytetty escomplex-työkalua [Boo14], joka tukee tässä tutkielmassa käytettyjen mittareiden lisäksi monia muita mittareita. Escomplex on tutkielman kirjoitushetkellä kattavin ja aktiivisimmassa kehityksessä oleva JavaScript-koodin staattiseen mittaamiseen kehitetty työkalu. Koska esimerkkisovellus hyödyntää jonkin verran funktionaalisen ohjelmoinnin työkaluja, tutkimuksessa olisi voinut olla hyödyllistä käyttää myös jotain

funktionaaliseen ohjelmointiin suunnattua mittaria. Aiheeseen liittyviä tutkimuksia on olemassa [Ryd04], mutta suosittuihin imperatiivisiin metriikoihin, kuten syklomaattiseen kompleksisuuteen verrattavissa olevia, vakiintuneita mittareita ei ole yleisesti käytössä. Myöskään tutkimuksessa käytetystä escomplex-työkalusta ei löydy tukea tällaisille metriikoille. Tässä tutkimuksessa käytettyjen imperatiivisten metriikoiden ja muutosskenaarioiden avulla on kuitenkin pyritty saamaan kattavasti tuloksia merkityksellisten johtopäätösten tekemiseen.

Jotta mittaus olisi mahdollisimman vaivaton suorittaa, toteutettiin sille JavaScript-kielinen apuohjelma luvussa 5.1 mainitun Grunt-työkalun avulla. Suoritettaessa mittausta ohjelma yhdistää ensin mitattavan sovellusversion kaikki JavaScript-tiedostot yhdeksi uudeksi tiedostoksi, jotta mittaus on mahdollista suorittaa koko lähdekoodille. Yhdistämisen jälkeen ohjelma suorittaa escomplex-työkalulla mittauksen yhdistetylle tiedostolle ja jokaisen komponentin sisältävälle yksittäiselle tiedostolle. Ohjelmalle voidaan antaa kutsuparametriksi tieto mitattavasta versiosta: esimerkiksi RxJS-version mittaaminen tapahtuisi komentoriviltä komennolla *grunt complexity:rxjs*.

Koodirivien lukumäärä

Koodirivien lukumäärä on eräs yksinkertaisimmista lähdekoodista mitattavista ominaisuuksista. Kuten Fentonin ja Pfleegerinkin kirjassa mainitaan, pelkkä rivien lukumäärä ei ole sellaisenaan kovin hyvä sovelluksen koon tai monimutkaisuuden mittari. Tämä johtuu etupäässä siitä, että jotkut rivit ovat vaikeampia ymmärtää kuin toiset, ja toisaalta siitä, että esimerkiksi kommentit tai lauseiden jakaminen useammalle riville lisäävät rivien lukumäärää mutta toisaalta saattavat helpottaa koodin ymmärtämistä. Rivien lukumäärän mittaamisesta on kuitenkin hyötyä, kun tuloksia käytetään yhdessä muiden mittareiden avulla saatujen tulosten kanssa. Yleisimmät tavat mitata koodin pituutta on laskea fyysisten ja loogisten rivien määrä erikseen. Fyysisten rivien määrä saadaan suoraan laskemalla sovelluksen lähdekoodin kokonaisrivimäärä. Loogisten rivien määrä saadaan fyysisestä rivimäärästä, kun siitä poistetaan esimerkiksi tyhjät rivit ja kommentit sekä lasketaan useammalle riville jaetut lauseet yhdeksi riviksi.

Sekä fyysisten että loogisten koodirivien mittaaminen on erittäin hyödyllistä tässä työssä tutkittavan sovelluksen kohdalla, koska näillä saadaan hyvin eri kokoisia tuloksia: RxJS:ää tai Flapjaxia käytettäessä tietovuo-operaatioita on luontevaa ketjuttaa peräkkäin. Jotta ketjutetuista koodiriveistä ei tulisi kohtuuttoman pitkiä ja

ne olisivat helposti luettavissa, operaatiot ovat jaettu omille riveilleen. Tällöin yksi operaatioketju vastaa useaa fyysistä koodiriviä, mutta vain yhtä loogista koodiriviä. Operaatioketjujen lisäksi muita pitkiä lauseita, kuten esimerkiksi useasta vertailusta koostuvia if-lauseita tai JavaScript-olioiden luontia, on jaettu useammalle riville luettavuuden parantamiseksi.

Escomplex mittaa fyysisten koodirivien lukumäärän laskemalla suoraan kaikki lähdekoodissa olevat rivit lukuun ottamatta aivan lähdekooditiedoston alussa tai lopussa olevia tyhjiä rivejä tai kommentteja. Kaikki muut rivit lasketaan fyysisiksi riveiksi.

```
var inputStream = keyStream
	.throttle(500)
	.merge($searchField.focusAsObservable())
	.map(function(x) {
		return {
			text: $(x.target).val(),
			key: x.keyCode
		};
	}).filter(function(x) {
		return x.text.length > 1 &&
			x.key !== mashup.helpers.keyCodes.escape &&
			x.key !== mashup.helpers.keyCodes.enter;
	});
```

Yllä oleva lähdekoodi on katkelma esimerkkisovelluksen RxJS-versiosta. Koodissa muodostetaan alustava tapahtumavuo hakukenttään kohdistuvista näppäinten painalluksista. Katkelman tapahtumavuolle tehtävä operaatioketju on jaettu useammalle riville, jonka lisäksi nimettömien funktioiden paluuarvoiksi määritellyt olion luonti ja ehtolause on molemmat jaettu useammalle riville. Koodista escomplexilla mitattavien fyysisten rivien lukumäärä on 13 ja loogisten rivien lukumäärä viisi. Fyysisiksi riveiksi lasketaan tässä tapauksessa jokainen koodirivi, mutta loogisiksi koodiriveiksi ainoastaan sijoitus *inputStream*-muuttujaan, tapahtumavuota muokkaavien nimettömien funktioiden määrittelyt sekä funktioiden paluuarvot määrittelevät lauseet.

McCaben syklomaattinen kompleksisuus

McCaben syklomaattinen kompleksisuus on ehkä yleisin ja tunnetuin lähdekoodin monimutkaisuuden mittareista. McCabe esitti sen vuonna 1976 tutkimuksessaan A Complexity Measure [Mcc76], ja se on edelleen erittäin käytetty mittari. Syklomaattisen kompleksisuuden ideana on se, että ohjelman mahdollisten

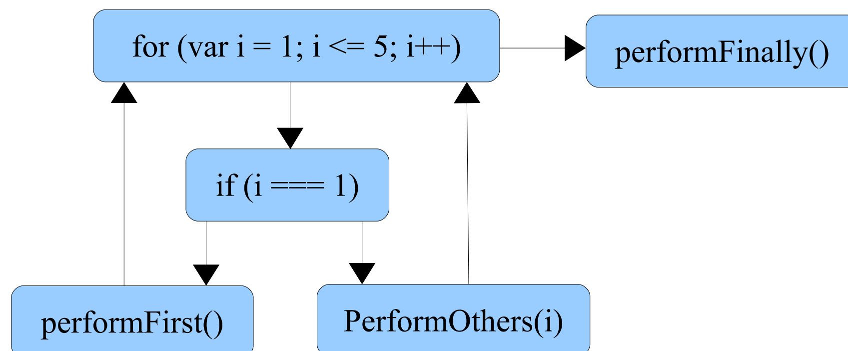
suorituspolkujen lukumäärä kertoo ohjelman monimutkaisuuden. Suorituspolkujen määrittelemisestä on myös hyötyä esimerkiksi yksikkötestien ohjelmoinnissa, koska poluista käy ilmi kaikki mahdolliset erilaiset testattavat tilanteet. Ohjelman syklomaattinen kompleksisuus on kokonaisluku, jonka arvo on yksi tai enemmän. Usein ajatellaan, että syklomaattisen kompleksisuuden ollessa viisi tai enemmän ohjelman osa alkaa olla monimutkainen, ja arvon ollessa enemmän kuin kymmenen ohjelmaa alkaa olla vaikea ymmärtää. Syklomaattinen kompleksisuus voidaan määrittää koko ohjelmalle tai yksittäisille funktioille. Mikäli yksittäisen funktion kompleksisuus kasvaa kovin suureksi, se voidaan pilkkoa pienempiin osiin. Tämä ei pienennä koko ohjelman syklomaattista kompleksisuutta, mutta yksinkertaistaa funktiota, jolloin koko ohjelman monimutkaisuus on helpommin hallittavissa.

Laskettaessa syklomaattista kompleksisuutta aluksi muodostetaan ohjausvuokaavio (control flow graph), joka on suunnattu verkko. Verkon muodostaminen tapahtuu käymällä läpi lähdekoodia. Peräkkäin suoritettavista lauseista muodostetaan suoritusverkkoon aina yksi solmu ja kaari edellisestä solmusta. Kun vastaan tulee lause, jossa ohjelman suoritus haarautuu, luodaan kaksi uutta solmua ja kaari näihin edellisestä solmusta. Lopullinen syklomaattinen kompleksisuus lasketaan kaavalla $M = E - N + 2P$, jossa E on verkon kaikkien kaarien lukumäärä, N verkon kaikkien solmujen lukumäärä ja P verkon yhtenäisten osien (connected component) lukumäärä. Kaavan mukaisesti M siis määrittää McCaben syklomaattisen kompleksisuuden.

```
for (var i = 1; i <= 5; i++) {
      if (i === 1) {
            performFirst();
      } else {
            performOthers(i);
      }
}
performFinally();
```

Yllä oleva koodilistaus muodostaa kuvan 7.1 mukaisen ohjausvuokaavion. For-silmukasta haarautuu kaksi kaarta, joihin päätyminen riippuu muuttujan *i*:n arvosta. Silmukan vertailun ollessa tosi jatketaan suoritusta if-lauseen sisältävään solmuun, josta voidaan päätyä kahteen eri solmuun ja takaisin for-solmuun. Silmukan vertailun ollessa epätosi, päädytään *performFinally*-funktiokutsun sisältävään solmuun ja lopetetaan suoritus. Vuokaaviosta voidaan laskea, että $E = 6$, $N = 5$ ja $P = 1$. Tällöin M – eli syklomaattinen kompleksisuus – on $6 - 5 + 2 * 1 = 3$. Joissain tapauksissa myös alku- ja lopputilaa kuvastavat solmut ja näiden kaaret lasketaan mukaan. Lopputulokseen ne eivät kuitenkaan vaikuta, koska molempia on aina vakiomäärä.

Kuvassa 7.1 alku- ja lopputilaa ei ole esitetty.



Kuva 7.1: Syklomaattisen kompleksisuuden muodostava ohjausvuokaavio

Syklomaattinen kompleksisuus on ensisijaisesti suunniteltu imperatiivisille ohjelmointikielille, koska ohjausvuo muodostetaan ehtolauseista ja silmukoista. Puhtaasti funktionaalisessa ohjelmoinnissa näitä rakenteita pyritään välttämään mahdollisimman pitkälle, josta johtuen syklomaattinen kompleksisuus lähestyy silloin aina yhtä. Funktionaalisten ohjelmien mittaamiselle ei kuitenkaan tällä hetkellä ole syklomaattista kompleksisuutta vastaavaa mittaria, joten syklomaattista kompleksisuutta on hyödynnetty tässä tutkimuksessa ensisijaisena monimutkaisuuden mittarina. Syklomaattinen kompleksisuus soveltuu tähän tutkimukseen myös siksi, että JavaScript ei ole puhtaasti funktionaalinen kieli, vaan yhdistelmä imperatiivista ja funktionaalista ohjelmointiparadigmaa.

```
[1, 2, 3, 4, 5].map(function(i) {
      if (i === 1) {
            performFirst();
      } else {
            performOthers(i);
      }
});
performFinally();
```

Aiemman koodiesimerkin for-silmukka voitaisiin korvata esimerkiksi funktionaalisessa ohjelmoinnissa käytetyllä *map*-funktiolla yllä olevalla tavalla. Tässä toteutuksessa ohjausvuokaavioon muodostuisi ainoastaan solmut lähtötilalle, *map*:in suoritukselle, if-lauseelle ja kolmelle funktiokutsulle, jolloin syklomaattinen kompleksisuus laskisi alkuperäisestä kolmesta kahteen. Koodin toiminnallisuus

voitaisiin myös toteuttaa reaktiivisesti RxJS-kirjaston avulla:

```
var values = Rx.Observable.range(1, 5);
values.first().subscribe(function() { performFirst(); });
values.skip(1).subscribe(function(x) { performOthers(x); });
performFinally();
```

Yllä olevassa koodissa koko ohjelman päätöslogiikka on toteutettu tapahtumavoiden avulla. *range*-metodilla luodaan luvuista yhdestä viiteen koostuva tapahtumavuo, jonka tapahtumia tilataan erikseen valitsemalla vuosta sopivat tapahtumat: ensimmäinen tapahtuma ja muut tapahtumat sen jälkeen. Kun tapahtumat saapuvat tilaajille, suoritetaan halutut funktiokutsut. Tässä tapauksessa syklomaattinen kompleksisuus laskee yhteen, koska koodissa ei silmukoita tai ehtolauseita, vaan päätöslogiikka on kapseloitu tapahtumavoihin. Voidaan kyseenalaistaa, onko tällä tavalla toteutettu ohjelma ratkaisevasti yksinkertaisempi kuin alkuperäinen silmukkaan ja ehtolauseihin perustuva imperatiivinen toteutus. Reaktiivisen version etuna on ainakin se, että siinä ei muokata muuttujien tilaa mitenkään, jolloin se on vähemmän riskialtis erilaisille virheille. Lisäksi versio on luonteeltaan deklaratiivinen, jolloin siitä on helppo lukea suoraan, mitä tehdään, sen sijaan, että ohjelman toimintaa luettaisiin toteutuksen kautta. Näistä syistä tässä tutkimuksessa on päädytty käyttämään syklomaattista kompleksisuutta monimutkaisuuden mittarina.

Halsteadin mittarit

Halsteadin mittarit ovat syklomaattisen kompleksisuuden ohella ensimmäisiä julkaistuja koodin sisäistä laatua määritteleviä mittareita. Mittareita on useampia, ja Halstead esitti ne kirjassaan Elements of Software Science [Hal77] vuonna 1977. Halsteadin mittarit perustuvat pohjimmiltaan koodista laskettavien operaattorien ja operandien määrään. Näiden lukumäärien avulla saadaan laskettua useita lähdekoodin sisäistä laatua määrittäviä mittareita. Halsteadin mittareita ovat pituus (length), sanavarasto (vocabulary), vaikeus (difficulty), tilavuus (volume), työmäärä (effort) ja virheet (bugs). Mittareista pituutta, sanavarastoa ja vaikeutta voidaan ajatella perusmittareina, koska ne ovat helposti laskettavissa suoraan operaattorien ja operandien määrästä. Perusmittareita hyödyntämällä saadaan laskettua tilavuus, työmäärä ja virheet. Pituus ja sanavarasto mittaavat lähdekoodin kokoa. Näiden avulla laskettava vaikeus puolestaan kuvastaa aikaa, joka vaaditaan lähdekoodin ymmärtämiseksi. Tilavuus arvioi vaadittua muistimäärää suoritettaessa ohjelmaa,

työmäärä koodimuutoksiin vaadittavaa aikaa ja virheet potentiaalisten virheiden esiintymistodennäköisyyttä. Fentonin ja Pfleegerin mukaan useat mittareista on kyseenalaistettu moneen kertaan, eikä näiden paikkansapitävyydestä ole empiiristä näyttöä. Fentonin ja Pfleegerin mukaan pituus ja sanavarasto ovat kuitenkin käyttökelpoisia mittareita, ja niitä on käytetty tässä tutkimuksessa tarjoamaan vaihtoehtoinen näkökulma koodirivien lukumäärästä ja syklomaattisesta kompleksisuudesta saatuihin tuloksiin.

Laskettaessa Halsteadin mittareita, ensin lasketaan lähdekoosta erilaisten operaattorien määrä n_1, erilaisten operandien määrä n_2, kaikkien operaattorien määrä N_1 ja kaikkien operandien määrä N_2. Halsteadin pituus saadaan laskemalla $N_1 + N_2$ ja sanavarasto laskemalla $n_1 + n_2$. Edellisessä luvussa esitellyn imperatiivisesti toteutetun koodiesimerkin tapauksessa operaattoreita ovat esimerkiksi *for*, *var* ja =. Esimerkin operandeja puolestaan ovat *i*, *0*, *5* ja *1*. Koodiesimerkin sanavarasto on 14 ja pituus 18. Esimerkistä johdetussa deklaratiivisessa RxJS-esimerkissä sanavarasto on 12 ja pituus 17.

Operaattorien lukumäärä on mahdollista laskea usealla tavalla: esimerkiksi lause *i*++ voidaan tulkita yhdeksi ++ operaattoriksi tai yhdistelmäksi *i* = *i* + *1*:n operaattoreita. Tässä tutkimuksessa käytetty escomplex-työkalu laskee ++ operaattorille annetun operandin ainoastaan yhteen kertaan.

7.2 Muutosskenaariot

Muutosskenaarioiksi on valittu sovellukseen toteutettavia ominaisuuksia, jotka edustavat mahdollisimman hyvin reaktiivisesta ohjelmoinnista hyötyviä tilanteita, kuten asynkronisia palvelupyyntöjä ja käyttöliittymäohjelmointia. Toteutettavien muutosten on haluttu vastaavan mahdollisimman hyvin tilanteita, joita voisi tulla vastaan työskenneltäessä oikeiden tuotantosovellusten parissa. Tällöin skenaarioista saatavia tuloksia on mahdollista yleistää myös suurempiin, oikeisiin ohjelmistoihin. Lisäksi skenaariot on valittu siten, että niiden avulla saadaan luvun alussa esitettyjä tutkimuksen toivottuja johtopäätöksiä mahdollisimman hyvin esiin. Muutosskenaarioiksi on valittu seuraavat:

- Kuvahaun järjestämisperusteen vaihtaminen käyttöliittymästä
- Palvelupyynnön uusiminen pyynnön epäonnistuessa

Kuvien järjestämisen vaihtaminen toteutetaan siten, että sovelluksen käyttöliittymään lisätään valintapainike, josta voi valita, haetaanko kuvat vanhimmasta uusimpaan vai

uusimmasta vanhimpaan. Kun kuvien järjestystä vaihdetaan, suoritetaan aina uusi palvelupyyntö kuvien hakemista varten. Haku itsessään tapahtuu lisäämällä järjestysperusteen määrittävä parametri Flickr-palvelupyyntöjen mukaan. Käyttöliittymä toteutetaan kahdella järjestystä kuvaavalla painikkeella, joista vain toinen voi olla kerrallaan aktiivisena. Painikkeet tyylitellään Bootstrapin avulla. Skenaarion tarkoituksena on tutkia, miten uuden käyttöliittymäelementin ja siihen liittyvien tapahtumien kuuntelemisen lisääminen onnistuu. Koska valintapainikkeella on myös aina voimassa jokin arvo, se on hyvä sovelluskohde reaktiivisen arvon käyttämiselle. Tällöin reaktiivisen ja ei-reaktiivisen toteutuksen välisten erojen lisäksi on mahdollista tutkia myös, onko tässä tapauksessa Flapjaxin reaktiivisten arvojen käyttämisestä etua RxJS:n tapahtumavoihin verrattuna.

Palvelupyynnön uudelleen yrittäminen toteutetaan siten, että minkä tahansa asynkronisen palvelupyynnön epäonnistuessa samaa pyyntöä yritetään kaksi kertaa uudelleen sekunnin välein. Käyttäjälle näytetään virheilmoitus vasta, kun kaikki palvelupyynnöt ovat epäonnistuneet. Skenaarion tarkoituksena on tutkia, minkälaisia etuja reaktiivisesta ohjelmoinnista saavutetaan, kun asynkronisille palvelupyynnöille tehdään yksittäistä pyyntöä monimutkaisempia toteutuksia. Uudelleen yrittäminen on erityisen hyödyllinen ominaisuus asynkronisten palvelupyyntöjen tapauksessa, koska yksittäiset pyynnöt saattavat epäonnistua esimerkiksi lyhytkestoisten verkko-ongelmien tai väliaikaisten kohderajapintojen ongelmien takia. Vaikka tässä työssä keskitytään ainoastaan reaktiivisen ohjelmoinnin hyödyntämiseen selainpuolella, tästä ominaisuudesta voitaisiin hyötyä suoraan myös palvelinpuolen reaktiivisessa ohjelmoinnissa.

7.3 Staattisen lähdekoodianalyysin tulokset

Lähdekoodin mittaus on suoritettu jokaisen toteutusversion kohdalla koko ohjelman lähdekoodille sekä yksittäisten komponenttien lähdekoodille. Tällöin ohjelman kokonaisarvojen lisäksi voidaan tarkastella kokonaisarvojen muodostavia osakokonaisuuksia. Yksittäisten komponenttien mittaustuloksiin on laskettu mukaan myös moduulien kaikille versioille yhteiset osat. Kaikissa mittaustuloksia esittävissä taulukoissa *LOC* määrittää fyysisten koodiriven määrän, *LLOC* loogisten koodirivien määrän, *CC* syklomaattisen kompleksisuuden, *H. length* Halsteadin pituuden ja *H. voc.* Halsteadin sanavaraston. Yleisenä kommenttina koodista voidaan sanoa, että RxJS- ja Flapjax-versiot ovat suhteellisen lähellä toisiaan, mutta JavaScript versiosta ne eroavat hyvin paljon.

	LOC	**LLOC**	**CC**	**H. length**	**H. voc.**
JavaScript	541	332	28	1977	334
RxJS	647	373	22	2516	382
Flapjax	652	384	27	2480	377

Taulukko 7.1: koko lähdekoodin mittaustulokset

Taulukossa 7.1 on esitetty koko lähdekoodille suoritettujen mittausten tulokset. Kuten luvussa 7.1 ennustettiin, reaktiivisten versioiden fyysinen rivimäärä on suhteellisen suuri johtuen tietovoiden ja reaktiivisten arvojen operaatioiden jakamisesta useammalle riville. Tuloksissa yllättävää on se, että reaktiivisten versioiden looginen rivimäärä ja Halsteadin arvot ovat jonkin verran JavaScript-versiota suurempia. Reaktiivista ohjelmointia esittelevien esimerkkien perusteella oltaisiin voitu olettaa arvojen pysyvän suurin piirtein samana tai vähenevän, koska esimerkiksi tapahtumavoiden käsittelyyn on tarjolla metodeita, joiden toteuttaminen pelkällä JavaScriptilla vaatii useita rivejä koodia.

RxJS-version syklomaattinen kompleksisuus on huomattavasti muita versioita pienempi. Flapjax-version suuri lukema johtuu etupäässä siitä, ettei kielessä ole yhtä monipuolisia operaatioita tietovoiden käsittelyyn kuin RxJS:ssä. Puuttuvien operaatioiden aiheuttamaa kompleksisuuden eroa esitellään tarkemmin myöhemmin tässä luvussa. Koko sovelluksen syklomaattisen kompleksisuuden määrittää käytännössä App-komponentin toteutus, koska kuten taulukoista 7.3 ja 7.4 on nähtävissä, ImageLoader- ja Flickr-komponenttien CC on kaikkien versioiden välillä identtinen. App-komponentin mittaustulokset ovat esitetty taulukossa 7.2. Komponentin tärkein syklomaattiseen kompleksisuuteen vaikuttava tekijä on kuvahaun käsittely. Kuvahaun koko toiminnallisuus on toteutettu kaikissa versioissa omana funktionaan, jolloin sen mittaaminen ja tulosten vertaileminen on helppoa. Funktion JavaScript-version syklomaattinen kompleksisuus on 6, RxJS-version 3 ja Flapjax-version 5. Verrattaessa näitä lukuja App-komponentin ja koko lähdekoodin tuloksiin voidaan todeta CC:n määrittyvän hyvin pitkälle juuri tästä funktiosta.

	LOC	LLOC	CC	H. length	H. voc.
JavaScript	206	130	12	704	136
RxJS	281	152	6	1151	182
Flapjax	295	170	11	1141	185

Taulukko 7.2: App-komponentin mittaustulokset

	LOC	LLOC	CC	H. length	H. voc.
JavaScript	4	4	2	21	13
RxJS	17	11	2	53	22
Flapjax	16	10	2	51	21

Taulukko 7.3: ImageLoader-komponentin mittaustulokset

	LOC	LLOC	CC	H. length	H. voc.
JavaScript	60	42	6	237	59
RxJS	78	54	6	297	64
Flapjax	70	48	6	273	62

Taulukko 7.4: Flickr-komponentin mittaustulokset

Tietovuo-operaatiot

Flapjax-version kompleksisuutta kasvattaa useiden RxJS:stä löytyvien tietovuo-operaatioiden, kuten *flatMap*-metodin (joka on alias metodille *selectMany*), ja esimerkiksi taulukosta tapahtumia muodostavan *fromArray*-metodin puuttuminen. flatMap-metodilla on mahdollista muodostaa useita tapahtumavoita sisältävistä tapahtumavoista yksi tapahtumavuo, jossa yksittäiset tapahtumat kulkevat peräkkäin. Tämä mahdollistaa monimutkaisten sisäkkäisten tapahtumavoiden näppärän käsittelemisen.

1) Javascript-versio kuvahaun vastauksen käsittelystä

```
flickr.searchPhotos(text, photosOnPage, page, function(data) {
      var pending = data.photos.photo.length;
      var errors = false;

      data.photos.photo.forEach(function(photo) {
            mashup.ImageLoader.load(photo.thumbURL, function() {
                  // Aseta pinni kartalle
                  pending--;
                  // Ovatko kaikki kuvat käsitelty?
                  // Piilota latauskuvake ja näytä mahdollinen virhe
            }, function() {
                  errors = true;
                  pending--;
                  // Ovatko kaikki kuvat käsitelty?
                  // Piilota latauskuvake ja näytä mahdollinen virhe
            });
      });
}, function() {
      // Näytä virheilmoitus
});
```

2) RxJS-versio kuvahaun vastauksen käsittelystä

```
flickrResponse
      .flatMap(function(x) {
            return Rx.Observable.fromArray(x.photo);
      })
      .flatMap(function(x) {
            return mashup.ImageLoader.load(x.thumbURL)
                  .map(function() { return x; })
                  .catch(Rx.Observable.return(null));
            });
      });
// Tilaa tapahtumavuo ja aseta pinnit kartalle
// Jos tapahtumavuon päättyessä vuossa ilmeni yksi tai useampi
// null-arvo, näytä virheilmoitus
```

3) Flapjax-versio kuvahaun vastauksen käsittelystä

```
flickrResponseE
      .mapE(function(x) {
            var pending = x.photos.length;
            var errors = false;

            x.photo.forEach(function(photo) {
                  mashup.ImageLoader.load(photo.thumbURL)
                        .mapE(function(ok) {
                              if (ok) {
                                    // Aseta pinni kartalle
                              } else {
                                    errors = true;
                              }
                              pending--;
                        });
                        // Ovatko kaikki kuvat käsitelty?
                        // Piilota latauskuvake ja näytä
                        // mahdollinen virhe
            });
      });
```

Edellä olevassa koodilistauksessa on sama toiminnallisuus kaikilla eri versioilla toteutettuna. Koodi ei ole täysin oikeaa sovellusta vastaava: osa riveistä on korvattu kommenteilla, jotta listaus saatiin pysymään kohtuullisen kokoisena ja sitä on helpompi seurata. JavaScript-versio on suoraviivainen imperatiivinen toteutus, jossa Flickr-palvelupyynnön onnistumisesta tai epäonnistumisesta ilmoitetaan takaisinkutsuilla. Kuvat iteroidaan läpi JavaScriptin *forEach*-funktiolla, jonka sijasta oltaisiin voitu käyttää perinteisempää for-silmukkaa. For-silmukan käyttäminen nostaisi syklomaattista kompleksisuutta vielä yhdellä. JavaScript-versiossa on pyritty käyttämään mahdollisimman paljon uudehkojen selainten natiivisti tukemia funktionaalisen ohjelmoinnin ominaisuuksia, kuten *forEach*- ja *map*-funktioita. Ominaisuuksia on käytetty, jotta mittaustulokset eivät vääristyisi funktionaalisen ohjelmoinnin hyödyntämisen takia, koska reaktiivisissa versioissa on jouduttu joka tapauksessa käyttämään tällaisia ominaisuuksia. JavaScript-versiossa pidetään lataamisen tilaa yllä kahdessa muuttujassa, koska latauskuvake piilotetaan ja mahdollinen virheilmoitus näytetään vasta, kun kaikki kuvat ovat käsitelty. p*ending*-muuttujassa pidetään kirjaa siitä, montako kuvaa on vielä ladattavana, ja *errors*-muuttujassa tietoa siitä, onko yksikin kuvalatauksista epäonnistunut.

RxJS-versio on kaikista versioista selvästi kompaktein ja elegantein. Flickr-rajapinnalta vastauksena saatavan tapahtumavuon sisältämästä kuvatietojen taulukosta muodostetaan ensin uusi vuo *Rx.Observable.fromArray*-metodin avulla. Metodin suorittamisen jälkeen päädytään tilanteeseen, jossa on sisäkkäisiä tapahtumavoita: alkuperäinen, Flickr-palvelupyynnön vastaukset sisältävä tapahtumavuo pitää sisällään yksittäisistä kuvatiedoista koostuvan tapahtumavuon. Jotta kuvia voitaisiin käsitellä yksittäisinä tapahtumina vuossa, sisäkkäiset tapahtumavuot "litistetään" yhdeksi vuoksi *flatMap*-metodin avulla, jolloin jäljelle jää vain yksi tapahtumavuo, jonka tapahtumia ovat yksittäiset kuvatiedot. flatMapia on hyödynnetty myös kuvatiedostojen latauksessa: kun ensimmäisestä flatMapista saapuu kuvien tietoja yksittäisinä tapahtumia, täytyy jokaiselle näistä suorittaa uuden tapahtumavuon palauttava kuvatiedoston esilataus. Esilataus palauttaa uuden tapahtumavuon, jolloin päädytään jälleen tilanteeseen, jossa alkuperäinen tapahtumavuo sisältää useita tapahtumavoita. Myös nämä ovat mahdollista muuntaa flatMapilla yhdeksi tapahtumavuoksi, jolloin lopputuloksena on kuvatiedoista koostuva vuo, jonka kuvatiedostoille on suoritettu esilataus. Epäonnistuneen kuvatiedoston lataamisen tapauksessa RxJS-version ImageLoader-moduuli ilmoittaa tästä virheenä tapahtumavuossa. Virhe otetaan kiinni *catch*-metodilla, ja palautetaan uusi, yhdestä null-arvosta koostuva tapahtumavuo. Lopullisessa tapahtumavuossa

null-arvot siis merkkaavat kuvia, joiden kuvatiedostojen esilataus on epäonnistunut. Tällaisia kuvia ei haluta lisätä kartalle, joten ne voidaan suodattaa pois vuosta *filter*-metodin avulla.

Flapjax-version toteutus on jossain määrin yhdistelmä JavaScript-versiota ja RxJS-versiota. RxJS-version tavoin Flickr-rajapinnasta saatava tieto ilmoitetaan tapahtumina tapahtumavuossa. Koska Flapjaxissa ei RxJS:n tavoin ole mekanismia ilmoittaa tapahtumavuon päättymisestä, joudutaan ladattavien kuvien tilaa ylläpitämään JavaScript-version tavoin *pending*-muuttujassa. Flapjaxissa ei myöskään ole RxJS:ää vastaavaa virheidenkäsittelyjärjestelmää, joten epäonnistuneet kuvatiedostojen esilataukset ilmoitetaan palauttamalla ImageLoader:ista totuusarvo. Puutteellisista tapahtumavuo-operaatioista johtuen Flapjax-koodi muistuttaa palvelupyyntöjen käsittelyä lukuun ottamatta JavaScript-versiota. flatMap ja fromArray-metodien lisääminen Flapjax-kirjastoon onnistuisi suhteellisen helposti, ja se kannattaisikin tehdä, mikäli Flapjaxia käytettäisiin tämän tutkimuksen ulkopuolella. Metodien toteuttamisella ja hyödyntämisellä saataisiin aikaan RxJS-versiota vastaava syklomaattinen kompleksisuus. Tähän tutkimukseen näitä tietovuo-operaatioita ei ole toteutettu, koska metodien puuttumisen vaikutuksen analysoinnista saatiin hyödyllisiä tutkimustuloksia.

Sovelluksen tilan käsittely

Syklomaattiseen kompleksisuuteen vaikuttaa olennaisesti koko sovelluksen tilan käsittely: JavaScript-versiossa tilaa ylläpidetään erillisissä muuttujissa, kun taas RxJS-versiossa ei ole yhtään tilaa ylläpitävää muuttujaa, vaan ohjelman tila on kapseloitu tapahtumavoihin. Kuten aiemmin todettiin, Flapjaxissa ei ole RxJS:ään verrattuna yhtä monipuolisia tietovuo-operaatioita, joten Flapjax-versiossa on jouduttu joissain tilanteissa turvautumaan paikallisiin muuttujiin.

1) JavaScript-version toteutus kuvahaun suodattamisesta

```
if ((text !== lastSearchText ||
      page !== currentPage ||
      photosOnPage !== currentPhotosOnPage) &&
      text !== undefined && text.length > 0) {
      // Suorita kuvien haku Flickr-rajapinnasta ja käsittele vastaus
}
```

2) RxJS-version toteutus kuvahaun suodattamisesta

```
searchStream.filter(function(x) {
      return x.text !== undefined && x.text.length > 1;
})
.distinctUntilChanged(function(x) { return x; }, compareSearch)
.subscribe(function(x) {
```

```
        searchPhotos(x.text, x.page, x.photosOnPage);
});
```

3) Flapjax-version toteutus kuvahaun suodattamisesta

```
searchB
        .changes()
        .filterE(function(x) {
              return x.text !== undefined && x.text.length > 1;
        })
        .mapE(function(x) {
              searchPhotos(x.text, x.page, x.photosOnPage);
        });
```

Edellä olevassa koodilistauksessa on kaikkien eri versioiden toteutus hakujen rajaamisesta toiminnallisten vaatimusten mukaisiksi. JavaScript-version tilaa ylläpidetään muuttujissa *lastSuggestionText*, *lastSearchText*, *currentPage*, *maxPages* ja *currentPhotosOnPage*. Muuttujat kertovat viimeisen tunnistehaun, viimeisen kuvahaun, parhaillaan näytettävän kuvakokoelmasivun numeron, sivujen maksimimäärän ja kartalla näytettävien pinnien lukumäärän. Osa arvoista voitaisiin lukea aina suoraan niitä vastaavista käyttöliittymäkomponenteista, mutta esimerkiksi viimeistä tunnistehakua ja kuvahakua ei ole mahdollista lukea suoraan hakukentästä. Tästä syystä myös muut arvot ovat sijoitettu omiin muuttujiinsa koodin yhdenmukaistamiseksi. JavaScript-version syklomaattinen kompleksisuus muodostuu ennen kaikkea muuttujien käyttämisestä ehtolauseissa. Muuttujien vertailu voitaisiin erottaa omaan funktioonsa, jolloin ehtolause näyttäisi hieman siistimmältä, mutta se ei vaikuttaisi kuitenkaan kompleksisuuteen.

RxJS ja Flapjax-versioiden toteutukset ovat suhteellisen lähellä toisiaan. Molemmat perustuvat siihen, että yhdessä tapahtumavuossa säilytetään hakutapahtumia, jotka ovat hakuparametrit sisältäviä JavaScript-olioita. Hakutapahtuma sisältää aina tiedon hakutekstistä, sivun numerosta ja sivulla näytettävästä kuvien määrästä. Koska minkä tahansa näistä muuttaminen aiheuttaa uuden kuvahaun, voidaan suodattaminen toteuttaa aina vertaamalla uutta tapahtumaa edelliseen. RxJS-toteutuksessa *searchStream*-tapahtumavuo sisältää hakutiedot sisältävät tapahtumat. Tapahtumavuosta suodatetaan ensin *filter*-metodin avulla pois liian lyhyet haut, jonka jälkeen *distinctUntilChanged*-metodilla suodatetaan pois kaikki tapahtumat, joiden sisältö ei eroa edellisestä tapahtumasta. Lopuksi jäljelle jääneet tapahtumat tilataan *subscribe*-metodilla ja kutsutaan *searchPhotos*-funktiota, joka suorittaa Flickr-palvelupyynnön ja käsittelee vastauksen.

Flapjax-versiossa on käytetty tapahtumavoiden lisäksi reaktiivisia arvoja. s*earchB* on reaktiivinen muuttuja, jonka arvo on aina hakuparametriolio, kuten RxJS-version

tapahtumavoissakin. *change*-metodin avulla reaktiivinen arvo voidaan muuntaa tapahtumavuoksi, johon ilmestyy tapahtuma aina, kun reaktiivinen arvo muuttuu. Tapahtumavuon käsittely tapahtuu Flapjax-versiossa käytännössä aivan kuten RxJS-versiossakin – ainoastaan käytettyjen metodien nimet ovat hieman erilaiset. Tässä tapauksessa reaktiivisen arvon käyttäminen verrattuna tapahtumavoihin ei vaikuta koodin kokoon tai kompleksisuuteen.

1) RxJS-version tapahtumavoiden yhdistäminen

```
var searchStream = Rx.Observable.return()
      .combineLatest(textSubject, countSubject, pageStream,
            function(_, text, photoCount, page) {
                  return {
                        text: text,
                        page: page,
                        photosOnPage: photoCount
                  };
            });
```

2) Flapjax-version reaktiivisten arvojen yhdistäminen

```
var searchB = F.liftB(function(text, count, page) {
      return {
            text: text,
            page: page,
            photosOnPage: count
      };
}, textB, countB, pageB);
```

Yllä on listattu RxJS-versiossa tapahtuvan hakuparametrit sisältävän tapahtumavuon muodostaminen ja Flapjax-versiossa tapahtuvan vastaavan reaktiivisen arvon muodostaminen. RxJS-version toteutus koostuu tapahtumavoista *textSubject*, *countSubject* ja *pageStream*, jotka yhdistetään uudeksi vuoksi *combineLatest*-metodin avulla. c*ombineLatest* pitää kirjaa sille parametreina annettujen tapahtumavoiden viimeisistä arvoista, ja funktio suoritetaan aina, kun jokin tapahtumavoista muuttuu. Tällöin suoritettaessa funktiolla on käytössään aina muuttuneen tapahtumavuon uusi arvo ja muiden tapahtumavoiden edellinen arvo.

Flapjax-versiossa käytetään reaktiivisia arvoja kuvaamaan sekä hakuja *searchB*-muuttujassa että yksittäisiä käyttöliittymän arvoja *textB*-, *countB*- ja *pageB*-muuttujissa. Näiden välille muodostetaan riippuvuus *liftB*-metodilla, jonka jälkeen *searchB*-muuttuja päivittyy aina, kun jokin käyttöliittymältä saaduista muuttujista päivittyy. Molempien toteutusten syklomaattinen kompleksisuus on yksi, rivimäärä suurin piirtein sama ja RxJS:n Halsteadin arvot jonkin verran suuremmat. Halsteadin arvojen ilmoittaman tuloksen ohella myös tarkasteltaessa koodia silmämääräisesti Flapjax-versio näyttää jonkin verran selkeämmältä. Vaikka reaktiivisten arvojen

käyttäminen verrattuna pelkkiin tietovoihin ei pienennä ratkaisevasti syklomaattista kompleksisuutta tai koodirivien määrää, selkeyttävät ne ainakin omasta mielestäni lähdekoodia. Flapjax-version tapauksessa olisi myös mahdollista hyödyntää implisiittistä nostamista, jota käsitellään seuraavaksi.

Implisiittinen nostaminen

Flapjaxin implisiittistä nostamista hyödyntäviä tilanteita on niin vähän, että vaikka Flapjax-versio olisi toteutettu kirjaston sijaan Flapjax-kielellä, ei tällä olisi ollut vaikutusta mittaustuloksiin. Nostamisen mahdollistavaa *liftB*-metodia on käytetty ainoastaan aiemmin esitellyssä, hakutekstin kuvamäärän ja sivunumeron yhdistävässä operaatiossa.

```
/* Eksplisiittisesti nostettu versio */
var searchB = F.liftB(function(text, count, page) {
      return {
            text: text,
            page: page,
            photosOnPage: count
      };
}, textB, countB, pageB);

/* Vastaava implisiittisesti nostettu versio */
var search = { text: textB, page: pageB, photosOnPage: countB };
```

Yllä oleva listaus esittää sovelluksen Flapjax-versiossa käytetyn, eksplisiittisesti nostetun lähdekoodin, jossa *liftB*-metodin avulla luodaan riippuvuus reaktiivisen muuttujan *searchB* sekä reaktiivisten muuttujien *textB*, *countB* ja *pageB* välille. Kuten listauksesta näkyy, tässä tapauksessa implisiittisellä nostamisella olisi säästetty ainoastaan yksi metodikutsu ja nimetön funktio. Koska muuttuvan koodin määrä on vähäistä, implisiittisellä nostamisella ei ole tässä sovelluksessa käytännössä mitään vaikutusta rivimääriin tai Halsteadin arvoihin. Yhden metodikutsun ja nimettömän funktion poistaminen ei myöskään vaikuta mitenkään syklomaattiseen kompleksisuuteen. Implisiittinen nostaminen saattaisi olla hyödyllinen ja koodia selkeyttävä ominaisuus sovelluksissa, joissa yhdistellään suuria määriä reaktiivisia arvoja. Tämän tutkimuksen tapaisissa sovelluksissa, joissa nostamista tarvitaan vain vähän, implisiittisestä nostamisesta ei saavuteta juurikaan etua.

7.4 Muutosskenaarioiden tulokset

Muutosskenaarioiden avulla on pyritty etenkin saamaan tietoa siitä, miten

reaktiivinen ohjelmointi vaikuttaa sovellusten koon kasvaessa. Koska sovelluksen alkuperäinen rivimäärä ja Halsteadin arvot olivat reaktiivisissa versioissa JavaScript-versiota suurempia, on hyödyllistä tutkia, vaatiiko uusien ominaisuuksien toteuttaminen edelleen enemmän koodia. Lähdekoodimuutosten esittelemisen lisäksi skenaarioiden lähdekoodia on mitattu alkuperäiseen sovellukseen käytetyillä mittareilla.

Kuvien järjestämisperusteen vaihtaminen käyttöliittymästä

Kaikkien versioiden Flickr-moduulin *searchPhotos*-funktioon on lisätty uusi kutsuparametri *sort*, joka määrittää Flickr-rajapinnasta haettavien kuvien järjestämisperusteen. Esimerkiksi JavaScript-version funktiokutsu on muotoa *searchPhotos(tag, perPage, page, sort, success, error).* Kaikille versioille yhteiseen HTML-tiedostoon lisättiin kaksi button-tyyppistä painiketta, joille asetettiin Bootstrapin tyylimäärittelyt. Painikkeista toisen halutaan olevan valittuna kerrallaan. Tämä on toteutettu käyttöliittymässä asettamalla aktiivisuutta esittävä tyylimäärittely valitulle painikkeelle. Tyylimäärittely muuttaa painikkeen ulkoasua siten, että painike näyttää olevan painettuna pohjaan. Painikkeiden ulkoasun muuttamista varten kaikkien versioiden App-moduuliin toteutettiin täysin samasta koodista koostuva, ulkoasun asettava funktio *setSortButton(sort).* Funktiolle annetaan kutsuparametrina merkkijono *'new'* tai *'old'* riippuen, kumman painikkeista halutaan olevan pohjassa. JavaScript-versiossa funktiota kutsutaan heti alustettaessa moduulia, jolloin oletusarvoksi valitaan *new*. Kaikille versioille asetetaan moduulin alustusvaiheessa myös muuttujat *$sortNewButton* ja *$sortOldButton*, jotka ovat jQuery-olioita ja tarjoavat näppärän tavan molempien HTML-elementtien käsittelyyn.

Liitteessä 4 esitetyn koodilistauksen alussa on esitetty JavaScript-version keskeisimpiä muutoksia. Tapahtumankäsittelijät painikkeiden klikkauksille asetetaan jQuery-olioiden *click*-metodin avulla. Tämä tapahtuu heti alustettaessa App-moduulia. Painiketta klikattaessa asetetaan käyttöliittymän tila ja kutsutaan *searchPhotos*-funktiota, joka käsittelee ja suorittaa haun. Suoritettaessa *searchPhotos*-funktiota, luetaan painikevalinnan arvo paikalliseen *sort*-muuttujaan, jota hyödynnetään myöhemmin if-lauseessa tutkittaessa, eroaako haku edellisestä hausta. Mikäli haku suoritetaan, asetetaan *sort*-muuttujan arvo lopuksi *currentSort*-muuttujaan, jossa säilytetään tietoa viimeisen suoritetun haun järjestämisperusteesta.

RxJS-version App-moduulin alustamisvaiheessa luodaan järjestämisperusteelle *Rx.Observable.BehaviorSubject*-tyyppinen tapahtumavuo. RxJS:n erilaiset *Subject*-

tapahtumavuot mahdollistavat tapahtumien tilaamisen lisäksi myös niiden työntämisen vuohon. Lause määrittelee vuolle yhden *'new'*-merkkijonon sisältävän tapahtuman, joka on järjestämisperusteen oletusarvo. RxJS-versiossa painikkeiden tapahtumien kuunteluun käytetään Javascript-version tavoin jQueryn *click*-metodia. Sen sijaan, että asetettaisiin käyttöliittymän tila ja kutsuttaisiin hakufunktiota, RxJS-versiossa ainoastaan työnnetään uusia tapahtumia vuohon. Käyttöliittymän tilan asettaminen tapahtuu tilaamalla tapahtumavuo ja asettamalla painikkeiden tila aina, kun tilaajalle ilmestyy tapahtumia. Järjestämisperusteen lisääminen viimeisen haun tilaan tapahtuu lisäämällä se *searchStream*:in muodostavaan *combineLatest*-funktioon. Flapjax-version toteutus on erittäin lähellä RxJS-versiota. Flapjax-versiossa hyödynnetään pelkkien tapahtumavoiden lisäksi myös reaktiivisia arvoja, mutta toteutuksen toimintalogiikka on tästä huolimatta molempien versioiden välillä lähes täysin sama.

Kuvien järjestäminen vaikuttaa käytännössä ainoastaan App-moduuliin, koska Flickr-moduulin muutokset ovat pieniä ja kaikkien versioiden välillä samat. Eri versioiden toteutukset vaativat loogisia rivimääriä seuraavasti: JavaScript 20 riviä, RxJS 17 riviä ja Flapjax 19 riviä. Lisätty koodimäärä on siis kaikille versioille käytännössä sama. JavaScript-version muutokset kasvattavat App-moduulin syklomaattista kompleksisuutta neljällä, jolloin lopullinen arvo on 16. RxJS ja Flapjax-versioiden arvo kasvoi muutoksista ainoastaan kahdella. Mikäli JavaScript-versioon tehtäisiin vielä toinen samantapainen toiminto samanlaisella toteutuksella, JavaScript-version syklomaattinen kompleksisuus kasvaisi edelleen neljällä ja reaktiivisten versioiden kahdella. Tästä voitaisiin päätellä, että hyödynnettäessä reaktiivista ohjelmointia, sovelluksen kompleksisuus kasvaa ei-reaktiivisesti ohjelmoitua versiota hitaammin.

Palvelupyyntöjen uudelleen yrittäminen

Palvelupyyntöjen uudelleen yrittäminen virhetilanteessa toteutettiin Flickr-moduulin *getRelatedTags*-metodille, joka hakee asynkronisesti Flickr API:sta parametrina annettuun tunnisteeseen liittyviä tunnisteita. Kaikki muutokset koskevat ainoastaan *getRelatedTags*-metodia. Toiminnallisuuden toteuttaminen vaati JavaScriptilla 10 uutta loogista koodiriviä, RxJS:llä 9 koodiriviä ja Flapjaxilla 17 koodiriviä. Metodin toteutus kaikilla toteutustavoilla on esitetty liitteessä 5.

JavaScript-version toteutus on melko suoraviivainen: palvelupyynnön suorittaminen on sijoitettu *processRequest()*-funktioon ja epäonnistuneen palvelupyynnön käsittely *processError*-nimiseen funktioon. Uudelleenyritysten lukumäärää ylläpidetään

funktion sisäisessä muuttujassa *tries*. *processError*-funktiossa vähennetään yritysten lukumäärää ja kutsutaan viiveellä uudestaan palvelupyynnön suorittavaa funktiota. Mikäli uudelleenyrityksiä ei ole enää jäljellä, suoritetaan virheestä ilmoittava takaisinkutsu.

RxJS tarjoaa *retry*-nimisen tapahtumavuo-operaation, jonka avulla tapahtumaa voidaan yrittää uudelleen, mikäli tapahtumavuossa ilmeni virhe. Mikäli toiminnallisuudelle ei olisi ollut vaatimuksena sekunnin viivettä yritysten välillä, RxJS-version toteuttaminen olisi onnistunut lisäämällä *getRelatedTags*:in palauttavalle tapahtumavuolle ainoastaan *retry(2)*-metodikutsu ennen palautusta. Uudelleen yrittäminen viiveellä ei kuitenkaan onnistu aivan näin suoraviivaisesti, vaan apuna on käytetty muutamaa RxJS:stä löytyvää tietovuo-operaatiota. *Rx.Observable.defer*-metodille annetaan parametriksi funktio, joka vastaa tapahtumavuon luomisesta. Funktio suoritetaan aina, kun tapahtumavuo tilataan. Tässä tapauksessa *retry*-metodi aiheuttaa uudelleen tilaamisen, joten funktio suoritetaan aina uudelleen, kun Flickr-rajapinnasta yritetään saada tietoa. Yritysten lukumäärästä pidetään kirjaa *first*-muuttujassa. Ensimmäisellä tilauskerralla kutsutaan *getRequestStream*-funktiota suoraan. Tämän jälkeisillä kerroilla luodaan ensin *Rx.Observable.timer*:in avulla uusi tapahtumavuo, johon lähetetään automaattisesti tapahtuma yhden sekunnin päästä. Vasta tapahtuman saapuessa kutsutaan *getRequestStream*-funktiota. Tällä toteutuksella yritysten välille saadaan aikaan viive. g*etRequestStream*-funktio palauttaa vuon, joka sisältää Flickr-palvelupyynnön tuloksen tapahtumana. Tästä seuraa sisäkkäinen tapahtumavuo, josta päästään eroon käyttämällä flatMap:ia. RxJS-version suurimpana vahvuutena on se, että toiminnallisuus voitaisiin helposti kapseloida omaksi, esimerkiksi *retryWithDelay*-nimiseksi tietovuo-operaatiokseen. Tällöin alkuperäistä toteutusta voitaisiin kutsua käskyllä *getRequestStream().retryWithDelay(2, 1000)*. Toteutus mahdollistaisi toiminnallisuuden erittäin hyvän uudelleenkäytettävyyden.

Flapjax-version koko on muita huomattavasti suurempi sen takia, että kirjasto ei tarjoa juurikaan toiminnallisuuden toteuttamista helpottavia operaatioita. JavaScript-version tavoin uudelleenyrityksistä pidetään lukua *tries*-muuttujassa. Flapjax-version toteutus perustuu pitkälti *response* ja *requests* -nimisiin tapahtumavoihin. r*esponse*-tapahtumavuo palautetaan *getRelatedTags*-metodin kutsujalle, ja tapahtumavuohon työnnetään metodikutsun lopullinen vastaus. *request*-vuohon lähetetään *getRequestStream*-funktion palauttamia tapahtumia. Flapjax-version samanniminen funktio toimii RxJS-versiota vastaavasti, paitsi että virheiden sijaan epäonnistuneista

palvelupyynnöistä ilmoitetaan null-arvona. Sisäkkäiset tapahtumavuot käsitellään vastaavasti kuin RxJS-versiossakin; ainoastaan puuttuvan flatMap-metodin sijaan käytetään *switchE*-metodia. Tapahtuman ilmestyessä *requests*-tapahtumavuohon vähennetään ensin yrityskertojen lukumäärää. Mikäli tapahtuma sisälsi onnistuneesta latauksesta saatua tietoa, lähetetään *response*-tapahtumavuohon tämä tieto. Jos palvelupyyntö oli epäonnistunut mutta yrityskertoja vielä jäljellä, suoritetaan ensin viive *F.timerE*-funktion avulla, joka vastaa hyvin lähelle RxJS-versiossa käytettyä *Rx.Observable.timer*-funktiota. Viiveen jälkeen lähetetään *requests*-tapahtumavuohon uusi palvelupyynnön sisältävä tapahtuma. Mikäli kaikki uudelleenyritykset oli suoritettu, lähetetään *response*-tapahtumavuohon epäonnistumisesta kertova *null*-arvo. Myös Flapjaxin tapauksessa toiminnallisuus voitaisiin erottaa omaksi tietovuo-operaatiokseen, jolloin sitä olisi helppo käyttää uudelleen muualla.

JavaScript ja RxJS-versioiden syklomaattinen kompleksisuus kasvoi kahdesta kolmeen ja Flapjax version neljään. Vaikka numeeriset tulokset kaikille versioille ovat suurin piirtein samoja, reaktiivisten versioiden selvänä etuna on kuitenkin parempi modulaarisuus. Tästä voitaisiin päätellä, että suuremmissa sovelluksissa monimutkaisuutta olisi helpompi hallita reaktiivisen ohjelmoinnin avulla.

7.5 Johtopäätökset

Reaktiivisten versioiden koko oli hieman JavaScript-versiota suurempi, mutta muutosskenaarioista saatujen tulosten avulla voitaisiin päätellä, että ohjelmiston koon kasvaessa ero tasoittuu. RxJS:n syklomaattinen kompleksisuus oli JavaScript-versiota ratkaisevasti pienempi. Muutosskenaarioista saadut tulokset myös kertovat, että reaktiivista ohjelmointia hyödyntämällä ohjelman kompleksisuus saattaa kasvaa ei-reaktiivista versiota huomattavasti hitaammin. Tästä voidaan todeta, että reaktiivisen ohjelmoinnin avulla saadaan hyvin hallittua lähdekoodin kompleksisuutta.

Tässä tutkielmassa käsitellyn esimerkkisovelluksen tapaisissa, pienehkössä sovelluksissa ohjelmiston kokoon vaikuttaa paljon, kuinka kattavasti reaktiivisessa kielessä tai kirjastossa on toteutettuna reaktiivisten arvojen ja tapahtumavoiden käsittelyyn tarkoitettuja operaatioita. Flapjax-toteutusversiosta saadut mittaustulokset vääristyivät puutteellisten operaatioiden takia huomattavasti. Eräänä reaktiivisen ohjelmoinnin etuna on myös operaatioiden tehokas kapselointi, jolloin niitä on helppo käyttää uudelleen.

Tutkimuksessa havaittiin, että sovelluksen toteuttaminen onnistui yhtä helposti pelkkien tapahtumavoiden avulla verrattuna reaktiivisten arvojen hyödyntämiseen tapahtumavoiden lisäksi. Reaktiivisten arvojen käyttämisestä saattaa olla kuitenkin apua lähdekoodin selkeyttämisessä lukijalle. Implisiittisestä nostamisesta ei ollut juuri hyötyä tämän kaltaisessa sovelluksessa. Siitä voitaisiin hyötyä jossain erikoistilanteissa, joissa on käytössä hyvin suuria määriä reaktiivisia arvoja.

8 Yhteenveto

Tässä tutkielmassa esiteltiin reaktiivisen ohjelmoinnin ja reaktiivisten ohjelmointikielten terminologiaa ja keskeisimpiä ominaisuuksia. Lisäksi esiteltiin web-ohjelmointia yleisesti, keskittyen selainpuolen ohjelmointiin. Selainpuolen ohjelmoinnin yhteydessä esitettiin tapoja hyödyntää reaktiivista ohjelmointia ja tutustuttiin reaktiiviseen RxJS-kirjastoon ja reaktiiviseen Flapjax-ohjelmointikieleen, joista edellinen on peräisin ohjelmistoteollisuudesta ja jälkimmäinen akateemisesta maailmasta.

Tutkielmaa varten toteutettiin Flickr & Google Maps Mashup -sovellus, jonka avulla tutkittiin reaktiivisen ohjelmoinnin vaikutusta selainpuolen lähdekoodin ominaisuuksiin. Saman toiminnallisuuden toteuttava sovellus toteutettiin pelkällä JavaScriptilla sekä RxJS ja Flapjax-kirjastojen avulla. Reaktiivisen ohjelmoinnin vaikutusta koodin ominaisuuksiin tutkittiin staattisen analyysin menetelmin mittaamalla lähdekoodin kokoa ja kompleksisuutta sekä muutosskenaarioiden avulla, joilla tutkittiin, miten jonkin toiminnallisen muutoksen tekeminen onnistuu eri versioiden kohdalla.

Analyysin johtopäätökseksi saatiin, että reaktiivinen ohjelmointi ei välttämättä pienennä lähdekoodin kokoa, mutta koodin yksinkertaistamiseen se tarjoaa hyvät mahdollisuudet. FRP-tyyppisten reaktiivisten kielten tarjoama reaktiivisten arvojen abstraktio on hyödyllinen, muttei välttämätön. Tämän sovelluksen puitteissa implisiittisellä nostamisella ei saavutettu juurikaan etuja eksplisiittiseen nostamiseen verrattuna. Lisäksi suuri vaikutus oli siinä, minkä verran kielestä tai kirjastosta löytyy tukea erilaisille operaatioille reaktiivisten arvojen tai tapahtumavoiden muokkaamista varten.

Koska reaktiivinen ohjelmointi eroaa melko paljon perinteisestä takaisinkutsuihin perustuvasta ohjelmointitavasta, sen hyödyntäminen tehokkaasti vaatii ohjelmoijalta jonkin verran kokemusta paradigman käytöstä. Mielenkiintoisena jatkotutkimus-aiheena olisi selvittää, kuinka luettavaksi RP:tä vain vähän tuntevat ohjelmoijat kokevat reaktiivisesti ohjelmoidun lähdekoodin ja kuinka kauan aikaa vaaditaan, että ohjelmointitapa alkaa tuntua luontevalta. Myös reaktiivisen ohjelmoinnin kompleksisuuden mittarien kehittäminen olisi mielenkiintoinen tutkimusaihe, koska kuten tutkimuksessa tuotiin esille, perinteiset ohjelman rakenteen mittarit eivät sovellu sellaisenaan kovin hyvin reaktiivisella ohjelmoinnilla toteutettujen ohjelmistojen mittaamiseen.

Lähteet

Ash14 Ashkenas, J. et al., CoffeeScript, http://coffeescript.org/, [13.2.2014].

App14 Safari Web browser, Apple Inc., https://www.apple.com/safari/, [16.7.2014].

Bai12 Bainomugisha, E. et al., A Survey on Reactive Programming. ACM Computing Surveys, 2012.

BeG92 Berry, G., Gonthier, G., The ESTEREL synchronous programming language: design, semantics, implementation, Science of Computer Programming, volume 19, issue 2, marraskuu 1992, sivut 87-152.

Boo14 Booth, P., escomplex, https://github.com/philbooth/escomplex, [17.7.2014].

Che10 Cherry, B., JavaScript Module Pattern: In-Depth, http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html, 2010, [17.8.2014].

ChH13 Christiensen, B., Husain, J., Functional Reactive in the Netflix with RxJava, The Netflix Tech Blog, Netflix, Inc., http://techblog.netflix.com/2013/02/rxjava-netflix-api.html, 2013, [15.1.2014].

Cro01 Crockford, D., JavaScript: The World's Most Misunderstood Programming Language, http://www.crockford.com/javascript/javascript.html, [29.3.2014].

Cro02 Crockford, D., JSLint, http://www.jslint.com/, [29.3.2014].

Cro08 Crockford, D., JavaScript: The Good Parts, O'Reilly Media, Inc., 2008.

Eco14 Scala Programming Language, École Polytechnique Fédérale de Lausanne, 2012, http://www.scala-lang.org, [23.1.2014].

ElH97 Elliott, C., Hudak, P., Functional Reactive Animation. Proc. 2nd ACM SIGPLAN international conference on Functional programming, elokuu 1997, sivut 263-273.

FeP98 Fenton, N. E., Pfleeger, S. L., Software Metrics: A Rigorous and Practical Approach, 2nd Edition, PWS Publishing Co., 1998.

Fla14 Flapjax Demos, The Flapjax Team, http://www.flapjax-lang.org/demos/, [16.7.2014].

Fli14 Flickr API, Yahoo Inc., https://www.flickr.com/services/api/, [6.4.2014].

FrW76 Friedman, D., Wise, D., The Impact of Applicative Programming on Multiprocessing, Tekninen raportti, Indiana University, Computer Science Department, 1976.

Gam94 Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements

of Reusable Object-Oriented Software, Addison-Wesley Professional

Gar05 Garrett, J. J., Ajax: A New Approach to Web Applications, http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/, 2005, [20.8.2014].

Git14 GitHub, Verkossa toimiva ohjelmistoprojektien hosting- ja versionhallintapalvelu, GitHub, Inc., https://github.com/, [11.1.2014].

Goo14a Dart programming language, Google Inc., https://www.dartlang.org/, [13.2.2014].

Goo14b Google Maps JavaScript API v3, Google Inc., https://developers.google.com/maps/documentation/javascript/, [6.4.2014].

Hal77 Halstead, M. H., Elements of Software Science (Operating and programming systems series), Elsevier Science Inc., 1977.

Has14 Yampa, Haskell, http://www.haskell.org/haskellwiki/Yampa, [23.1.2014].

Hid14 Hidat, A., PhantomJS, http://phantomjs.org/, [16.7.2014].

Joy14 Node.js, Joyent, Inc. [29.3.2014].

Jqu14 jQuery, The jQuery Foundation, http://jquery.com/, [23.1.2014].

KBD13 Kambona, K., Boix, E., G., De Meuter, W., An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications. Proc. 7th Workshop on Dynamic Languages and Applications, Montpellier, Ranska, heinäkuu 2013, artikkeli no. 3.

Kow14 Kowal, K., M., Q, https://github.com/kriskowal/q, [23.1.2014].

MaO12 Maier, I., Odersky, M., Deprecating the Observer Pattern with Scala.React. Tekninen raportti, EFPL-REPORT-176887, École Polytechnique Fédérale de Lausanne, Lausanne, Sveitsi, 2012.

Mcc76 McCabe, T. J., A Complexity Measure, IEEE Transactions on Software Engineering, Issue 4 (1976), sivut 308-320.

Mey09 Meyerovich, L. A. et al., Flapjax: A Programming Language for Ajax Applications. Proc. 24th ACM SIGPLAN conference on Object-oriented programming systems languages and applications, Orlando, Florida, USA, lokakuu 2009, sivut 1-20.

Mic14a MS Open Tech Open Sources Rx (Reactive Extensions) – a Cure for Asynchronous Data Streams in Cloud Programming, Microsoft Open Technologies Blog, http://msopentech.com/blog/2012/11/06/ms-open-tech-open-sources-rx-reactive-extensions-a-cure-for-asynchronous-data-streams-in-cloud-programming/, 2012, [9.1.2014].

Mic14b Rx (Reactive Extensions), Microsoft Open Technologies, Inc.,

https://rx.codeplex.com/ [23.1.2014].

Mic14c The Reactive Extensions for JavaScript (RxJS), Microsoft Open Technologies, Inc. http://rxjs.codeplex.com/, [23.1.2014].

Mic14d RxJS examples, Microsoft Open Technologies, Inc. https://github.com/Reactive-Extensions/RxJS/tree/master/examples, [16.7.2014].

MiT07 Mikkonen, T., Taivalsaari, A., Web Applications – Spaghetti Code for the 21st Century. Tekninen raportti, SMLI TR-2007-166, Sun Microsystems Inc., California, USA, 2007.

Net14a Netflix, Verkossa toimiva videoiden suoratoistopalvelu, Netflix, Inc., http://www.netflix.com/, [11.1.2014].

Net14b Functional Reactive Programming (FRP), RxJava Wiki, Netflix, Inc., https://github.com/Netflix/RxJava/wiki, 2014, [8.1.2014].

OtT14 Otto, M., Thornton, J., Bootstrap, http://getbootstrap.com/, [16.7.2014].

Paa14 Paananen, J., Bacon.js, https://github.com/baconjs/bacon.js, [23.1.2014].

Per14 Perriault, N., CasperJS, http://casperjs.org/, [16.7.2014].

Ryd04 Ryder, C., Software Measurement for Functional Programming, Väitöskirja, The University of Kent at Canterbury, 2004.

Sel14 Selenium, http://docs.seleniumhq.org/, [16.7.2014].

Ste10 Stefanov, S., JavaScript Patterns, O'Reilly Media, Inc., 2010.

Tai11 Taivalsaari, A. et al., The Death of Binary Software: End User Software Moves to the Web. 9th International Conference on Creating, Connecting and Collaborating through Computing, Kioto, Japani, tammikuu, 2011, sivut 17-23.

The13 The Reactive Manifesto (v1.1), http://www.reactivemanifesto.org/, 2013, [10.1.2014].

Typ14 Akka, Typesafe, Inc., http://akka.io, [23.1.2014].

W3C14 A Short History of JavaScript, Web Education Community Group Wiki, World Wide Web Consortium (W3C), [6.4.2014].

Liite 1. Toiminnalliset testit

Esimerkkisovelluksen automaattisten toiminnallisten testien tuloste on listattu alla. Testit ovat jaettu viiteen testijoukkoon: alkutilan, tunniste-ehdotusten, kuvahaun, navigaation ja ponnahdusikkunoiden testeihin. Testijoukoissa on yhteensä 78 testiä ja niiden suorittaminen testilaitteistolla (Apple MacBook Pro, 2.3 Ghz Intel Core i7, 8 GB 1600 Mhz DDR3) kesti keskimäärin vähän alle minuutin. Alkuperäiseen tulosteeseen on lisätty rivivaihtoja ja sisennyksiä, jotta se olisi luettavampi tässä yhteydessä.

```
Test file: test.js

# Testing initial state
PASS Loader is hidden
PASS Autocomplete is hidden
PASS Search field is empty
PASS Current page has inactivity indicator
PASS Maximum pages has inactivity indicator
PASS Previous button is disabled
PASS Next button is disabled
PASS Page field is disabled
PASS No markers on map
PASS No info window open
PASS Error is hidden

# Testing suggestions
PASS Do not load suggestions for under two character searches
PASS Do not load suggestions before 500ms timeout from last input
PASS Load suggestions after 500ms timeout from last input
PASS Show loader when loading suggestions
PASS Hide loader when suggestions have been loaded
PASS Show suggestions when they have been loaded
PASS First suggestion in list is the entered value
PASS Correct suggestions are shown
PASS Hide suggestions when search changes
PASS Do not load suggestions for previous search
PASS Do not show previous suggestions before 500ms timeout
PASS Show previous suggestions again after 500ms timeout
PASS Hide suggestions when search field loses focus
PASS Show previous suggestions when search field gains focus
PASS Do not load suggestions when search field focus is set
PASS Hide suggestions when esc key is entered
PASS Load suggestions when search field focus is set and field
     contains text different to previous search
PASS Show error for failing suggestion Flickr request
PASS Hide loader for failing suggestion Flickr request
PASS Hide previous error when starting a new suggestion search

# Testing search
PASS Changing marker count must not start search if search term is not
     defined
PASS Close suggestions when search is started with enter
PASS Close suggestions when suggestion is selected
PASS Set selected suggestion as search field value
PASS Show loader when loading photos
```

```
PASS Start loading photos when suggestion is selected
PASS Do not hide loader until all thumbnails have been loaded
PASS Hide loader when thumbnails have been loaded
PASS Correct markers are shown when thumbnails have been loaded
PASS First page is selected when marker count is changed
PASS Correct markers are shown when marker count is changed
PASS Correct number of pages is set when marker count is changed
PASS Show error for failing search request
PASS Hide previous error when starting a new search request
PASS Do not show error until all thumbnails have been handled
PASS Show error when at least one thumbnail fails
PASS Hide previous error caused by thumbnail loading when starting a
     new search request
PASS Show markers other than with failing image

# Testing navigation
PASS Previous button is disabled on first page
PASS Next button is enabled on first page
PASS Page field is enabled on first page
PASS First page is selected
PASS Second page is selected when next button is clicked
PASS Previous button is enabled on second page
PASS Next button is enabled on second page
PASS Page field is enabled on second page
PASS Correct markers are shown when photos for second page have been
     loaded
PASS Previous button is enabled on last page
PASS Next button is disabled on last page
PASS Page field is enabled on last page
PASS Correct markers are shown when photos for last page have been
     loaded

# Testing infowindow
PASS Info window opens with correct image when marker is hovered
PASS Info window opens with correct image when another marker is
     hovered
PASS Show info window loader when loading photo data
PASS Clicking info window starts loading photo data
PASS Show info window loader when loading large image
PASS Start loading large image when photo data is loaded
PASS Hide info window loader when large image has been loaded
PASS Info window displays correct HTML content
PASS Detail info window closes and thumbnail opens when another marker
     is hovered
PASS Show error for failing photo info request
PASS Hide previous error when starting a new photo info request
PASS Show error for failing photo image
PASS Hide loader for failing photo image
PASS Hide previous photo load error when starting a new photo info
     request
PASS Show loader again after failing photo image
PASS Info window is closed when esc key is pressed

PASS 78 tests executed in 46.325s, 78 passed, 0 failed, 0 dubious, 0
skipped.
```

Liite 2. Tiedostorakenne

Esimerkkisovelluksen tiedostorakenteen päätasolla sijaitsee ainoastaan sovelluksen sisältävä *index.html*-sivu, mittauksen suorittava *Gruntfile.js*-tiedosto ja hakemistoja. *css* ja *img* -hakemistoissa sijaitsevat sovelluksen tyylitiedostot ja kuvat. *js*-hakemiston juuritasolla sijaitsevat sovelluksen eri toteutusversioille yhteinen JavaScript-koodi ja versioiden mukaan nimetyissä alihakemistoissa versiokohtainen koodi. *lib*-hakemisto sisältää sovelluksen hyödyntämät kirjastot: Bootstrapin, Flapjaxin, jQuery:n ja RxJS:n. Kirjastot ovat sijoitettu omiin hakemistoihinsa. Sovelluksen toiminnalliset testit ovat sijoitettuna *test*-hakemistoon.

- index.html
- Gruntfile.js
- css/ → styles.css
- img/ → loader-16x16.gif, loader-32x32.gif
- js/ → common.js, flickr.js, imageloader.js, maps.js, script.js
    - flapjax/ → app.js, flickr.js, imageloader.js
    - js/ → app.js, flickr.js, imageloader.js
    - rxjs/ → app.js, flickr.js, imageloader.js
- lib/
    - bootstrap/
    - flapjax/
    - jquery/
    - rx/
- test/ → mock.js, test.js

Liite 3. Rajapintakuvaukset

Esimerkkisovelluksen kaikkien toteutusversioiden kaikkien komponenttien julkiset rajapinnat on listattu alla. Kaikki komponentit sijaitsevat yhteisen *mashup*-olion alla, joka toimii sovelluksen nimiavaruutena. Komponentin nimeä edeltää version nimi, mikäli komponentti on versiokohtainen ja nimen jälkeen komponentin tyyppi. Tässä tapauksessa kaikki komponentit ovat joko moduuleja tai tavallisia JavaScript-olioita. Metodien paluuarvo *undefined* on JavaScriptin tyyppi, joka tarkoittaa, että metodi ei palauta mitään.

ImageLoader (module)
```
undefined    setDataProvider(function provider)
undefined    imageRequest(function url, function success, function error)
```

Flickr (module)
```
undefined    setDataProvider(function provider)
undefined    flickrRequest(String method, Array params, function success,
                            function error)
Object       createPhotoObject(Object data)
```

Maps (module)
```
undefined           init(String elementID)
Google.Maps.Marker  addMarker(Object photo, function overCallback,
                              function clickCallback)
Maps.Infobox        createInfobox()
undefined           removeMarkers()
```

Maps.Infobox (Object)
```
undefined    open(Google.Maps.Marker marker)
undefined    close()
undefined    setImage(String image)
undefined    setLoader()
undefined    setDetails(Object photo)
```

Helpers (module)
```
keyCodes.escape
keyCodes.enter
undefined    setSuggestions(jquery $suggestions, Array items)
```

js/App (module)
```
undefined    init()
```

js/ImageLoader (module)
```
undefined    load(function url, function sucess, function error)
```

js/Flickr (module)
```
undefined    getRelatedTags(String tag, function success, function error)
undefined    searchPhotos(String tag, Number perPage, Number page,
                           function success, function error)
undefined    getPhotoInfo(Number id, function success, function error)
```

rxjs/App (module)
```
undefined    init()
```

rxjs/ImageLoader (module)
```
Rx.AsyncSubject     load(function url)
```

rxjs/Flickr (module)

```
Rx.AsyncSubject     getRelatedTags(String tag)
Rx.AsyncSubject     searchPhotos(String tag, Number perPage, Number page)
Rx.AsyncSubject     getPhotoInfo(Number id)
```

flapjax/App (module)

```
undefined           init()
```

flapjax/ImageLoader (module)

```
F.EventStream       load(function url)
```

flapjax/Flickr (module)

```
F.EventStream       getRelatedTags(String tag)
F.EventStream       searchPhotos(String tag, Number perPage, Number page)
F.EventStream       getPhotoInfo(Number id)
```

Liite 4. Käyttöliittymäskenaarion lähdekoodi

Alla olevassa lähdekoodissa on kuvahaun järjestämisperusteen lisäävän muutosskenaarion lähdekoodimuutoksien keskeisimmät osat. Koodiin on lisätty kommentteja luettavuuden parantamiseksi.

```
/*
 * JavaScript-version keskeisimmät muutokset
 */

// Asetetaan tapahtumankäsittelijät painikkeille
$sortNewButton.click(function() {
	setSortButton('new');
	searchPhotos(lastSearchText, 1);
});
$sortOldButton.click(function() {
	setSortButton('old');
	searchPhotos(lastSearchText, 1);
});

// Luetaan painikkeen arvo
var sort = $sortOldButton.hasClass('active') ? 'old' : 'new';

// Tutkitaan, eroaako haku edellisestä
if ((text !== lastSearchText ||
	page !== currentPage ||
	photosOnPage !== currentPhotosOnPage ||
	sort !== currentSort) &&
	text !== undefined && text.length > 0) {
	…

/*
 * RxJS-version keskeisimmät muutokset
 */

// Luodaan tapahtumavuo järjestämisperusteelle
var sortSubject = new Rx.BehaviorSubject('new');

// Asetetaan tapahtumankäsittelijät painikkeille
$sortNewButton.click(function() {
	sortSubject.onNext('new');
});
$sortOldButton.click(function() {
	sortSubject.onNext('old');
});

// Tilataan tapahtumat ja päivitetään käyttöliittymään
sortSubject.subscribe(function(x) { setSortButton(x); });

// Lisätään järjestämisperuste osaksi lopullista hakuvuota
var searchStream = Rx.Observable.return()
	.combineLatest(textSubj, countSubj, pageStream, sortSubj,
		function(_, text, photoCount, page, sort) {
			return {
				text: text,
				page: page,
				photosOnPage: photoCount,
				sort: sort
			};
		});
```

```
/*
 * Flapjax-version keskeisimmät muutokset
 */

// Luodaan tapahtumavuo järjestämisperusteelle
var sortE = F.receiverE();
$sortNewButton.click(function() {
	sortE.sendEvent('new');
});
$sortOldButton.click(function() {
	sortE.sendEvent('old');
});

// Muodostetaan tapahtumavuosta reaktiivinen arvo, jonka muuttuessa
// päivitetään käyttöliittymää
var sortB = sortE.startsWith('new');
sortB.liftB(function(x) {
	setSortButton(x);
});

// Lisätään järjestämisperuste osaksi viimeisen haun määrittävää
// reaktiivista arvoa
var searchB = F.liftB(function(text, count, page, sort) {
	return {
		text: text,
		page: page,
		photosOnPage: count,
		sort: sort
	};
}, textB, countB, pageB, sortB);
```

Liite 5. Palvelupyyntöskenaarion lähdekoodi

Alla olevassa lähdekoodissa on kaikkien eri versioiden getRelatedTags-metodin toteutus. Koodiin on lisätty kommentteja luettavuuden parantamiseksi.

```
/*
 * JavaScript-versio
 */
module.getRelatedTags = function(tag, success, error) {
	var tries = 3;
	processRequest();
	function processRequest() {
		getRequest(tag, function(tags) {
			success(tags);
		}, function() {
			processError();
		});

	function processError() {
		tries--;
		if (tries > 0) {
			setTimeout(processRequest, 1000);
		} else {
			error();
		}
	}
};

/*
 * RxJS-versio
 */
module.getRelatedTags = function(tag) {
	var first = true;
	return Rx.Observable.defer(function() {
		if (first) {
			first = false;
			return getRequestStream(tag);
		} else {
			return Rx.Observable.timer(5000).flatMap(
				function() { return getRequestStream(tag); });
		}
	})
	.retry(2);
};

/*
 * Flapjax-versio
 */
module.getRelatedTags = function(tag) {
	var tries = 3;
	var response = F.receiverE();
	var requests = F.receiverE();
	requests
		.switchE()
		.mapE(function(x) {
			tries--;
			if (x !== null) {
				response.sendEvent(x);
			} else if (tries > 0) {
				F.timerE(5000).mapE(function() {
					requests.sendEvent(getRequestStream(tag));
				});
			} else {
```

```
                        response.sendEvent(null);
                }
        });

        requests.sendEvent(getRequestStream(tag));

        return response;
};
```