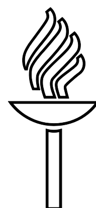
Aleksi Munter

KÄYTTÖLIITTYMÄOHJELMOINNIN PARADIGMAMUUTOS JA SEN TAUSTATEKIJÄT 2013-2019



JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2020

TIIVISTELMÄ

Munter, Aleksi
Käyttöliittymäohjelmoinnin paradigmamuutos ja sen taustatekijät 2013-2019
Jyväskylä: Jyväskylän yliopisto, 2020, 26s.
Tietojärjestelmätiede, kandidaatintutkielma
Ohjaaja: Ruohonen, Toni

Käyttöliittymien ohjelmoinnissa ja siinä hyödynnettävissä ohjelmointikehyksissä on vuoden 2013 jälkeen tapahtunut merkittävä siirtymä imperatiivisesta ohjelmointimallista deklaratiiviseen ja datavirtojen hallinnassa kaksisuuntaisista yksisuuntaisiin malleihin.

Tutkielmassa tarkastelen kirjallisuuskatsauksena mitkä kognitiiviset tekijät ovat ajaneet muutosta ja onko muutoksen myötä syntynyt mitattavissa olevia hyötyjä käyttöliittymäohjelmoinnin ja ohjelmistokehityksen teknisiin haasteisiin. Kognitiivisten tekijöiden suhteen arvioin minkälaisia tunnettuja eroja ohjelmointimallien välillä on, miten ne voisivat selittää siirtymän alkusyitä ja onko mallien välillä olemassa havaittavia eroja sen suhteen, miten kognitiivinen kuorma jakautuu ohjelmointityössä.

Asiasanat: käyttöliittymäohjelmointi, deklaratiivinen ohjelmointi, imperatiivinen ohjelmointi, yksisuuntaiset datavirrat, ohjelmistotekniikka

ABSTRACT

Munter, Aleksi
User interface programming paradigm shift and its drivers 2013-2019
Jyväskylä: University of Jyväskylä, 2020, 26pp.
Computer Science, bachelor's thesis
Supervisor: Ruohonen, Toni

Since 2013, user interface programming and related frameworks have gone through a monumental shift from imperative to declarative programming paradigm and in data flow handling from bi-directional to unidirectional models.

This literature review aims to identify the cognitive drivers behind the shift and whether the shift has caused measurable benefits in regard to technical challenges within user interface programming and software development. It also aims to review the known differences between the two programming paradigms, how the differences could explain the origins of the shift, and whether cognitive load distribution in programming work is affected by the two paradigms.

Keywords: user interface programming, declarative programming, imperative programming, unidirectional data flow, software development

KUVIOT

KUVIO 1 Stack Overflow -sivuston kysymykset kirjastoittain 2009-2019 12
KUVIO 2 Kirjastokohtaiset kuukausittaiset latausmäärät 2015-2019 13
KUVIO 3 State of JS kyselytutkimuksen tulokset vuosina 2016-2018.................. 14

SISÄLLYS

TIIVISTELMÄ
ABSTRACT
KUVIOT

1 JOHDANTO 6

2 KÄYTTÖLIITTYMÄOHJELMOINTI 8

3 PARADIGMAMUUTOS LUKUINA 11

4 OHJELMOINTIMALLIT JA DATAVIRRAT 15
4.1 Ohjelmointimallien erot 16
4.2 Arkkitehtuurimallien erot 19
4.3 Datankuljetusmallit 20

5 YHTEENVETO 22

1 JOHDANTO

Vuoden 2013 jälkeen käyttöliittymäohjelmoinnissa on tapahtunut merkittävä muutos imperatiivisesta ohjelmointimallista deklaratiiviseen, ja kaksisuuntaisista datasidoksista yksisuuntaiseen datankuljetusmalliin. Muutos sai alkunsa käyttöliittymäohjelmoinnin erityispiirteistä kumpuavista teknisistä tarpeista ja sen valtavirtaistumisen alkupisteeksi voidaan nähdä Facebookin 2013 avoimen lähdekoodin julkaisu sisäisesti kehitetystä React[1] -ohjelmointikehyksestä. Vaikka aiempi kehitys jo enteili muutosta ja React itsessään ei tarjoa ratkaisuja kaikkiin muutoksen taustalta löytyviin ongelmiin, sen julkaisu johti ympäröivän ekosysteemin nopeaan kasvuun. Samalla esille on noussut muita kilpailevia tai täydentäviä ohjelmointikehyksiä, kuten Vue.js[2], ja osa jo olemassa olleista kehyksistä—esimerkiksi Angular[3] ja EmberJS[4]—ovat omaksuneet kehityksen myötä syntyneitä ohjelmointimalleja.

Käyttöliittymäkehitykseen sisältyy erityispiirteitä, joiden ratkaiseminen ei perinteisesti ole ollut helppoa. Käyttöliittymien loppukäyttäjälle näkyvän osan—ulkoasu, siirtymät, interaktiot—suunnittelu on kokonaan oma ongelmakenttänsä, jota tämä tutkielma ei käsittele, vaan tarkastelun kohteena on käyttöliittymien ohjelmointiin ja niiden arkkitehtuurisuunnitteluun liittyvät haasteet. Käyttöliittymäohjelmoinnin erityishaasteet on tunnistettu verrattain varhain: Myers (1993) esittelee kirjoitusajankohdasta kumpuavien ongelmakohtien lisäksi seuraavat, edelleen relevantit haasteet: käyttäjän interaktioihin reagointi; samanaikaisuus ja asynkronisuus; virheensieto ja -hallinta; graafisen käyttöliittymän testaaminen; komponenttien vastuualueiden rajaaminen (Myers, 1993). Karagkasidis (2008) tunnistaa ongelmakohdiksi käyttöliittymän luomisen ja koostamisen, käyttäjäinteraktiot, bisneslogiikan sitomisen käyttöliittymään ja monimutkaisten, käyttäjäinteraktion ja sovelluslogiikan väliseen toimintaan perustuvien skenaarioiden toteuttamisen (Karagkasidis, 2008). Foust, Järvi ja Parent (2015) korostavat erityisesti asynkronisuudesta, käyttäjän toimiin reagoimisesta ja tapahtumien järjestyksen hallinnasta syntyviä haasteita. Toteutuk-

[1] https://reactjs.org
[2] https://vuejs.org
[3] https://angular.io
[4] https://emberjs.com

sissa, joissa otetaan vastaan syötteitä käyttäjältä, tulee varautua koodin toisintumiseen niin toimintalogiikassa kuin validoinnissa ja käyttäjien kulttuuri määrittelee esitystapaan liittyviä ominaisuuksia, tekstin tulostussuunnasta numeroiden ja päivämäärien muotoiluun. (Cerny, Chalupa & Donahoo, 2012). Edellä mainittujen lisäksi päätelaiteiden moninaistuessa toteutuksissa tulee varautua ennalta määrittelemättömään määrään ruutukokoja, pikselitiheyksiä, näytön orientaatioita, saavutettavuusasetuksia ja ominaisuusvalikoimia, mikä tekee käyttöliittymien suunnittelusta ja kehityksestä oleellisesti dynaamisempaa ja huomattavasti monimutkaisempaa kuin hyvin määriteltyjen, staattisten järjestelmien.

Voidaan todeta, että miltei lähteestä riippumatta käyttöliittymäkehityksen merkittävimmäksi ongelma-alueeksi nähdään ennustamaton dynaamisuus: asynkronisuudesta, reaktiivisuudesta ja sovelluksen tilan ja sen muutosten hallinnasta kumpuavat haasteet. Toissijaisena—joskin lähes yhtä merkittävänä—esille nousee esille sovelluksen staattisemmat osa-alueet, käyttöliittymän koostaminen ja sisäinen vastuunjako. Vaihtoehtoinen jako voisi olla käyttäjälähtöiset haasteet ja sovellustason haasteet. Tämän tutkielman aiheena oleva siirtymä kehityksessä sai alkusysäyksensä ongelmakentän dynaamisten osa-alueiden kautta (Hunt, 2013; Fisher & Chen, 2014), mutta ekosysteemin kasvun myötä on noussut esille uusia, hyväksi todettuja tapoja hallita sovelluksen staattisempia osia.

Tässä työssä pyritään osoittamaan, että uudet ohjelmointimallit tukevat käyttöliittymäohjelmointiin erityisesti sopivaa mentaalista mallia uudelleenjakamalla kognitiivista kuormaa tilanhallinnan ja komponenttien esitystavan suhteen eri tavalla kuin aiemmat ohjelmointimallit. Koska työssä keskitytään JavaScript-ympäristössä tapahtuvan verkkokäyttöliittymien kehityksen muutokseen, tulisi termi käyttöliittymäkehitys myöhemmin käsittää selaimessa esitettävien käyttöliittymien kehittämiseksi, ja maininnat ohjelmointiparadigmoista tulisi ymmärtää viitteinä ohjelmointitapoihin JavaScript-kielen sisällä. On kuitenkin hyvä pitää mielessä, että muutos on heijastunut myös tämän rajauksen ulkopuolelle; Googlen esittelemä Flutter[1] ja Facebookin ComponentKit[2] ovat pyrkineet tuomaan deklaratiivista ohjelmointimallia mobiiliapplikaatioiden käyttöliittymäkehitykseen ja React Native[3] on varteenotettava usean alustan hybridikehys. Johtopäätelmiä ohjelmointimalleista ja niiden soveltuvuudesta käyttöliittymäkehitykseen tulisikin käsitellä yleispätevinä alustasta riippumatta.

---

[1] https://flutter.dev
[2] https://componentkit.org
[3] https://facebook.github.io/react-native

2 KÄYTTÖLIITTYMÄOHJELMOINTI

Ennen vuotta 2009—ja hyvän aikaa sen jälkeenkin—käyttöliittymäkehityksen mallit olivat varsin hajanaisia. Käyttöliittymät tuotettiin lähes poikkeuksetta suoraan taustajärjestelmistä staattisina, ja dynaamiset ja interaktiiviset ominaisuudet lisättiin jo tuotettuun HTML-koodiin päätelaitteessa. Syitä tähän on monia: valtavirtaistunut internet itsessään oli verrattain nuori teknologia; selainten ominaisuuserot olivat merkittäviä; päätelaitteiden laskentatehoista ei ollut takeita ja mobiilipäätelaitteista ei voinut varsinaisesti edes puhua ennen kosketusnäytöllistä laitekantaa; asynkronisen ja atomisen datan kuljettamiseen ei ollut vakiintuneita metodeja selainten välillä; kaiken tämän lisäksi ekosysteemi ei tarjonnut riittävän kypsiä ja yleisesti käytettyjä työkaluja tehokkaaseen kehittämiseen.

Selainten JavaScript-toteutusten standardoitumisen, kehitysyhteisön rakentumisen, työkalujen kypsymisen, laitteiden suorituskyvyn jatkuvan kehityksen taittumisen ja kuvauskielten standardien kehittymisen myötä alkoi perusta nykyisenkaltaiselle pääasiallisesti päätelaitteissa tapahtuvalle, taustajärjestelmistä itsenäiselle käyttöliittymäkehitykselle olla valettuna. Näistä lähtökohdista kehittyi kutakuinkin samaan aikaan useampia kilpailevia sovelluskehyksiä, kuten Angular, EmberJS, ja Backbone[1], joista kaikki tarjosivat *Model-View-Controller*-malliin (MVC) tai johonkin sen varianttiin perustuvan kehyksen verkkokäyttöliittymien kehitykseen sekä datan automaattisen sidonnan niiden tuottamiin näkymiin. Nämä sovelluskehykset vastasivat pääasiallisesti sovelluskehityksen staattisluonteisiin ongelmiin. Vastuunjaossa hyödynnetyt arkkitehtuurimallit noudattelivat staattisia näkymiä tuottavien palvelinpään toteutusten käytäntöjä ja käyttäjälähtöiseen, dynaamiseen ongelmakenttään ei esitelty varsinaisesti uusia ratkaisuja. Datan automaattinen sidonta liittyy näennäisesti sovelluksen dynaamisiin osa-alueisiin, mutta lähemmin tarkasteltuna sen pääasialliset hyödyt koskivat näkymien muodostamista: esittäviä komponentteja ei datan muuttuessa tarvinnut päivittää manuaalisesti. Toteutuksiin useimmiten liittynyt datan sidonnan kaksisuuntaisuus, jossa näkymät pystyivät suoraan vaikuttamaan sovelluksen tilaan, jopa korosti dynaamiseen ongelmaken-

[1] https://backbonejs.org

tän haasteita—tähän palataan myöhemmin arkkitehtuurimallien eroja tarkastellessa.

React lähtee monilta osin samoista lähtökohdista (Hunt, 2013), mutta sisältää muutaman erityispiirteen, jotka toimivat seuranneen murroksen vahvoina ajureina. Sinällään yksinkertainen ajatus näkymien kuvauskielen tuomisesta osaksi komponentteja JSX-kielilaajennoksella poistaa tarpeen tilanhallintaan ja näkymien päivittämiseen erikoistuneisiin kontrollereihin; komponentit esittävät lähtökohtaisesti vain ja ainoastaan niille annetun tilan. Toinen merkittävä piirre on lähtökohtaisesti tekninen: sovelluksen komponenttien kunkin hetkisen tilan muodostaminen erotettiin selaintoteutuksesta varjomallin avulla. Varjomalli kuvaa selaimessa esitettävän rakenteen ja rakenteessa tapahtuneet muutokset päivitetään vain tarvittaessa selaimen sisältämään dokumenttimalliin. Vaikka kyseessä on tietyllä tapaa vain toteutusyksityiskohta, tämän voidaan nähdä johtaneen mallina uudenlaiseen lähestymistapaan sovelluksen tietovirtojen hallinnassa: sovelluksen tilan voidaan käsittää heijastelevan sen varjomallia. Ensimmäinen askel tähän suuntaan oli Facebookin esittelemä Flux-arkkitehtuurimalli, jossa sovelluksen tilaa käsitellään esittävistä komponenteista täysin erillisinä tietovarastoina (Fisher & Chen, 2014). Jos aiemmin sovelluksen tila ja tilankäsittelyfunktiot oli upotettu komponenttitasolla niiden malleihin ja kontrollereihin, niin nyt tila käsiteltiin omissa komponenteissaan. Tästä luonteva askel, jonka yhtenä merkittävänä ajurina oli kolmannen osapuolen kirjasto Redux[1], oli siirtyä malliin, jossa sovelluksen tila on esitystavasta erillään ja muuttumaton (*immutable*). Tilan muutokset johtavat aina kokonaan uuteen tilaan, ja muutokset tapahtuvat tilaa muokkaavien, puhtaiden funktioiden kautta (Redux, n.d.). Kolmannekseen, React otti näkymien muodostamiseen ja koostamiseen MVC-mallin tai jonkun sen variantin sijaan komponenttipohjaisen arkkitehtuurin. Komponenttipohjaisessa arkkitehtuurissa kukin näkymä on itsenäinen komponentti, ja näitä komponentteja voidaan koostaa vapaasti. Komponentit muodostavat—hyvinkin samalla tapaa kuin sovelluksen dokumenttimalli—puurakenteen, joka kuvaa käyttöliittymän kokonaistilan.

Ja tätä kautta päästään nykytilanteeseen, jossa sovelluksen esitystapa on puhtaimmillaan vain sen tilan funktio, ja sovellusta kehitettäessä päähuomio siirtyy teknisistä yksityiskohdista sovelluksen tilan hallintaan. Esittävät komponentit pelkistyvät puhtaiksi funktioiksi, joiden ulkoasun määrittelee sovelluksen kokonaistila. Ja tästä tilanteesta kumpuaa tämän tutkimuksen keskeinen ajatus: lähestyttäessä käyttöliittymäkehitystä ensisijaisesti sovelluksen tilanhallinnan kautta, saavutetaan hyötyjä, joita on vaikea saavuttaa perinteisin keinoin.

1. Tilan ottaessa etusijan ohjelmoijan ajattelussa on vaikeampi päätyä ennalta määrittelemättömiin tiloihin, mikä vähentää ohjelmointivirheitä.
2. Kun tila on hyvin määritelty, on sen esittäminen triviaalia. Ei-triviaalit komponentit toimivat henkisinä varoitusmerkkeinä komponenttien rajojen epäoptimaalisesta määrittelystä. Tämä tukee komponenttirajojen optimaalista löytämistä ja tätä kautta sovelluksen osien uudelleenkäytettävyyttä.

[1] https://redux.js.org

3. Sovelluksen esittäminen tilan funktiona poistaa useita sovelluskehityksen dynaamisen ongelmakentän haasteita. Jokaisen tilan muutoksen johtaessa automaattisesti uuteen esitystapaan, ei viestinvälityksestä tarvitse huolehtia.
4. Sovelluslogiikka saadaan eriytettyä esitystavasta lähes täysin, ja tietyissä tapauksissa siirrettyä kokonaan taustajärjestelmiin, mikä tekee samaa sovelluslogiikkaa hyödyntävien ohuiden käyttöliittymien toteuttamisesta huomattavasti kevyempää.

Pohjimmiltaan nämä hyödyt kuvaavat valitun ohjelmointi- ja tilankäsittelymallin mahdollistamaa kognitiivisen kuorman siirtymistä ja fokuksen uudelleenkohdentumista.

3 PARADIGMAMUUTOS LUKUINA

Käyttöliittymäkehityksessä tapahtunutta murrosta voi tarkastella tutkimalla pakettienhallintajärjestelmien ja tukisivustojen tarjoamia tilastoja, sekä kehittäjäyhteisölle suunnattuja kyselyitä. Tarkasteltaviksi on valittu seuraavat kirjastot ja sovelluskehykset:

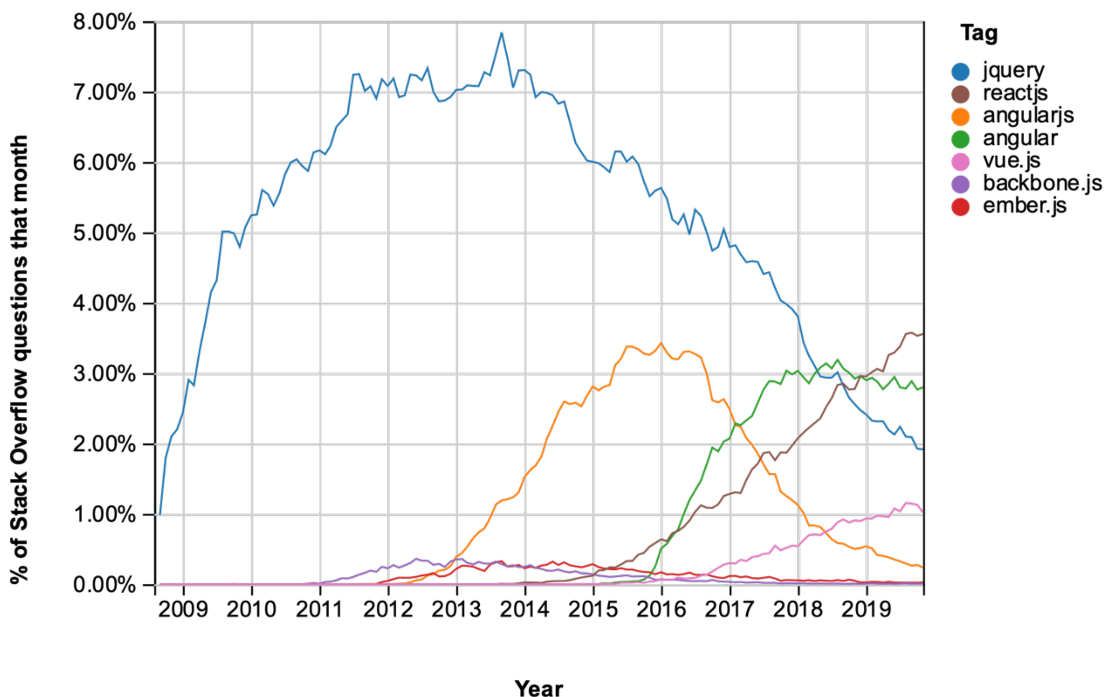
1. jQuery[1]
2. Ember ja Backbone
3. AngularJS ja Angular
4. React
5. Vue.js

Tarkastelu on tehty ensisijaisesti Stack Overflow -sivuston Trends[2] -työkalua käyttäen, tarkastellen sivustolta löytyvien kysymysten määrää kirjastokohtaisesti (aineisto haettu 25.12.2019). Kuviosta 1 nähdään, kuinka miten eri ohjelmistokehyksiin liittyvien kysymysten määrät sivustolla ovat kehittyneet vuoden 2008 jälkeen. Vaikka kysymysten määrä ei ole välttämättä kaikkein kuvaavin mittari sovelluskehysten ja kirjastojen suosiolle—sehän voi myös kertoa kirjaston vaikeasta ymmärrettävyydestä ja käytettävyysongelmista—se voidaan kuitenkin nähdä parempana indikaattorina kirjastojen käyttöönottoasteesta kuin paketinhallintajärjestelmien raportoimat latausmäärät.

Syitä tähän on useita. Paketinhallintajärjestelmien tilastot ovat luonteeltaan kumulatiivisia, koska vanhat edelleen ylläpidetyt sovellukset aiheuttavat latauksia, mikä tekee trendien havaitsemisesta vaikeampaa. Ekosysteemin murrosvaiheessa paketinhallintajärjestelmiä oli useita, joista tärkeimpiä olivat npm[3] ja Bower[4]. Viimeistään Bower:n lopettamisen jälkeen (Stankiewicz, 2017) npm:ää voidaan pitää käytännön standardina. Paketinhallintajärjestelmien suhteellisen lyhyestä elinkaaresta ja kentän jakautuneisuudesta johtuen kattavaa ja yhtenäistä tietoaineistoa ei ole saatavilla. Pidemmän historiansa lisäksi Stack

[1] https://jquery.com
[2] https://insights.stackoverflow.com/trends
[3] https://www.npmjs.com
[4] https://bower.io

Overflow on kirjastojen käyttöönottovaiheessa todennäköisesti herkempi muutoksille kuin paketinhallintajärjestelmät, koska kiinnostus teknologiaan näkyy kysymysten määrän kasvuna jo ennen kuin varsinaisia toteutuksia julkaistaan.



KUVIO 1 Stack Overflow -sivuston kysymykset kirjastoittain 2009-2019

jQueryä voi hyvällä syyllä kutsua 2000-luvun tärkeimmäksi JavaScript-kirjastoksi ja se osaltaan vaikutti merkittävästi JavaScriptin vakiintumiseen käyttöliittymäohjelmoinnissa. Vaikka se on luonteeltaan suhteellisen matalan tason kirjasto ja pohjimmiltaan imperatiivinen, se siltasi selainten välisiä eroja, tarjosi tehokkaan tavan muokata selaimen dokumenttimallia ja suosionsa myötä tuki vahvasti ekosysteemin ja työkalujen kehitystä. jQueryn hallitseva asema on ilmiselvää tarkastellessa lähtötilannetta, ja huolimatta toisen sukupolven sovelluskehysten julkaisun jälkeisestä voimakkaasta laskusta sen näkyvyys on edelleen merkittävää. Pääasiallisena syynä tähän lienee sen suosio sisällönhallinta-alustoissa — mm. Wordpress [1] ja Drupal [2] — joiden oletusasennuksiin se yleensä sisältyy.

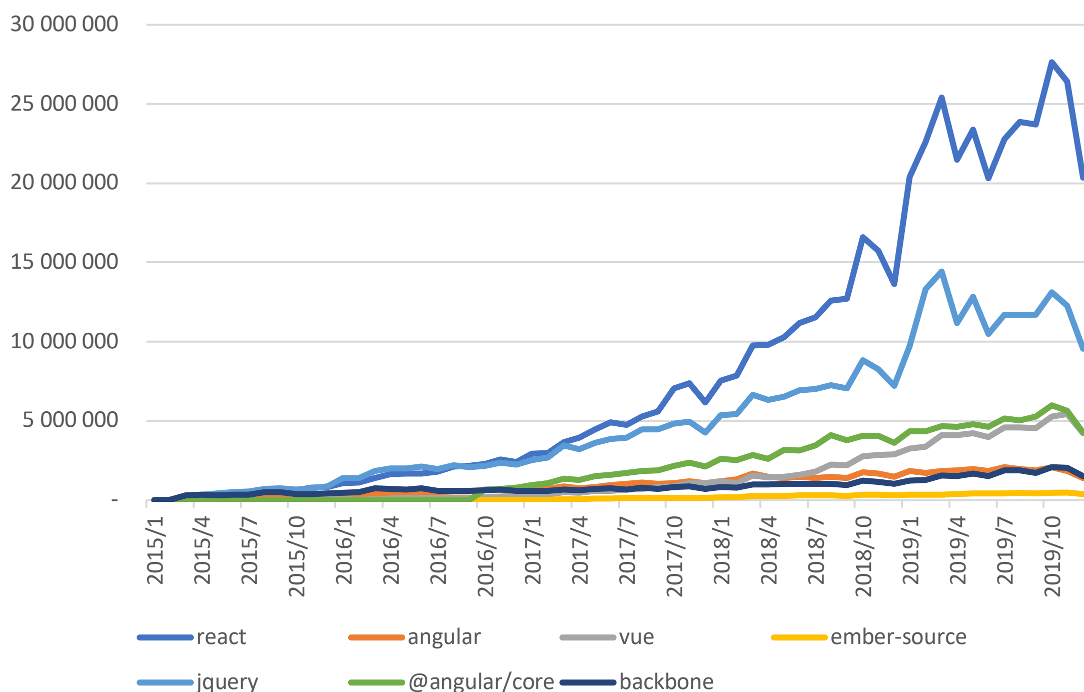
EmberJS ja Backbone on valittu esimerkeiksi toisen sukupolven sovelluskehyksistä, jotka tarjosivat näkymien muodostamisen ja datan automaattisen sidonnan näkymiin. Niiden suhteellisen vähäinen osuus verrattuna muihin kuvion kirjastoihin kertoo osaltaan vastaavien kirjastojen lukuisuudesta — samoja ominaisuuksia tarjosi moni muukin. Molempien tapauksessa kasvu vuosina 2011-2013 on selvää, mutta tämän jälkeen lasku on ollut tasaista, ja voidaan todeta, että molemmat kirjastot ovat tällä hetkellä kuriositeetteja.

[1] https://wordpress.org
[2] https://www.drupal.org

Erilliseen tarkasteluun on valittu toisen polven sovelluskehyksistä voittajaksi selvinnyt AngularJS, joka kasvoi vahvasti vuoteen 2016 asti, kunnes React alkoi kasvattaa näkyvyyttään vahvasti. AngularJS:n toinen versio, Angular (aiemmin Angular 2), muovautui vahvasti samojen käytäntöjen pohjalta kuin React ja johtuen AngularJS:n suosiosta sen käyttöönotto oli nopeaa. Angularin kasvu kuitenkin on taittunut laskuun vuoden 2018 aikana ja jälkeen.

Murroksen aloittanut React julkaistiin 2013, mutta sen suosio alkoi kasvaa vasta vuonna 2015. Osaltaan viivettä selittänee sovelluskehystä ympäröivän ekosysteemin muodostumiseen kulunut aika, osaltaan uusien käytäntöjen oppimiskynnys. Kasvu tämän jälkeen on ollut huomattavan ripeää ja tasaista, eikä se ole näyttänyt merkkejä taittumisesta. Vue.js:n olen valinnut esimerkiksi uudesta sovelluskehyksestä, joka on ottanut pohjakseen useita samoja ohjelmointimalleja kuin React huolimatta kosmeettisesti suurehkoista eroista ja jonka kehitys on ollut nopeaa sen julkaisun jälkeen.

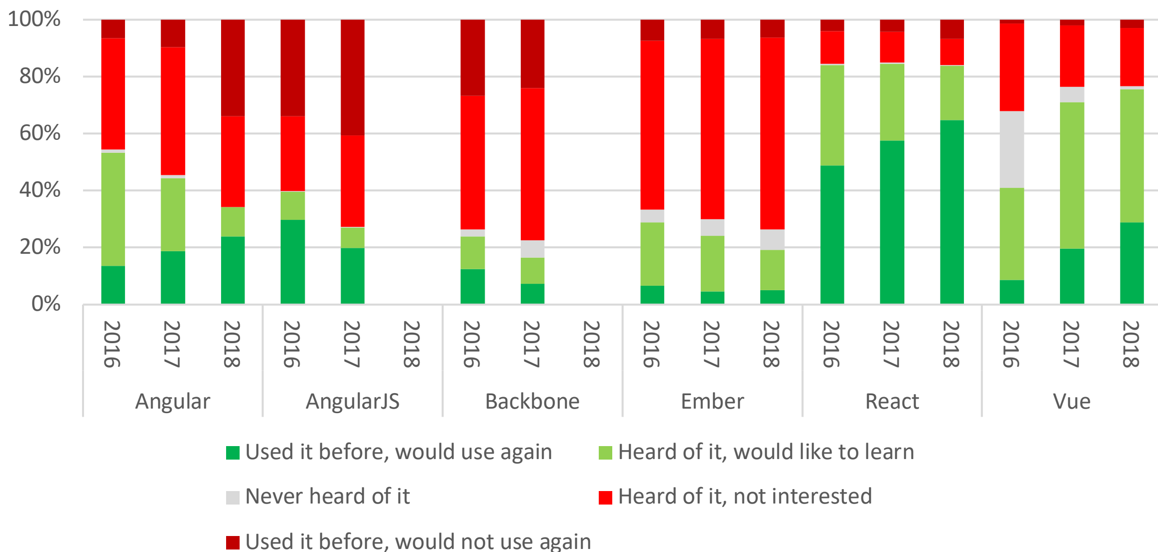
Kuviossa 2 nähdään npm-paketinhallintajärjestelmän kuukausittaiset latausmäärät kirjastoittain siltä ajalta, kun tilastoja on saatavilla tammikuusta 2015 joulukuuhun 2019 (aineisto noudettu 25.12.2019 osoitteesta https://api.npmjs.org). Tilastojen tulkitseminen on vaikeampaa, mutta tässäkin nähdään Vue.js:n nousu, Emberin stagnaatio ja Angularin molempien versioiden kasvun hidastuminen suhteessa Reactiin ja Vue.js:ään. Backbonen latausmäärissä on yllättävää kasvua, jota ei Stack Overflow -kysymyksiä tarkasteltaessa ole havaittavissa.



KUVIO 2 Kirjastokohtaiset kuukausittaiset latausmäärät 2015-2019

Kuvio 3 näyttää StateOfJS -kyselytutkimuksen tulosten kehittymisen vuosina 2016-2018. Tulokset tukevat aiempaa analyysia — deklaratiivista ohjelmoin-

timallia hyödyntävät sovelluskehykset kasvattavat suosiotaan, imperatiiviset kehykset ovat joko kadonneet marginaaliin tai pysähtyneet paikoilleen.



KUVIO 3 State of JS kyselytutkimuksen tulokset vuosina 2016-2018

Tarkasteltavat deklaratiiviset sovelluskehykset—Angular, React ja Vue.js—ovat kaikki kasvattaneet "Käyttänyt aiemmin, käyttäisi uudestaan" -osuuttaan. Angularin osalta tosin huomattavaa on merkittävä käyttäjien epätyytyväisyys. Toisen polven sovelluskehyksistä molemmat, AngularJS ja Backbone, ovat kutistuneet siinä määrin että tuloksia ei ole saatavilla enää vuodelle 2018. Ember on onnistunut pitämään osuutensa, joskin kiinnostuksen lasku on selvää.

Yllä esiteltyjen statistiikkojen perusteella—huolimatta vertaisarvioidun tutkimuksen puutteesta—voitaneen suhteellisen turvallisesti todeta, että siirtymä imperatiivisista, perinteisiä arkkitehtuureja noudattavista sovelluskehyksistä deklaratiivisiin ja komponenttipohjaisiin sovelluskehyksiin on todellinen ilmiö. Seuraavissa osioissa esitellään tarkemmin ohjelmointi- ja datankuljetusmallit historiallisessa kontekstissa, käsitellään niiden eroja, ja pyritään osoittamaan, että nykysuuntaus on käyttöliittymäohjelmoinnin kannalta merkittävästi parempi kuin aiemmin hyödynnetyt mallit.

4 OHJELMOINTIMALLIT JA DATAVIRRAT

Ohjelmointimalli (*paradigm*) kuvaa ohjelmointikielen ominaisuuksia, mutta käytännössä jaottelu on suhteellisen teoreettinen ja keinotekoinen; monet suositut ohjelmointikielet toteuttavat monia ohjelmointimalleja. Jopa alun perin yksiselitteinen olio-ohjelmointikieli Java on tarjonnut versiosta 8 alkaen funktionaaliseen ohjelmointiin sopivia mekanismeja ensimmäisen asteen ja nimettömien funktioiden kautta. Tämän vuoksi tämän tutkimuksen tarkastelun rajaus *imperatiivisiin* ja *deklaratiivisiin* kieliin ja sovelluskehyksiin tulisikin ottaa viitteellisenä, sovelluskehysten ja kirjastojen pääasiallista ohjelmointitapaa vastaavana lajitteluna. Myös JavaScript on kielenä moniparadigmainen ja mahdollistaa usean eri ohjelmointityylin prototyyppipohjaisesta olio-ohjelmoinnista funktionaaliseen ohjelmointiin. JavaScriptin funktionaaliset ominaisuudet ovat olleet keskeisessä roolissa siirryttäessä käyttöliittymien imperatiivisista toteutuksista niiden deklaratiivisen kuvauksen mahdollistaviin sovelluskehyksiin.

Imperatiivinen ja deklaratiivinen ohjelmointi nähdään usein toistensa vastakohtina: imperatiivisessa ohjelmoinnissa pyritään kertomaan, *miten ohjelmisto toimii* tietyssä järjestyksessä annetuilla komennoilla; deklaratiivisessa ohjelmoinnissa pyritään kuvaamaan *mitä ohjelman pitäisi tehdä* ottamatta kantaa varsinaisen sisäiseen toteutukseen. Tietokoneiden sisäinen toimintamalli on käytännössä poikkeuksetta imperatiivinen ja heijastellen tätä useat ohjelmistokielet ovat kehittyneet imperatiivisessa kontekstissa. Tämä näkyy myös tarkastellessa koko ohjelmistosuunnittelun kenttää: valtavirrassa suositut ohjelmointikielet ovat olleet lähes poikkeuksetta imperatiivisia ja siirtymät ohjelmointimallista toiseen ovat tapahtuneet pääasiallisesti imperatiivisen paradigman sisällä, ensin strukturoidusta ohjelmoinnista proseduraaliseen ja tästä olio-ohjelmointiin. Tämä on näkynyt myös koulutuksessa; vielä vuonna 2001 The Joint Task Force on Computing Curricula IEEE Computer Society Association for Computing Machinery suosittelee Computing Curriculassa (2001) perusopintojen rakenteessa ensisijaisesti imperatiivisen ja olio-ohjelmoinnin opetusta, ja saman tahon Computer Science Curriculassa (2013) todetaan, että ehdotusten välisenä ajanjaksona käytännön opetuksen trendit ovat olleet siirtyminen matalan tason kielistä korkeamman tason kieliin, esimerkiksi C:stä Javaan, ja tulkittujen kielien, kuten Python ja JavaScript, esiinnousu. Yhteistä näille on kuitenkin saman pa-

radigman sisällä pitäytyminen. Tältä pohjalta ei olekaan yllättävää, että päätelaitteissa tuotettujen verkkokäyttöliittymien yleistymisen alkuaikoina JavaScriptin pääasiallinen käyttötapa oli imperatiivinen, ja tarkemmin proseduraalinen: prototyyppipohjainen olio-ohjelmointi ei vaikuttanut ottavan valtavirrassa tulta alleen, ja vasta JavaScriptin ES6 -version esittelemä luokkasyntaksi vaikutti tekevän luokkien käytöstä helposti omaksuttavaa klassiseen oliomalliin tottuneille. Proseduraalisuuden osuutta korosti käyttöliittymien muodostamistapa yhdistelmänä staattisia, palvelimella tuotettuja näkymiä ja selaimessa rikastettuja dynaamisia ominaisuuksia.

Deklaratiivisuudesta puhuttaessa tulisi erottaa puhtaasti deklaratiiviset kielet—kuten SQL—ja toisaalta tämän tutkimuksen näkökulmasta mielenkiintoisemmat sovelluskehykset. Käyttöliittymäohjelmoinnin kehitys on tapahtunut ensisijaisesti JavaScriptin kontekstissa, ja sen on mahdollistanut kielen joustavuus paradigmojen suhteen. Deklaratiivisuus on kiinteästi sidoksissa funktionaalisuuteen ja vallitsevien sovelluskehysten sisäinen kehitys heijastelee tätä sidosta. Reactin ensimmäiset versiot hyödynsivät luokkia komponenttien rakennuspalikoina siinä kuin muutkin aikalaisensa. Viimeisimmät versiot ovat kuitenkin siirtyneet vahvasti tilattomaan ja funktionaaliseen näkymien koostamistapaan—komponentit ovat pelkistyneet funktioiksi, jotka esittävät jonkun näkymän sovelluksen sen hetkisestä tilasta ilman sisäistä tilaa tai sovelluslogiikkaa. Ajatus käyttöliittymän muodostamisesta muuttuvan tilan funktiona ei ole uusi. Elliott ja Hudak (1997) esittelevät funktionaalisen reaktiivisen ohjelmoinnin (*functional reactive programming,* FRP) käsitteen tuomalla ajallisesti muuttuvat arvot osaksi funktionaalista ohjelmointia. Animaatioiden kontekstissa tapahtuvan tarkastelun voidaan nähdä vertautuvan suoraan käyttöliittymien muodostamiseen. Hanus (2000) tarkastelee Curry-ohjelmointikielen käyttämistä käyttöliittymäohjelmoinnissa, hyödyntäen sen funktionaalisia ominaisuuksia käyttöliittymän muodostamiseen ja logiikkaohjelmointia käyttöliittymän datan riippuvuuksien mallintamiseen. Tästä huolimatta toteutuksia ei ole juuri noussut valtavirtaan. Czapliskin ja Chongin (2013) esittelemää ELM-ohjelmointikieli, joka tuottaa selaimessa suoraan toimivaa JavaScriptiä, on saavuttanut jonkinasteista näkyvyyttä, mutta valtavirtaistuneeksi sitä on vaikea kutsua.

Syitä siihen, että puhtaan funktionaaliset kielet eivät kuitenkaan ole nousseet valtavirtaan, vaikka tämän hetkiset sovelluskehykset hyödyntävät vahvasti niiden mekanismeja, voi etsiä niin funktionaalisten kielten omaksuttavuudesta kuin pragmaattisista syistä. Kynnys ohjelmoimiseen JavaScriptillä on matalahko sen yleismaailmallisuuden ja ympäristöriippumattomuuden vuoksi ja sen syntaksi on suhteellisen tutun näköistä muihin kieliin tottuneille.

4.1 Ohjelmointimallien erot

Ohjelmoinnin oppimista ja siihen vaikuttavia tekijöitä on tutkittu jo pitkään (Sleeman, 1986) ja opetuksessa käytetyn kielen valinta on saanut erityishuomiota tässä tarkastelussa. Tästä huolimatta tutkimusasetelmat ovat heijastelleet aiemmin mainittua valtavirtaistuneiden ohjelmointikielien siirtymää saman

paradigman sisällä operoivien kielien välillä (Nikula, Gotel & Kasurinen, 2011). Paradigmojen välisistä eroista oppimisesta on vaikeahko löytää vertailevaa tutkimusaineistoa, joskin on viitteitä, että funktionaaliset kielet johtavat sovelluskehityksessä parempaan rakenteeseen (Joosten, Berg & Hoeven, 1993). Opetuskielet myös valikoitunevat vallitsevien valtavirtakielien perusteella pragmaattisista syistä, joten on ymmärrettävää, että koulutus on keskittynyt imperatiivisiin kieliin.

Ohjelmoinnin opiskeleminen itsessään ei ole yksinkertainen tehtävä. Ongelmia esiintyy lähtien ongelmien oikeellisesta tunnistamisesta ja niiden jakamisesta järkeviin osiin. Koodin lukeminen ja sisäistäminen on vaikeaa, ja samanaikainen koodin syntaksin ja semantiikan opiskeleminen, ongelmanratkaisutaitojen kehittäminen ja kaiken tämän soveltaminen on merkittävä haaste aloitteleville ohjelmoijille. Pitäen mielessä, että imperatiivisessa kontekstissa käyttöliittymäkehitykseen dynaamisluonteiset ongelmat—reaaliaikaisuus, reaktiivisuus ja tapahtumien hallinta—on nähty perinteisesti vaikeina ongelma-alueina myös kokeneille ohjelmistokehittäjille, niin ei ole yllättävää, että käytännön toteutuksissa ongelmat ovat lukuisia. Maier, Rompf ja Odersky (2010) arvioivat Adoben itse ilmoittamia virhemääriä Adoben työpöytäohjelmistoissa: kaikesta koodista kolmasosa käsittelee tapahtumien hallintaa, mutta noin puolet löytyneistä ohjelmointivirheistä aiheutuu näistä sovelluksen osa-alueista. Foust ym. (2015) tarkastelevat useamman suuren toimijan verkkosivuilta löytyviä yhteisiä komponentteja ja toteavat kaikista löytyvän epäjohdonmukaista toiminallisuutta.

Syitä edellä mainittuihin ongelmiin voidaan etsiä perinteisiin tapahtumakuuntelijoihin perustuvan kuuntelijamallin (*observer*) sisäsyntyisistä puutteista ja siitä syntyvistä vaikeasti hallittavista riippuvuuksista. Mallissa esittävät komponentit rekisteröivät kuuntelijan niiden sisältämän datan muutoksille, ja kuuntelijaa kutsuttaessa komponentti tarvittaessa reagoi muutoksiin. Vaihtoehtoisesti, ohjelmiston sisäiset mallit rekisteröivät kuuntelijan käyttöliittymässä tapahtuville muutoksille, ja muokkaavat omaa tilaansa kuuntelijoiden perusteella. Kuuntelija-malli johtaa helposti sivuvaikutuksiin ja kapselointiongelmiin kuuntelijoiden vaatiessa yhteistä tilaa. Kuuntelijat jakautuvat usein moniin eri komponentteihin, mikä johtaa helposti koostamisen mahdottomuuteen ja tekee toiminnallisuuden poistamisesta vaivalloista. Tämä johtaa myös vastuunjaon hämärtymiseen toisaalta yhden vastuualueen jakautuessa useampaan komponenttiin, toisaalta yksittäisten kuuntelijoiden vaikuttaessa useisiin vastuualueisiin. Vastuualueiden hämärtyminen johtaa nopeasti skaalautumisongelmiin ja koodin selkeyden heikkenemiseen. Abstraktiotaso on usein matala ja kuuntelijoiden täytyy epäoptimaalisesti pitää huolta resurssien käytöstä. Lisäksi kuuntelijoiden käyttäminen johtaa semanttiseen etäisyyteen, jossa ohjelmoijan tarkoitus ei välttämättä näy paikallisesta koodista tarkastelematta laajempaa kokonaisuutta (Maier, Rompf & Odersky, 2010).

Elliott ja Hudak (1997) vertailevat animaatioiden *mallinnusta* reaktiivisesti niiden imperatiiviseen *esitykseen* ja esittävät vertailussaan suoran yhteyden deklaratiivisten ja imperatiivisten ohjelmointikielten välillä. Deklaratiivisesta mallinnuksesta kumpuavien yleistettävien etujen, kuten selkeys, koostamisen helppous, laajennettavuus ja semanttinen puhtaus lisäksi artikkelissa listataan

sovellusalakohtaisia etuja. Ensimmäisenä—ja kognitiivisen kuorman jakautumisen kannalta suhteellisen mielenkiintoisena huomiona—nostetaan esille *authoring* tai *julkaiseminen* paremman käännöksen puutteessa. Tietyissä konteksteissa loppukäyttäjien kiinnostuksen kohteet ja osaaminen eivät välttämättä kohdistu siihen, *miten* ohjelmisto tuottaa lopputuloksen, vaan ennemmin siihen *mitä* pitäisi tuottaa. Vaikka Elliott ja Hudak käsittelevät animaatioiden tuottamista on tämä suoraan rinnastettavissa käyttöliittymien kehitykseen niiden asynkronisten, esitystapaan liittyvien ja samanaikaisten ominaisuuksien suhteen. Muita hyötyjä on optimoitavuus ja alustariippumattomuus, jotka kumpuavat esitystavan kuvaamisen ja toteutuksen erottamisesta, sekä säädeltävyys, jossa esittävä kerros pystyy helpommin määrittävään tarvittavan säätelyn tason (Elliott & Hudak, 1997).

Funktionaalinen ja funktionaalinen reaktiivinen ohjelmointi (FRP) on kuitenkin sisältänyt ongelmakohtia käytännön käyttöliittymätoteutuksissa. Merkittävänä ongelmana voi nähdä malleihin kiinteästi liittyvän tilattomuuden. Käyttöliittymillä on lähtökohtaisesti joku sovelluksen sisäisestä tilasta riippuva esitystila. Funktionaalinen reaktiivinen ohjelmointi ei pysty puhtaana vastaamaan käyttöliittymäkehityksen kaikkiin haasteisiin, vaikka se pystyykin kuvaamaan sovelluksen ajallisesti muuttuvien osien toiminnallisuuden (Jeltsch, 2016). Lisäksi ongelmia on esiintynyt suorituskyvyn suhteen. Elliottin ja Hudakin (1997) esittelemä Fran sisälsi signaalimallin, jossa sallittiin ajankohdasta vapaat riippuvuudet, mikä aiheutti vakavia suorituskykyongelmia ja suoritusajan myötä kasvanutta lineaarista muistinkulutusta (Czapliszki & Chong, 2013). Czapliski ja Chong (2013) käyvät läpi tähän ongelmakohtaan kehitettyjä ratkaisuja, joista kaikki asettavat joitain rajoituksia puhtaille FRP-toteutuksille, ja samassa artikkelissa esitelty ohjelmointikieli ELM sisältää oman tapansa vastata suorituskykyongelmiin. Tämän lisäksi käyttöliittymät toimivat harvoin tyhjiössä, joten niiden on pystyttävä hyödyntämään ympäristön tarjoamia kirjastoja ja rajapintoja, joista mittava osa on toteutettu imperatiivisessa kontekstissa. Ignatoff, Cooper ja Krishnamurthi (2006) tarjoilevat ratkaisuna olio-ohjelmointirajapintojen ja funktionaalisen reaktiivisen ohjelmoinnin yhdistämiseksi kokoelman yleiskäyttöisiä makroja, jotka laajentavat luokkien rajapintoja. Makrot ratkaisuna rajapintojen käyttämiseen on riippuvainen ohjelmointikielen ominaisuuksista ja artikkelissa kuvataan muistinkäsittelyn eroista syntyneitä ennustamattomia tilanteita ja suoritusjärjestyksessä esiintyneitä virheitä, joita varten piti kirjoittaa erillisratkaisut (Ignatoff, Cooper & Krishnamurthi, 2006). ELM hyödyntää erillistä *foreign function interface* -ratkaisua, joka toimii samalla tavalla siltana ulkoisten rajapintojen kanssa (Czapliszki & Chong, 2013).

Voitaneen suhteellisen turvallisesti todeta, että imperatiiviset toteutukset ovat sisäsyntyisesti epäoptimaalisia käyttäjän toimista aikasidonnaisesti muokkautuvan dynaamisen datan esittämiseen, mutta puhtaan funktionaaliset ratkaisut ovat olleet myös epätyydyttäviä. Funktionaaliset kielet ovat suhteellisen vaikeasti omaksuttavia: ongelmia esiintyy niin anonyymien kuin korkeamman asteen funktioiden omaksumisessa (Chakravarty & Geller, 2004), ja funktionaalisten ilmaisujen syntaksin ja semantiikan ymmärtäminen on aiheuttanut haasteita (Joosten ym., 1993; Ebrahimi, 1994). Kielten matemaattinen luonne voi tuntua vieraalta, jos kehittäjät eivät ole matemaattisesti orientoituneita ja ortodok-

sinen lähestymistapa puhtaaseen, tilattomaan funktionaalisuuteen monimutkaistaa tilanhallintaa merkittävästi.

4.2 Arkkitehtuurimallien erot

Esitystavan, datan ja sovelluslogiikan omiin komponentteihinsa eriyttävä *Model-View-Controller*-malli (MVC) ja sen variaatiot ovat pyrkineet vastaamaan niin sovelluksen sisäiseen vastuunjakoon, osa-alueiden kapseloimiseen ja näkymien koostamiseen ja esittämiseen. Vastuiden jako mallin osa-alueiden välillä on perinteisesti ollut seuraava:

- Kontrolleri muuntaa ja koostaa ympäristökohtaiset, käyttäjäinteraktiosta syntyvät tapahtumat sovelluskohtaisiksi metodikutsuiksi.
- Malli huolehtii näkymien koostamisesta ja sovelluslogiikasta.
- Näkymä esittää mallista saamansa tai hakemansa datan ja huolehtii tarvittavista transformaatioista.

Näistä lähtökohdista on havaittavissa vahva sidos mallin ja näkymän välillä, sekä potentiaalinen vastuunjaon epäselvyys. Vaikka MVC-malli olisikin riittävä yksinkertaisissa käyttötapauksissa, sen heikkoudet nousevat esiin monimutkaisissa käyttöliittymissä. Sovelluslogiikan sijoittaminen ja sen sitominen malleihin ei ole itsestään selvää järjestelmissä, joissa muutokset vaikuttavat useisiin komponentteihin. On myös helppo löytää tilanteita, joissa MVC-malli itsessään rikkoutuu omien rajoitteidensa seurauksina—esimerkiksi tilanteessa, jossa nimellisesti oikein suunnitellussa järjestelmässä näkymä tarvitsee useamman mallin tukea. Semanttisen etäisyyden kasvun myötä tuloksena on helposti vaikeasti ymmärrettävää ja seurattavaa sovelluskoodia, ja vahvojen sidosten myötä sen ylläpito on vaivalloista (Karagkasidis, 2008).

Komponenttipohjainen kehitys on konseptina vanha (McIlroy, 1968). Komponenttipohjaisissa arkkitehtuureissa johtoajatuksena on pyrkiä koostamaan ohjelmistot itsenäisistä, hyvin määritellyistä ja toisistaan riippumattomista komponenteista. Tästä kumpuava modulaarisuus, koostettavuus ja mahdollisuus dynaamiseen konfiguraatioon tekee niistä hyvin sopivia adaptiivisten käyttöliittymien kehitykseen (Alvares, Rutten & Seinturier 2017). Komponenttipohjainen kehitys soveltuu myös erityisen hyvin inkrementaaliseen sovelluskehitykseen, jossa ominaisuuksia lisätään pienissä osissa tarpeiden noustessa esille tai osana iteratiivista prosessia (Lau, Ng, Rana & Tran, 2012). React esitteli komponenttimallin, jossa näkymät tuotetaan osana komponentin sisäistä koodia ilman erillisiä näkymäpohjia ja jaottelua näkymän, mallin tai kontrollerin välillä ei tunneta—kaikki toiminnallisuudet voi ja tulisi toteuttaa komponentteina osana koostettua käyttöliittymää (Hunt, 2013). Näin syntyvä komponenttipuu on optimaalisesti puhdas funktio, joka tuottaa sovelluksen tilasta aina saman käyttöliittymän ilman sivuvaikutuksia. Tällä saavutetaan ylläkuvattuja hyötyjä: puhtaat funktiot ovat yksinkertaisia testata ja niiden koostaminen on helppoa, ja funktionaaliset komponentit yksinkertaistavat ominaisuuskokonaisuuksien kapselointia (Hunt, O'Shannessy, Smith & Coatta, 2016). Komponenttien sisäinen tilattomuus tukee tilan käsittelyä esitystavasta erillisenä, muuttu-

mattomana rakenteena, mikä vahvistaa deklaratiivisesta näkymien kuvauksesta saatavia hyötyjä.

Huolimatta suhteellisen merkittävästä määrästä komponenttipohjaisten arkkitehtuurien tutkimusta (Vale ym., 2015) lähes kaikki tutkimus ja käytännön sovellukset ovat kuitenkin tapahtuneet staattisluontoisemmissa taustajärjestelmissä. Tämän lisäksi, riippumatta arkkitehtuurimallin merkittävästi roolissa nykyaikaisessa käyttöliittymäkehityksessä edes lähiaikoina ei ole syntynyt juurikaan uutta tutkimusta mallin hyödyntämisestä käyttöliittymäkehityksessä. Tilattomien, funktionaalisten komponenttien käyttämisen lisääntyminen on kuitenkin tulkittavissa merkiksi siitä, että komponenttipohjainen arkkitehtuuri käyttöliittymissä toimii suunnitellusti ja oletetut hyödyt ovat todellisia. Kuitenkin, ongelmia näyttäisi olevan komponenttien rajojen löytämisessä—on havaittavissa yhtäältä toisteista koodia sisältäviä, toisaalta ylikapseloituja toteutuksia (Yang, Liu, Lin & Yu, 2018).

4.3 Datankuljetusmallit

Toisen polven käyttöliittymäsovelluskehyksissä datan sidonta näkymiin tehtiin pääsääntöisesti määrittelemällä deklaratiivisissa näkymäpohjissa ennalta määritellyin attribuutein näkymään sidottava sovelluksen tila. Tämä sidonta oli poikkeuksetta toteutettu kaksisuuntaisesti, siten että näkymä pystyi suoraan vaikuttamaan sovelluksen tilaan. Tämä on ollut pidetty ominaisuus: valtaosa Angular-kehittäjistä näkee kaksisuuntaisen datan sidonnan ja sen helpon toteutuksen hyvänä asiana (Ramos, Valente, Terra & Santos, 2016). Tämä on ymmärrettävää tarkastellessa käyttöliittymäkehityksen historiaa—ennen näitä kirjastoja yhtenäisiä malleja muuttuvan datan sitomiseen selaimen esittämään käyttöliittymään ei ollut ja tilamuutokset piti tehdä käsin, mikä oli virheherkkää ja työlästä.

React esitteli yksisuuntaisen datan sidonnan, jossa kaikki sovelluksen tilan muutokset tehdään eksplisiittisillä muutosfunktioilla (Hunt, 2013). Tietyssä mielessä Reactin malli on toisaalta vain näennäisesti datan sidontaa; ottaen huomioon Reactin tavan tuottaa sovellus sen hetkisen tilan funktiona, ei tilaa niinkään sidota näkymäpohjiin tai komponentteihin, vaan se annetaan esittävälle komponentille sitä kutsuttaessa sovelluksen uudelleentuottamisen yhteydessä (Hunt ym., 2016). Tästä seuraavat askeleet olivat Flux (Fisher & Chen, 2014), jossa data erotetaan omiin komponentteihinsa, ja kolmannen osapuolen Redux, jossa data nostetaan omalle ylätasolleen kokonaan erilleen esitystavasta. Molemmille näille on yhteistä tilan muuttumattomuus (*immutability*) ja se, että dataa muokataan ennalta määritellyillä toiminnoilla.

Datankuljetusmallien eroista käyttöliittymäkehityksessä on huomiota herättävän vähän julkaistua tutkimusta. Grenmyr & Magnusson (2016) tarkastelee valitun datankuljetusmallin vaikutusta ohjelmistojen ylläpidettävyyteen teknisten mittareiden kautta, mutta pitäviä tuloksia ei opinnäytteessä saavuteta. Tämän lisäksi datamallien eroja pitää käytännössä tarkastella kirjastojen omien lähtökohtien kautta ja niiden sisäisissä arkkitehtuureissa tapahtuneita muutok-

sia tutkimalla. Facebookin esitellessä Flux:n data-arkkitehtuurimalliksi React-sovelluksiin, perusteluina oli ensisijaisesti datan sidonnan kaksisuuntaisuudesta ja sen myötä monesta lähteestä saapuvien päivitysten seurauksena syntyvät vaikeasti ennustettavat ja kertautuvat sovelluksen tilan muutokset. Tämän nähtiin aiheuttavan huomattavia ongelmia käyttäjäinteraktioiden vaikutuksen arvioinnista sovelluksen kokonaistilaan (Fisher & Chen, 2014). Reduxin lähtökohtana on samat perustelut—järjestelmät, jotka toteuttavat datan sidonnan kaksisuuntaisena päätyvät ennen pitkää epädeterministisiksi ja tätä kautta vaikeasti ylläpidettäviksi ja testattaviksi. Lisäksi yleisten virheiden lähteeksi nähdään tilamutaatioiden ja asynkronisuuden yhdessä luoma vaikea hallittavuus, johon ratkaisuksi nähdään niiden erottaminen toisistaan säädellyllä tilanpäivitysmekanismilla vapaan kaksisuuntaisuuden sijaan (Redux, n.d.). Kilpailevien ohjelmistokehysten siirtymä yksisuuntaiseen datankuljetusmalliin antaa tukea Reactin ja Reduxin lähtökohdille. EmberJS:n kehittäjät eivät avaa muutoksen taustoja (Katz, 2015), mutta AngularJS:n kehittäjäyhteisö yhtyy yllämainittuihin perusteluihin: kaksisuuntainen datan sidonta aiheutti ongelmia kehyksen tarjoamassa sisäisessä muutoksenhallinnassa ja ratkaisuksi tarjotaan yksisuuntaista datan sidontaa (AngularJS, n.d.).

5 YHTEENVETO

Vuosina 2013-2019 tapahtunutta murrosta käyttöliittymäkehityksessä edelsi siirtymä palvelimella staattisesti tuotettujen käyttöliittymien rikastamisesta päätelaitteessa interaktiivisilla ominaisuuksilla käyttöliittymien tuottamiseen MVC-mallia, kaksisuuntaista datan sidontaa ja olio-ohjelmointiparadigmaa hyödyntävillä sovelluskehyksillä. Keskeisin piirre tälle siirtymälle oli käyttöliittymien tuottamisen siirtyminen päätelaitteisiin, ja tätä kautta aiempaa laajemmat mahdollisuudet aidosti dynaamisten käyttöliittymien toteuttamiseen. Tämän tutkielman tarkasteleman murroksen lähtöpisteenä voidaan pitää Reactin julkaisua, minkä seurauksena valtavirrassa suosituimmat sovelluskehykset ovat omaksuneet samat periaatteet. Tämä murros on todennettavissa tarkastelemassa kehittäjäyhteisön käyttämiä työkaluja ja olen nähdäkseni pystynyt osoittamaan, että kyseessä on aito ilmiö.

Reactin julkaisun taustalla oli korostetusti esillä tarve sovelluksen asynkronisten mutta samanaikaisten tilamuutosten ja käyttäjäinteraktioiden hallintaan. Aiemman tutkimuksen perusteella näyttäisi siltä, että imperatiivinen ohjelmointimalli ei sovellu hyvin näiden käyttöliittymäohjelmointiin erityisesti liittyvien ongelma-alueiden ratkaisemiseen. Deklaratiivisen—usein funktionaalisen tai reaktiivisen funktionaalisen—ohjelmointimallin edut temporaalisten ja asynkronisten muutosten hallinnassa ovat myös olleet tutkimusyhteisön tiedossa jo kauan. Tästä huolimatta funktionaaliset kielet tai sovelluskehykset eivät ole saavuttaneet suosita, eikä vieläkään voida puhua tästä: JavaScript on moniparadigmainen kieli, ja React tarjoaa vain ratkaisun vain osaan ongelmakentästä. Syitä tähän olen etsinyt yhtäältä kielten yleisestä käyttöasteesta, missä imperatiiviset kielet hallitsevat niin markkinoilla kuin koulutuksessa, toisaalta kielten omaksuttavuudesta: funktionaaliset kielet vaikuttaisivat olevan vaikeammin omaksuttavia johtuen niiden käsitteellisistä ominaisuuksista, tai mahdollisesti niiden matemaattisluonteisuudesta. Funktionaaliset käytännöt ovat kuitenkin ottaneet tukevan jalansijan sovelluskehityksessä ja näkisin että kehitys tulee jatkumaan samansuuntaisena vielä hyvän aikaa. Kehityssuunta vaikuttaa suhteellisen positiiviselta myös sovellusten rakenteen suhteen: funktionaalinen lähestymistapa tukee sovellusten komponenttipohjaista koostamista. Näkisin, että pyrkimys tilattomien komponenttien kirjoittamiseen ja uudet mekanismit,

kuten React 16.8:ssa esitellyt *hook* -funktiot tukevat luontevien komponenttirajojen löytämistä, parantaen sovellusten ylläpidettävyyttä ja koostettavuutta.

Sovelluksen tilan hallinnasta on vaikeampi sanoa mitään sitovaa johtuen aiemman tutkimuksen puutteesta tai sen soveltumattomuudesta ongelmakenttään. Vaikuttaisi siltä, että pystytään todentamaan, että funktionaalisilla kielillä saadaan tuotettua rakenteellisesti parempia käyttöliittymäsovelluksia. Viime aikojen siirtymä muuttumattomaan ja yksisuuntaisesti kuljetettavaan sovellustilaan liittyy läheisesti funktioiden puhtauteen, joka on funktionaalisten kielten perusominaisuuksia. Tätä kautta voisi argumentoida, että aiemmin todetut rakenteelliset edut voitaisiin ulottaa myös tilanhallintaan, mutta en pystynyt löytämään tätä oletusta käsittelevää tutkimusta. Kognitiivisen kuorman jakautumisen suhteen tutkimusta ei myöskään ole—siirtymä tilan irrottamiseen esittävistä komponenteista ja sen yksisuuntaiseen kuljetukseen voidaan kuitenkin nähdä merkkinä siitä, että kehittäjäyhteisö itse kokee tämän hyödylliseksi. Molemmat näkökulmat kaipaavat kuitenkin kipeästi lisätutkimusta.

Miksi murros käyttöliittymäohjelmoinnissa tapahtui juuri nyt ja nimenomaan JavaScriptin kontekstissa? Syyt ovat moninaisia, joskin liki poikkeuksetta pragmaattisia. JavaScriptin moniparadigmaisuus tekee siitä soveliaan eri ohjelmointimalleja hyödyntäville kielilaajennoksille ja sovelluskehyksille, mikä madaltaa kynnystä uusien lähestymistapojen iteratiiviseen kokeilemiseen. Näkisin myös, että sen ympärille kehittynyt ekosysteemi on luonteeltaan pioneerimainen—uusia käsitteitä esitellään nopeaan tahtiin ja vanhoja ei kaihdeta hylätä. Hyvänä esimerkkinä tästä on Reactissa itsessään tapahtunut siirtymä korkeamman tason komponenteista ns. *render props* -komponentteihin, ja edelleen *hooks*-mekaniikkaan. Tämä sisäinen siirtymä tapahtui kutakuinkin alle vuodessa, ja kirjastot ovat mukautuneet samaan tahtiin. Yhdessä tämä on johtanut siihen, että käytännön sovelluskehityksessä on pragmaattisesti yhdistelty toimivia osia hallitsemaan käyttöliittymätoteutusten eri osa-alueita. Reactin komponenttimalli ja yksisuuntainen tilakuljetus johti tilankäsittelyyn erikoistuneiden kirjastojen nousuun. Samalla kuitenkin jo olemassa olevia imperatiivisia toteutuksia on pystytty hyödyntämään muutosten rinnalla ilman merkittäviä ongelmia. Nähdäkseni JavaScriptin joustavuus ja paradigma-agnostisuus alustana onkin ollut viime vuosien uusien käsitteiden ja mallien laajan omaksunnan tärkeimpiä ajureita.

Yksi lähtökohdistani oli osoittaa, että uusi ohjelmointimalli tukee käyttöliittymäohjelmointiin erityisesti sopivaa mentaalista mallia uudelleenjakamalla kognitiivista kuormaa tilanhallinnan ja komponenttien esitystavan suhteen eri tavalla kuin aiemmat ohjelmointimallit. Tähän en kuitenkaan kyennyt, sillä tutkimusta kognitiivisen kuorman jakautumisen suhteen ei löydy käytännössä ollenkaan; tutkimusta ohjelmointityössä tapahtuvien kognitiivisten prosessien suhteen tuntuu olevan vaikea löytää yleisellä tasolla. Käsittelemäni tutkimus kuitenkin antaisi viitteitä siitä, että oletukseni on oikeilla jäljillä. Jatkotutkimuksen suhteen olisikin mielenkiintoista, että miten käytännön ohjelmointityössä ohjelmoijan fokus ja kognitiivinen kuorma jakautuu sovellusten eri osa-alueiden suhteen ja miten eri ohjelmointimallit ja -tavat vaikuttavat jakautumiseen.

LÄHTEET

Alvares, F., Rutten, E. & Seinturier, L. (2017) A domain-specific language for the control of self-adaptive component-based architecture. *The Journal of Systems and Software 130* (94-112).

AngularJS (n.d.). Angular 1 and Angular 2 integration: the path to seamless upgrade [blogiteksti]. Haettu 22.1.2020 osoitteesta http://blog.angularjs.org/2015/08/angular-1-and-angular-2-coexistence.html

Cerny, T., Chalupa, V. & Donahoo, M.J. (2012). Impact of user interface generation on maintenance. Teoksessa *Proceedings, 2012 IEEE International Conference on Computer Science and Automation Engineering* (621-625).

Chakravarty, M. & Keller, G. (2004) The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming, 14 (1)* (113-123).

Czapliski, E. & Chong, S. (2013). Asynchronous functional reactive programming for GUIs. *ACM SIGPLAN Notices*, June 2013.

Ebrahimi, A. (1994) Novice programmer errors: Language constructs and plan composition. *International Journal of Human Computer Studies, 41(4)* (457-480).

Elliott, C. & Hudak, P. (1997) Functional reactive animation. Teoksessa *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming* (263–273). New York: ACM

Fisher, B. & Chen, J. (6.5.2014). Flux: An Application Architecture for React [blogikirjoitus]. Haettu osoitteesta https://reactjs.org/blog/2014/05/06/flux.html

Foust, G., Järvi, J. & Parent S. (2015). Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. Teoksessa *GPCE 2015: Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (121–130). New York: ACM

Grenmyr, D. & Magnusson, E. (2016). *An Investigation of Data Flow Patterns Impact on Maintainability When Implementing Additional Functionality.* (bachelor/master thesis). Linnaeus University.

Hanus, M. (2000). A Functional Logic Programming Approach to Graphical User Interfaces. Teoksessa *PADL '00: Proceedings of the Second International*

Workshop on Practical Aspects of Declarative Languages (47-62). Berlin: Springer-Verlag.

Hunt, P. (5.6.2013). Why did we build React? [blogikirjoitus]. Haettu osoitteesta https://reactjs.org/blog/2013/06/05/why-react.html

Hunt, P., O'Shannessy, P., Smith, D. & Coatta, T. (2016). React: Facebook's Functional Turn on Writing JavaScript. *Communications of the ACM 59(12)* (56-62). New York: ACM.

Ignatoff, D., Cooper, G. & Krishnamurthi, S. (2006). Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages. Teoksessa *Functional and Logic Programming. FLOPS 2006. Lecture Notes in Computer Science 3945* (259-276). Berlin: Springer.

Jeltsch, W. (2016). Abstract categorical semantics for resourceful functional reactive programming. Teoksessa *Journal of Logical and Algebraic Methods in Programming 85(6)* (1177-1200). Amsterdam: Elsevier.

Joosten, S., Berg, K. & Hoeven, G. (1993). Teaching functional programming to first-year students. *Journal of Functional Programming, 3(1)* (49-65).

Karagkasidis, A. (2008). Developing GUI Applications: Architectural Patterns Revisited. Teoksessa *EuroPLoP 2008: 13th Annual European Conference on Pattern Languages of Programming Vol. 610*. Irsee, Germany, July 9-13, 2008.

Katz, Y. (10.5.2014). The Transition to Ember 2.0 in Detail [blogikirjoitus]. Haettu osoitteesta https://blog.emberjs.com/2015/05/10/run-up-to-two-oh.html

Lau, K., Ng, K., Rana, T. & Tran, C. (2012). Incremental construction of component-based systems. Teoksessa *CBSE '12: Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering* (41-50). New York: ACM

McIlroy, D. (1968). Mass-produced software components. Teoksessa *NATO Conference* (88–98).

Maier, I., Rompf, T., & Odersky, M. (2010). *Deprecating the Observer Pattern* (EPFL-REPORT-148043). Lausanne: EPFL.

Myers, B. (1993). *Why are Human-Computer Interfaces Difficult to Design and Implement?* (1993 Technical Report). Pittsburgh: Carnegie Mellon University.

Nikula, U., Gotel, O. & Kasurinen, J. (2011). A Motivation Guided Holistic Rehabilitation of the First Programming Course. Teoksessa *ACM Transactions on Computing Education 11(4)*. New York: ACM.

Ramos, M., Valente, M., Terra, R. & Santos, G. (2016). AngularJS in the Wild: A Survey with 460 Developers. Teoksessa *PLATEAU 2016: Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools* (9-16). New York: ACM.

Redux (n.d.). *Motivation* (Verkkosivu). Haettu 22.1.2020 osoitteesta https://redux.js.org/introduction/motivation

Sleeman, D. (1986). The challenges of teaching computer programming. *Communications of the ACM 29(9)* (840–841).

Stankiewicz, A. (2.10.2017). How to migrate away from Bower? Haettu 14.1.2020 osoitteesta https://bower.io/blog/2017/how-to-migrate-away-from-bower/

The Joint Task Force on Computing Curricula Association for Computing Machinery (ACM) IEEE Computer Society (2013). Computer Science Curricula 2013. Haettu osoitteesta https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf

The Joint Task Force on Computing Curricula IEEE Computer Society Association for Computing Machinery (2001). Computing Curricula 2001. Haettu osoitteesta https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2001.pdf

Vale, T., Crnkovic, I., Almeida, E., Neto, P., Cavalcanti, Y. & Meira, S. (2015). Twenty-eight years of component-based software engineering. Teoksessa *The Journal of Systems and Software 111* (128-148). Amsterdam: Elsevier.

Yang, C., Liu, Y., Lin, Y. & Yu, J. (2018). Leveraging the Power of Component-based Development for Front-End Components: Insights from a Study of React Applications. *The 30th International Conference on Software Engineering and Knowledge Engineering*. Redwood, CA, July 2018.