Ville Virtanen

REAKTIIVISEN OHJELMOINTIPARADIGMAN SOVELTAMINEN ANGULARISSA

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittelyn koulutus

2023

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Tradenomi (AMK)
Tekijä	Ville Virtanen
Työn nimi	Reaktiivisen ohjelmointiparadigman soveltaminen Angularissa
Toimeksiantaja	CGI Suomi Oy
Vuosi	2023
Sivut	44 sivua
Työn ohjaaja	Janne Turunen

TIIVISTELMÄ

Reaktiivisella ohjelmointiparadigmalla tarkoitetaan tapahtumapohjaista interaktiivisten sovellusten ohjelmointiin käytettyä ohjelmointimallia. Reaktiivinen ohjelmointi on useimmiten myös deklaratiivista sekä funktionaalista ja sitä voidaan ajatella vastakkaisena ohjelmointityylinä perinteisemmälle imperatiiviselle ja proseduraaliselle ohjelmoinnille. Reaktiivisessa ohjelmoinnissa muuttujien arvot on sidottu toisiinsa, ja ne reagoivat muutoksiin sekä päivittyvät automaattisesti.

Tässä opinnäytetyössä tutkittiin reaktiivista ohjelmointia ja esiteltiin sen käyttämistä Angular-ohjelmointikehyksessä. Angular on interaktiivisten verkkosovellusten kehittämiseen luotu viitekehys, joka tarjoaa monia reaktiivisen ohjelmoinnin mahdollistavia työkaluja. Näistä tärkeimpänä lienee RxJS-kirjasto, joka on kokoelma reaktiiviseen ohjelmointiin JavaScriptillä suunniteltuja välineitä. Opinnäytetyössä refaktoroitiin olemassa olevan Angular-sovelluksen toiminnallisuus, joka oli aiemmin toteutettu imperatiivisesti ja proseduraalisesti.

Toimeksiantajana opinnäytetyölle toimi CGI Suomi Oy, joka toteuttaa IT-ratkaisuja julkiselle ja yksityiselle sektorille. Toimeksiannon tarve syntyi halusta varmistaa ja parantaa sovelluksen frontend-koodin laatua. Angular soveltuu erinomaisesti reaktiiviseen ohjelmointiin, mutta valitettavan pieni osa sovelluskehittäjistä on sisäistänyt reaktiivisen ohjelmoinnin idean ja käyttää Angularin ja RxJS:n työkaluja epäoptimaalisesti. Työn yhtenä tavoitteena oli osoittaa, miten reaktiivisesti kirjoitettu ohjelmakoodi tekee siitä luettavampaa, ymmärrettävämpää, yksinkertaisempaa ja muokattavampaa nostaen huomattavasti koodin laatua. Täten omaksumalla reaktiivisen ohjelmointimallin on toimeksiantajan mahdollista saavuttaa merkittäviä koodin laadullisia parannuksia.

Opinnäytetyön teoreettinen viitekehys muodostui reaktiivisen ohjelmoinnin määritelmien tutkimisesta sekä arvioimisesta ja Angularissa käytettävien reaktiivisten työkalujen esittelemisestä. Lisäksi käsiteltiin tyypillisiä reaktiivisten työkalujen väärinkäytöksiä, joita tulisi pyrkiä välttämään. Käytännön osuudessa tutkittiin vanhaa koodipohjaa pyrkien löytämään siitä mainitut väärinkäytökset. Nämä osuudet koodista kirjoitettiin uudelleen reaktiivisesti. Lopputuloksena on reaktiiviseksi refaktoroitu toiminnallisuus, jota toimeksiantaja voi käyttää malliesimerkkinä sovelluksen jatkokehityksessä.

Asiasanat: ohjelmistokehitys, ohjelmistoarkkitehtuuri, ohjelmistokehys, reaktiivinen ohjelmointi, web-ohjelmointi, Angular, RxJS

South-Eastern Finland
University of Applied Sciences

Degree title	Bachelor of Business Administration
Author	Ville Virtanen
Thesis title	Reactive Programming in Angular
Commissioned by	CGI Suomi Oy
Time	2023
Pages	44 pages
Supervisor	Janne Turunen

ABSTRACT

Reactive programming paradigm is an event-based programming model used to program interactive applications. Reactive programming is usually also declarative and functional, and it can be thought of as the opposite style of programming to more traditional imperative and procedural programming. In reactive programming, the values of variables are bound to each other, and they react to changes or events and update automatically.

This thesis studied reactive programming and introduced its use in the Angular framework. Angular is a framework created to develop interactive web applications, and it uses reactive programming ideology in many of its functions. It heavily relies on the RxJS library, which is a collection of tools that enable reactive programming with JavaScript. In this thesis an Angular application's functionality was refactored from previous imperative style to follow the reactive programming paradigm.

The thesis was commissioned by CGI Suomi Oy, a producer of IT solutions for the public and private sector. The need for the work arose from the desire to ensure and improve the quality of the application's frontend code. Angular is very well suited for reactive programming, but an unfortunately small portion of developers has embraced the idea of reactive programming. Consequently, most developers don't use the tools of Angular and RxJS optimally. One of the goals of the work was to show how reactive code would be more readable, understandable, simpler, and editable, significantly increasing the quality of the code. By adopting the reactive programming model, it is possible for the commissioner to achieve significant improvements in code quality.

The theoretical framework of the thesis consisted of evaluating the definitions of reactive programming and introducing the reactive tools used in Angular. In addition, the thesis discussed typical abuses of these tools that developers should try to avoid. In the practical part, the old code base was refactored so that such abuses were rewritten reactively. The result was the previous functionality reactively refactored. The commissioner can use this refactored code as an example in the further development of the application.

Keywords: software development, software architecture, software framework, reactive programming, web development, Angular, RxJS

SISÄLLYS

1 JOHDANTO 5

2 REAKTIIVINEN OHJELMOINTI JA ANGULAR 6

2.1 Reaktiivinen ohjelmointiparadigma 6

2.2 Angular-ohjelmistokehys 9

2.3 RxJS – Reactive Extensions for JavaScript 12

3 ANGULAR SOVELLUKSEN REFAKTOROINTI REAKTIIVISEKSI 20

3.1 RxJS:n käyttäminen Angularissa 20

3.2 Refaktoroitava toiminnallisuus ja sen imperatiivinen toteutus 21

3.3 Reaktiiviseksi refaktoroitu palvelu ja komponentit 33

4 PÄÄTÄNTÖ 40

LÄHTEET 42

1 JOHDANTO

Reaktiivinen, deklaratiivinen ja funktionaalinen ovat iskusanoja, joita ohjelmointia käsittelevät blogit ja opetusvideot vaikuttavat olevan tällä hetkellä tulvillaan. Usein nämä termit jäävät kuitenkin merkitykseltään ontoiksi ja konkreettisten esimerkkien puutteen vuoksi näiden ohjelmointityylien erot imperatiiviseen ja proseduraaliseen ohjelmointiin voivat jäädä epäselviksi. Ohjelmoinnissa sama lopputuotos voidaan yleensä saavuttaa useammalla eri tavalla toteutettuna. Yksi vaikeimmista päätöksistä sovelluksen ohjelmoinnin alkuvaiheessa onkin valinta toteutukseen parhaiten soveltuvista työkaluista ja ohjelmointimalleista. Jokaisella ohjelmoijalla lieneekin oma lempilapsensa lukuisten eri ohjelmointimallien, -kielten ja -viitekehysten joukosta. Tässä opinnäytetyössä avaamme reaktiivisen ohjelmointiparadigman käsitettä ja sen suhdetta toisaalta deklaratiiviseen ja funktionaaliseen ohjelmointiin ja toisaalta sen eroja imperatiiviseen ja proseduraaliseen ohjelmointiin. Lisäksi pyritään osoittamaan hyötyjä, joita reaktiivinen ohjelmointitapa voi tuoda sovelluksen luettavuuteen ja ymmärrettävyyteen.

Opinnäytetyön tavoitteena on toimeksiantajayritys CGI Suomi Oy:n toteuttaman verkkosovelluksen yhden osan refaktorointi. Refaktoroinnilla tarkoitetaan sovelluskoodin uudelleen kirjoittamista sen toiminnallisuutta muuttamatta. Refaktoroitavaksi osaksi valikoitui yksi sovelluksessa käytetty Angularin palvelu sekä komponentti. Nämä termit määritellään tarkemmin teoreettisessa osassa, mutta tässä kohtaa riittänee se, että mielletään tämä kohteena oleva osa (palvelu ja komponentti) yhtenä sovelluksen toiminnallisuutena.

Sovellus on toteutettu käyttämällä Angular-ohjelmistokehystä ja refaktoroinnissa tarkoituksena on hyödyntää reaktiivista ohjelmointimallia ja Angularin sekä RxJS-kirjaston reaktiivisia työkaluja. Teoreettisessa osuudessa luvussa kaksi tutkitaan reaktiivista ohjelmointiparadigmaa yleisesti sekä sen käyttöä Angular-sovelluksissa RxJS-kirjastoa hyödyntäen. Teoriaperusta muodostuu kirjallisista lähteistä sekä reaktiivista ja imperatiivista koodia vertailevista esimerkeistä. Teoriaosuuden tavoitteena on tuoda esille reaktiivisen ohjelmoinnin tuottamat konkreettiset hyödyt, jotka perustelevat työn käytännön osuuden tarpeellisuutta.

Käytännön kehittämisongelmana on refaktoroida olemassa olevan sovelluksen yksi toiminnallisuus (*feature*) käyttäen reaktiivista ohjelmointityyliä. Nykyinen koodipohja on osittain reaktiivinen ja osittain imperatiivinen, joten refaktoroinnin kohteeksi valikoitui toiminnallisuus, joka on toteutettu epäoptimaalisesti ja jonka refaktoroinnista on todellista hyötyä toimeksiantajalle. Refaktorointi ja tehdyt muutokset on raportoitu luvussa kolme, jossa vanhaa ja uutta koodia vertailemalla on esitetty, miten reaktiivinen koodi parantaa luettavuutta ja tekee koodin helpommaksi ymmärtää.

Lopuksi arvioidaan sitä, miten tässä johdannossa esitetyt tavoitteet on saavutettu ja millaisia mahdollisia ongelmia työn aikana on kohdattu. Päätännössä tarkastellaan myös sitä, millaista hyötyä toimeksiantaja on saanut työstä ja onko työssä päästy toimeksiantajan asettamiin tavoitteisiin. Lisäksi pohditaan, missä määrin on mielekästä sitoutua täysin reaktiiviseen ohjelmointiin ja milloin olisi tarkoituksenmukaisempaa rikkoa rajoja eri ohjelmointimallien välillä.

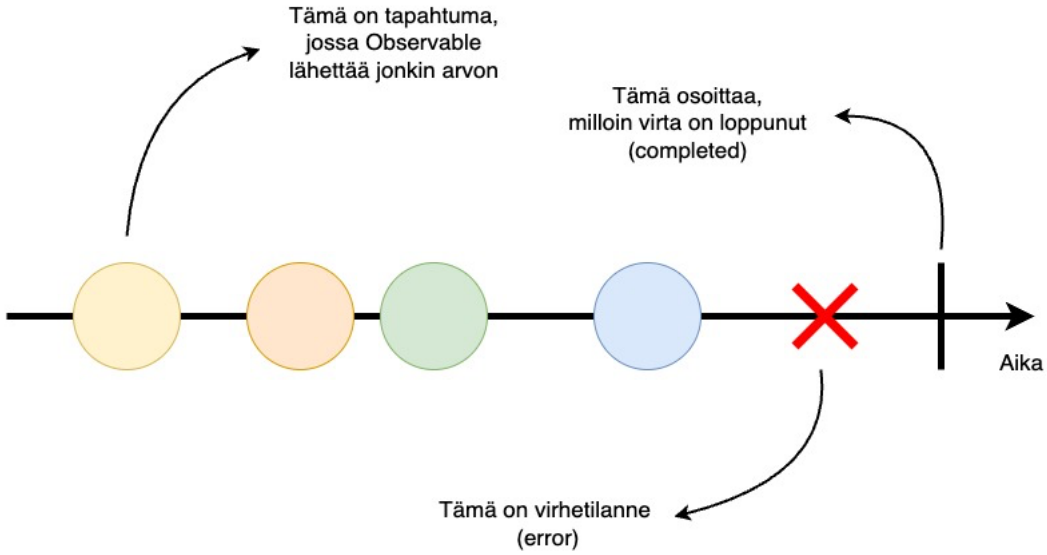
2 REAKTIIVINEN OHJELMOINTI JA ANGULAR

2.1 Reaktiivinen ohjelmointiparadigma

Reaktiivista ohjelmointia (*Reactive Programming*) on vuosien saatossa pyritty määrittelemään useaan eri otteeseen ja erilaisia määritelmiä löytyykin perinpohjaisuudeltaan laidasta laitaan. Esimerkiksi Stovell (2010) on esittänyt löyhän määritelmän reaktiivisesta ohjelmoinnista niin kutsutun kohtalo-operaattorin (*destiny operator*) käyttämisenä. Tämä kohtalo-operaattori on operaattori, joka sitoo muuttujan arvon jonkin toisen muuttujan arvoon. Tällä hän tarkoittaa, että näiden kahden muuttujan ”kohtalot” ovat toisistaan riippuvaisia (tai oikeammin toisen muuttujan b kohtalo riippuu ensimmäisen muuttujan a kohtalosta). Tässä ajatusmallissa siis a:n muuttuessa seuraa välttämättä vastaava muutos b:ssä. Myös esimerkiksi Carniato (2021) jakaa vastaavan löyhän käsityksen ja määrittelee reaktiivisen ohjelmoinnin idean kiteytyvän deklaratiiviseen ilmaukseen ajassa vaihtuvien arvojen suhteesta (*The declarative expression of the relationship between values that change over time*).

Toista ääripäätä tällä reaktiivisen ohjelmoinnin määrittelyasteikolla edustaa Conal Elliottin urallaan kehittämä semantiikka **funktionaalisesta reaktiivisesta ohjelmoinnista** (*Functional Reactive Programming – FRP*). Hän on pyrkinyt määrittelemään matemaattisella tarkkuudella funktionaalisen reaktiivisen ohjelmoinnin reunaehdot ja osoittamaan, miten tietokoneita voidaan ohjelmoida käyttäen tätä kuvattua tekniikkaa. Hänelle tärkeimpinä FRP:tä kuvaavina tekijöinä ovat ohjelman denotatiivisuus (*denotative*) ja jatkuva ajallisuus (*continuous time*). Näillä käsitteillä ymmärretään yksinkertaistaen sitä, että ohjelmissa dataa käsitellään funktioiden avulla, jotka yksiselitteisesti kuvaavat niiden aikaansaaman muutoksen datassa. Lisäksi dataa ajatellaan ajassa jatkuvasti muuttuvana arvona, eikä eri ajanhetkinä muuttuvina yksittäisinä arvoina. Analogiana tälle ajatukselle ajassa muuttuvista arvoista toimii hyvin bittikartta- ja vektorigrafiikan erot toisiinsa nähden. Muissa reaktiivisen ohjelmoinnin määritelmissä datavirta on usein ajateltu ajassa muuttuvina arvoina – kuten virran mukana kulkevina esineinä tai asioina (bittikarttagrafiikka), eikä tällaisena katkeamattomana arvojen virtana (vektorigrafiikka). (Krouse 2019.)

Tätä yleisempää ajatusta datasta ajassa muuttuvina arvoina kuvaavat myös reaktiivisen ohjelmoinnin dokumentaatiossa yleiset marmoridiagrammit, kuten kuvassa 1 esitetty diagrammi. Tässä kuvassa dataa sisältävä Observable-olio lähettää tai emittoi eri arvoja ajan kuluessa. Tässä ei tarkalleen ottaen ole kyse datavirrasta sillä kyseessä ei ole niin kutsuttu jatkuva aika siinä mielessä, jossa esimerkiksi Elliott termiä käyttää. Näissä tiukemmissa määritelmissä mennään tämän työn aihetta syvemmälle koneohjelmointiin, joten tässä yhteydessä ei oletetakaan reaktiiviselta ohjelmoinnilta näin tarkkoja reunaehtoja – ja kyseisistä Observableista käytetään datavirta-termiä sen vapaammassa merkityksessä. Lisäksi Elliott (2011) itse ei pidä funktionaalista reaktiivista ohjelmointia tätä määrittelemäänsä ohjelmointimallia parhaiten kuvaavana terminä, vaan mieltää kuvaavammaksi lyhenteen DCTP (Denotational Continuous-Time Programming). Samaan ajatukseen yhtyy myös Staltz (2015) julistaessaan, että Reactive eXtensions -kirjasto on FRP:tä (Functional Reactive Programming).

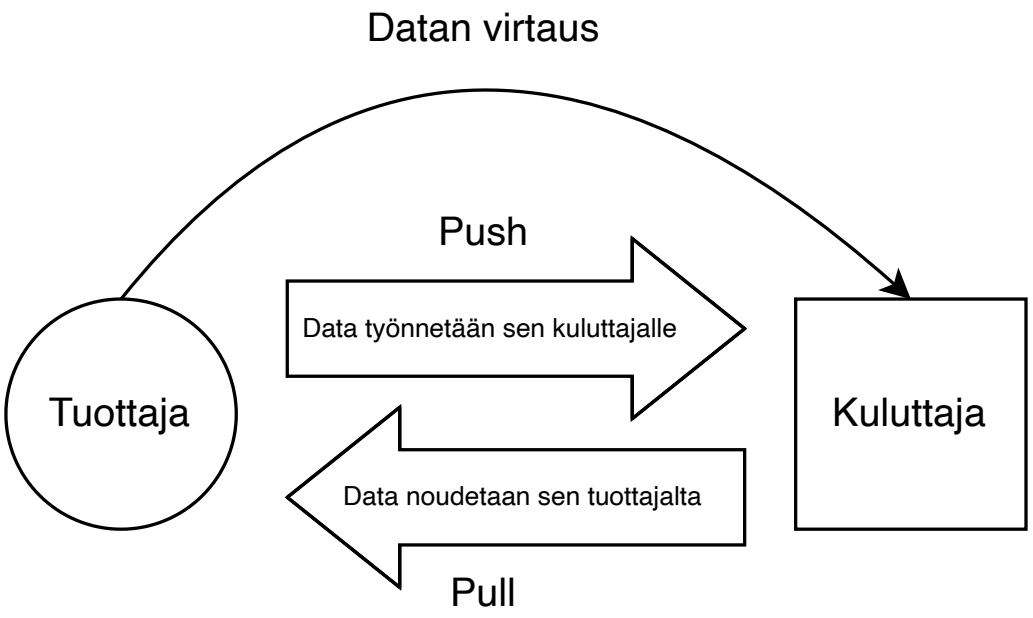


Kuva 1. Observable-datavirta, joka lähettää (*emit*) arvoja ajan kuluessa (mukaillen Staltz 2014)

Useimmiten reaktiivista ohjelmointia kuvaillaan kuitenkin ajassa muuttuvien **datavirtojen** (*stream*) avulla. Yksi tunnetuimmista reaktiivisen ohjelmoinnin esittelyteksteistä lienee Staltzin (2014) artikkeli The introduction to Reactive Programming you've been missing, jossa hän esittää määritelmän, että reaktiivinen ohjelmointi olisi ohjelmointia asynkronisilla datavirroilla. Tämän lisäksi Staltzin (2014) mukaan reaktiiviseen ohjelmointiin kuuluu funktionaalinen ulottuvuus, eli näitä datavirtoja voidaan muokata funktionaalisen ohjelmoinnin tyyliin esimerkiksi ReactiveX-kirjaston avulla. Ajatusta datasta virtoina voi pitää niinkin perustavanlaatuisena, että reaktiivisen ohjelmoijan mantrana tulisi Staltzin (2014) mukaan olla: ”kaikki data on virtoja (*Everything is a stream*)”.

Reaktiivista ohjelmointia voidaan siis kuvailla tai yrittää määritellä hieman eri lähtökohdista. Bainomugisha ym. (2013) ovat tehneet kattavan katsauksen erilaisiin ohjelmointikieliin ja -kirjastoihin, jotka mieltävät itsensä reaktiivisiksi. Yhteistä näille kielille ja kirjastoille vaikuttaa olevan se, että niissä käytetään ajallisesti muuttuvia arvoja ja arvojen muutokset etenevät ohjelman sisällä ikään kuin automaattisesti (*propagation of change*). Eroja kielten välillä löytyy esimerkiksi siinä, millaisia nämä ajalliset arvot ovat: käytöksiä (*behavior*), tapahtumia (*event*), signaaleja (*signal*) ja niin edelleen, sekä siinä etenevätkö muutokset työntämällä (*push*) vai noutamalla (*pull*), kuten on esitetty kuvassa 2. Lyhyesti voitaneen tiivistää, että käytökset ovat jatkuvia ajallisia arvoja ja tapahtumat taas eri ajanhetkinä lähetettäviä arvoja, joita kuvaa esimerkiksi ku-

vassa 1 esitetty Observable-datavirta. Datan muutosten etenemismalli on pitkälti riippuvainen ohjelmointikielestä ja sen asettamista reunaehdoista. Kaikki innokkaat kielet (*eager languages*) käyttävät työntämiseen perustuvaa etenemismallia, kun laiskat kielet (*lazy languages*) voivat käyttää jompaakumpaa tai kumpaakin mallia yhdessä. (Bainomugisha ym. 2013, 2–6.)



Kuva 2. Muutoksien eteneminen ohjelmassa **tuottajalta** (*producer*) **kuluttajalle** (*consumer*); push ja pull -etenemismallien erottelu (mukaillen Bainomugisha ym. 2013, 5)

Tämän työn piirissä reaktiivisella ohjelmointiparadigmalla tarkoitetaan Staltzin ajatusta lähellä olevaa määrittelyä, jota myös ReactiveX-projektissa käytetään. Tämän määritelmän mukaan reaktiivinen ohjelmointi on ohjelmointia ajallisilla tapahtumista koostuvilla datavirroilla, joita voidaan muokata funktionaalisesti. Tämä rajaus on tehty siksi, että Angularissa suuressa roolissa on nimenomaan RxJS (*Reactive Extensions for JavaScript*) -kirjasto, jonka avulla Angular-sovelluksia voidaan ohjelmoida reaktiivisesti. ReactiveX-kirjastoja on kirjoitettu useille eri ohjelmointikielille, mutta tämän työn piirissä keskitytään sen JavaScript-kieliversioon RxJS:ään, jonka ominaisuuksiin palaamme myöhemmässä sitä käsittelevässä luvussa.

2.2 Angular-ohjelmistokehys

Angular on Googlen julkaisema ohjelmistokehys web-sovelluksien ohjelmoinnin helpottamiseksi. Sen avulla voidaan kehittää verkkosovelluksia käyttäen

TypeScriptia ja HTML:ää. Angular sai alkunsa vuosien 2008 ja 2009 aikana tuolloin Googlella toimineen Miško Heveryn sivuprojektina. Tämän projektin tarkoituksena oli luoda ohjelmoijille (ja hänelle itselleen) työkalu, jonka avulla web-sovellusten kehittäminen olisi helpompaa. Angular – jota alkuvaiheessaan kutsuttiin AngularJS:ksi – sai nimensä HTML-kielessä käytetyistä kulmasulkeista (< ja >). (Gavigan 2018.)

Virallisesti Google otti Angularin kehityksen omaksi projektikseen vuonna 2010, ja julkaisi ensimmäisen version AngularJS:sta markkinoille. AngularJS:n versionumero 1.x:n kehitys jatkui, kunnes kilpailu viitekehysten kesken koveni ja etenkin Facebookin kehittämä React-ohjelmistokehys alkoi vallata markkinaa. Tällöin Angularin kehittäjätiimille tuli selväksi, että AngularJS:n tarjoamat mahdollisuudet ovat rajalliset ja viitekehys vaatisi kattavan uudelleenkirjoituksen, jonka jälkeen uusi versio ei enää olisi yhteensopiva vanhan kanssa. Tämän takia AngularJS brändättiin uudelleen siten, että nykyään se tunnetaan pelkästään nimellä ”Angular”, ja uusi versionumero 2.x julkaistiin 2016. (Gavigan 2018.) Vanhan AngularJS:n viimeisin – vuonna 2022 julkaistu – versio on versionumero 1.8.3, ja sen kehitys on virallisesti lopetettu.

Uuden Angularin versionumerointi on semanttinen ja tällä hetkellä (5.11.2023) viimeisin Angular versio on numero 16.2.12. Angularin kehitys jatkuu aktiivisesti ja uusissa versioissa on mukana paljon käyttäjien toivomia parannuksia ja uusia ominaisuuksia viitekehykseen. Viimeaikaisia uudistuksia ovat esimerkiksi Angularin Twitterissä @angular-tilillään (2023) mainostamat itsesulkeutuvat komponenttitagit (kuten *<angular/>)* ja etenemissuunnitelmassaan (2023a) mainitsemat itsenäiset komponentit, signaalit, parannukset hydraatioon ja sisällön renderöintiin palvelimella sekä tiukka lomakkeiden tyypitys.

Angularilla kehitetyt sovellukset ovat niin kutsuttuja yksisivuisia sovelluksia (*Single Page Application*), joissa käyttäjä lataa palvelimelta vain yhden HTML-verkkosivun, jota päivitetään sen mukaan, mitä sivun rakennusosia käyttäjälle halutaan näyttää. Tällöin vain osaa sivusta voidaan päivittää, kun siinä tapahtuu muutoksia, eikä koko sivua tarvitse ladata uudelleen palvelimelta. Yksisivuisissa sovelluksissa ei myöskään ole eri reiteille eri HTML-tiedostoja, vaan reitin vaihtuessa käyttäjälle näytetään ne komponentit, jotka on määritelty tässä reitissä näytettäviksi. (Thakur 2021.)

Rakenteeltaan Angular-sovellukset ovat ikään kuin osista koostuvia palapelejä. Näitä osia Angularissa kutsutaan näkymiksi. Näkymät koostuvat TypeScript kielellä kirjoitetusta **komponentista** (*Component*) ja **HTML-templaatista** (*Template*), ja ne ovat aina osa yhtä moduulia (*NgModule*). Moduulit lienee helpointa ajatella paketeiksi, jotka määrittävät, mitä riippuvuuksia, importeja ja exporteja paketin komponenteilla on. (Thakur 2021.) Yleisesti näkymistä käytetään nimitystä komponentti, vaikka sillä voidaan viitata myös pelkkään TypeScript-tiedostoon. Tässäkin työssä sanalla komponentti tullaan yleensä tarkoittamaan näkymän kokonaisuutta, joka sisältää niin HTML-tiedoston kuin sitä vastaavan TypeScript-tiedostonkin.

Angularin uusissa itsenäisissä komponenteissa on pyritty selkiyttämään komponenttien ja moduulien rakennetta poistamalla moduulit ja määrittelemällä kunkin komponentin vaatimat riippuvuudet komponentissa itsessään – eikä siis moduulissa, jonka osa komponentti on. Komponenttien määrittelemiä näkymiä voidaan yhdistää halutulla tavalla, jolloin muodostetaan elementtipuu, josta kulloinkin käyttäjälle näytettävä verkkosivu koostuu. Komponentin TypeScript-tiedoston tehtävänä on toimia kontrollerina templaatille ja välittää siihen dataa ja metodeja, jotka sidotaan templaatissa käytettäviin muuttujiin ja tapahtumiin. (Angular 2022a.)

Edellä esitettyä mallia voinee avata ajatus, että käyttäjän tekemät klikkaukset, hiiren liikkeet ja niin edelleen ovat tapahtumia, jotka sidotaan komponentin johonkin metodiin. Nämä metodit muuttavat jollakin tavalla komponentissa määriteltyjä muuttujia, jotka on sidottu templaatin sisältöön. Esimerkiksi p-elementin sisältönä oleva teksti, elementille määritelty CSS-luokka tai vaikkapa input-elementin teksti voidaan sitoa muuttujaan ja niitä voidaan muuttaa ohjelmallisesti erilaisiin tapahtumiin reagoiden. Tällä tavoin voidaan ohjelmoida interaktiivisia sovelluksia, joissa käyttäjän toimintoihin reagoidaan ja näkymää päivitetään sen mukaan, mitä tapahtumia käyttäjä sivulla laukaisee.

Yhtenä tärkeänä ominaisuutena Angularissa ovat myös **palvelut** (*Service*) ja niiden **injektoiminen** (*Dependency Injection*) komponentteihin. Palvelut ovat TypeScript-tiedostoissa määriteltyjä luokkia, jotka on merkitty *@Injectable* dekoraattorilla. Palveluissa ylläpidetään tietoa, joka ei välttämättä ole suoraan

komponenttiin liittyvää, ja ne ovat usein käytettävissä ja yhteisiä kaikissa komponenteissa eli niin kutsuttuja ainokaisia (*singleton pattern*). (Angular 2022a.) Palvelun voi halutessaan määritellä myös olevan saatavilla vain yhdessä moduulissa tai komponentilla ja sen lapsikomponenteilla, jolloin samasta palvelusta voi luoda useita instansseja käytettäväksi eri komponenteissa tarpeen mukaan.

Angular tarjoaa myös sisäänrakennetun reitittimen, jonka avulla voidaan määritellä, mitä näkymiä tietyssä verkkosivun reitissä näytetään. Angularin reititin keskeyttää HTML-pyynnöt ja tulkitsee niiden reitin, minkä jälkeen Angular rakentaa halutun näkymän reitistä löydettyjen tietojen perusteella. (Angular 2022a.) Koska Angular-sovellukset ovat yksisivuisia, on reititys ohjelmallista, eikä käyttäjälle lähetetä eri HTML-tiedostoa sen perusteella, missä reitissä hän vierailee.

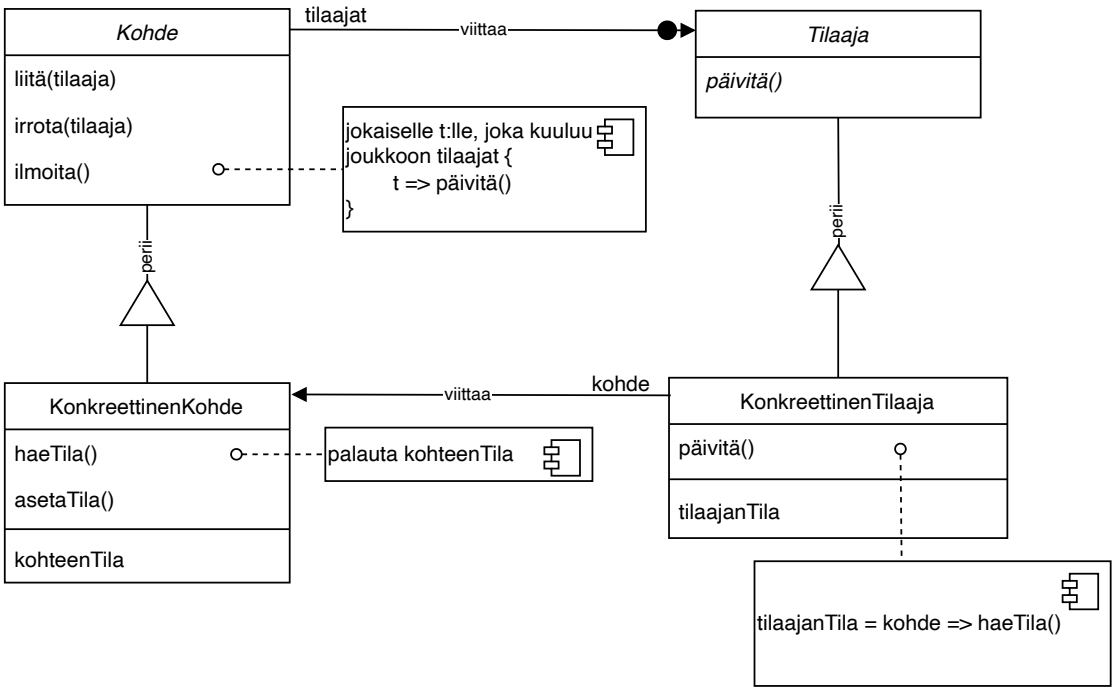
Mainittujen ominaisuuksien lisäksi Angular tarjoaa työkalut esimerkiksi HTTP-pyyntöjen käsittelyyn, mahdollisuuden kirjoittaa omia HTML-direktiivejä, valmiita HTML-tiedostoissa käytettäviä **putkia** (*Pipes*) sekä reaktiiviset työkalut lomakkeiden käsittelyyn. Angular on datankäsittelymalliltaan reaktiivinen ja käyttää suurelta osin hyödyksi RxJS-kirjastoa muokkaamaan dataa ja hakemaan/lähettämään sitä backend-palvelimille HTTP-pyyntöjen kautta. Varsinaisesti Angular sovelluksissa ei edellytetä RxJS:n ja reaktiivisen ohjelmointiparadigman käyttöä, mutta tällöin mielestäni taistellaan Angularin toimintaperiaatteita vastaan. Seuraavassa luvussa tutustutaan hieman paremmin RxJS-kirjastoon, jota voidaan käyttää Angular sovelluksien reaktiiviseen ohjelmointiin ja jonka käyttöä tässä työssä tullaan perustelemaan koodin luettavuuden ja ymmärrettävyyden näkökulmasta.

2.3 RxJS – Reactive Extensions for JavaScript

RxJS on lyhenne termistä Reactive Extensions (myös ReactiveX) for JavaScript, joka vapaasti suomennettuna tarkoittaa reaktiivisten lisäosien kirjastoa JavaScript-ohjelmointikielelle. Se on siis kokoelma koodia, jonka avulla reaktiivista ohjelmointiparadigmaa voidaan hyödyntää JavaScriptissä. Tässä lu-

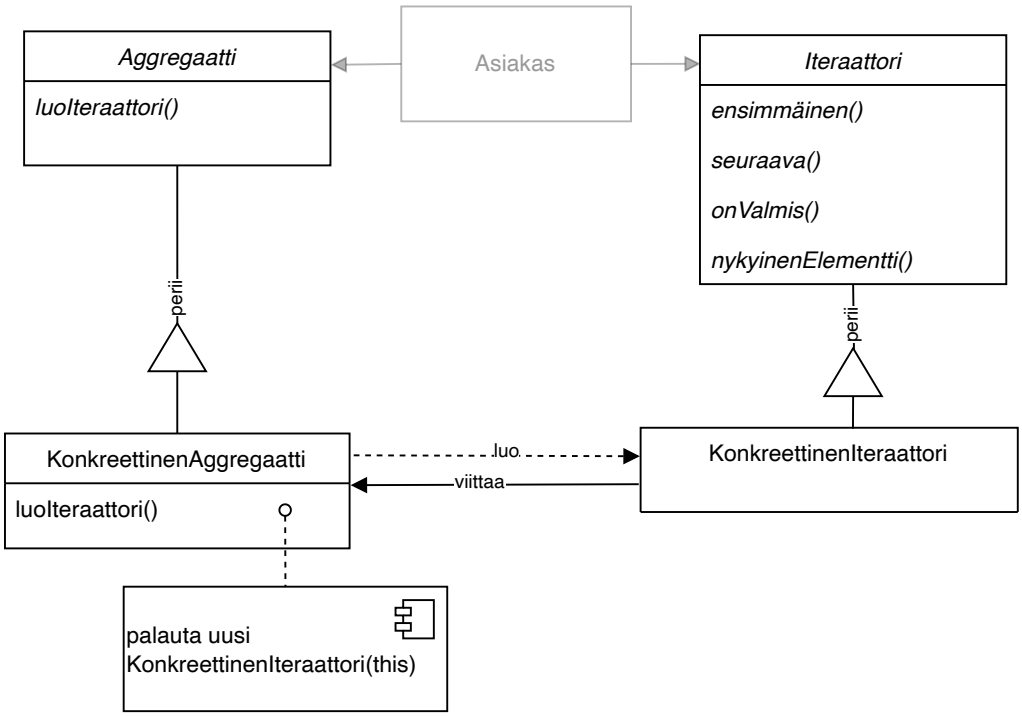
vussa tutustutaan tarkemmin RxJS:n tarjoamiin työkaluihin, jotka on suunniteltu helpottamaan reaktiivista ohjelmointia JavaScriptillä ja siten myös Angular-sovelluksissa.

ReactiveX yhdistää **tarkkailijamallin** (*Observer pattern*), **iteraatiomallin** (*Iterator pattern*) sekä **funktionaalisen ohjelmoinnin** (*Functional programming*) tarjotakseen ideaalin tavan hallita ajallisia tapahtumien sarjoja (RxJS 2023a). Kuvassa 3 esitettävässä tarkkailijamallissa määritellään yksi-moni riippuvuus kohteen ja tilaajien välille siten, että kohteen muuttuessa siitä riippuville tilaajille ilmoitetaan muutoksesta ja niitä päivitetään automaattisesti. Tässä mallissa kohde (*Subject* tai *Provider* – huomaa ero myöhemmin esiteltävään RxJS:n Subject-tyyppiin) on tietoinen tilaajistaan (*Observer* tai *Subscriber*) ja pitää niistä listaa itsellään. Tähän listaan voidaan lisätä tilaajia *liitä()*-metodilla tai poistaa tilaajia *irrota()* -metodilla. Kohde ilmoittaa *ilmoita()*-metodinsa avulla kaikille tilaajilleen omista muutoksistaan. Tämä ilmoitus laukaisee tilaajan *päivitä()*-metodin, joka muuttaa tilaajan tilaa vastaamaan kohteen tilaa. Tilaaja voi olla tilannut useamman kohteen, joten tyypillisesti kohde lähettää tilaajille myös tiedon siitä, mistä lähteestä kyseinen tieto tulee, jotta tilaaja voi reagoida eri tavalla eri lähteistä tulevaan tietoon. (Gamma ym. 1995, 293–295.)



Kuva 3. Tarkkailijamalli (Observer Pattern) (mukaillen Gamma ym. 1995, 294)

Iteraatiomallia taas esitetään kuvassa 4. Siinä ajatuksena on tarjota asiakkaalle (*Client*) pääsy aggregaatin (*Aggregate*) eli monielementtisen olion elementteihin ilman, että sen rakennetta paljastetaan. Iteraattori (*Iterator*) on tässä mallissa olio, joka on vastuussa pääsystä aggregaattina toimivan listan elementteihin sekä tiedosta, mitkä elementit on jo käyty läpi ja missä ”kohtaa” listaa ollaan etenemässä. Iteraattorilla on jokin joukko metodeja, joita kutsumalla voidaan lukea tämän aggregaatin elementtejä. Metodien joukko vaihtelee toteutuksesta riippuen, mutta esimerkiksi metodeina voisi olla kuvassa 4 esitetyt: *ensimmäinen()*-, *seuraava()-, onValmis()*- ja *nykyinenElementti()* -metodit. (Gamma ym. 1995, 257–259.)



Kuva 4. Iteraatiomalli (Iterator Pattern) (mukaillen Gamma ym. 1995, 259)

Näitä suunnittelumalleja käytetään RxJS:ssä luomaan **Observableja** (*Observable*), jotka ovat yhdistelmiä molemmista malleista (tarkkailija- ja iteraatiomalleista). Observablet toimivat tarkkailijamallin kohteina, joita voidaan tilata ja ne lähettävät dataa tilaajilleen, kun kyseinen data muuttuu. Ne siis ilmoittavat automaattisesti itseensä tapahtuvista muutoksista tilaajilleen. Toisaalta Observablet toimivat myös iteraatiomallin mukaisesti, sillä ne ovat ajallisia aggregaatteja, joilla on iteraattori datan läpikäymiseksi. Ajallisiksi aggregaateiksi Observablet voidaan ajatella, sillä niiden data on ajassa eroteltu joukko elementtejä. Vertauskuvallisesti normaali lista olisi paikallisesti eroteltu elementti-

kokoelma ja Observable taas ajallisesti eroteltu elementtikokoelma. Observablejen iteraattorimetodit mahdollistavat pääsyn sen sisältämään dataan. Iteraattorin tavoin Observable lähettää arvoja järjestyksessä, jonka lisäksi se voi lähettää tiedon lähettämisen päättymisestä. Nämä kaksi ominaisuutta tekevät Observablesta iteraattorin, mutta toisin kuin perinteisesti iteraattorit, niin Observable työntää datan, eikä asiakkaan tai kuluttajan tarvitse noutaa sitä kutsumalla iteraattorimetodeita. (Mota 2016.)

Avattuamme RxJS kirjaston asettumista reaktiivisen ohjelmoinnin kentälle ja sen rakennetta suunnittelumallien näkökulmasta, voimme syventyä sen perustana toimiviin tärkeimpiin datatyyppeihin: **Observable**, **Observer**, **Operator**, **Subscription** ja **Subject** -tyyppeihin.

Luultavasti tärkein näistä tyypeistä on **Observable**, jota voidaan kuvailla useamman arvon laiskaksi dataa työntäväksi kokoelmaksi. Toisin sanoen tyyppiä *Observable<T>* oleva olio lähettää tyyppiä T olevia arvoja silloin kun se tilataan (*subscribe*). (RxJS 2023b.) Tarkemmin määriteltynä Observable on funktio, joka sitoo Observer tyypin olion datatuottajaan (*Producer*). Observable voi itse luoda kyseisen datatuottajan, mutta se ei ole välttämätöntä. Ero kuumien ja kylmien Observableiden välillä seuraa siitä, luoko Observable itse datatuottajan – kuuma Observable ei luo itse datatuottajaa vaan viittaa johonkin sen ulkopuolella luotuun dataan ja kylmä Observable taas luo itse sen käyttämän datatuottajan. (Lesh 2016a.) Yksinkertaistettuna Observable siis yhdistää datatuottajan (esimerkiksi nappulan painamistapahtuman tai HTTP-pyynnön vastauksena saadun datavirran) ja datan tilaajana toimivan Observer-olion (joka määrittelee, mitä tilatulle datalle tehdään).

Yllä mainittu Observablejen erottelu kuumiin ja kylmiin – joka vastaa yleensä multicast-unicast erottelua – on huomionarvoista ainakin kahdessa tapauksessa: Ensinnäkin tapauksissa, joissa on tärkeää, että kaikki tilaajat saavat saman datan on käytettävä kuumaa Observablea. Kylmä Observable loisi uuden datan kullekin tilaajalleen. Myös tapauksissa, joissa Observable tuottaa sivuvaikutuksia tai datan tuottaminen on raskas operaatio, on yleensä tarkoituksenmukaista tehdä Observablesta kuuma. Esimerkiksi HTTP-pyyntöjen vastaukset ovat yleensä Observableja, joiden halutaan olevan kuumia. Tällöin

vältytään lähettämästä turhia ja usein raskaita HTTP-pyyntöjä API-rajapintoihin. (Taylor 2019.)

Observer-olio taas on kolmesta takaisinkutsufunktiosta (*callback function*) koostuva kokoelma, joka määrittelee, mitä Observablen lähettämällä datalla tehdään, kun se tilataan. Nämä Observerin kolme takaisinkutsufunktiota ovat *next()*, *error()* sekä *complete()*, ja ne välitetään Observablelle yllämainitussa *subscribe()*-metodissa. Observablen tilaaminen tapahtuu kyseistä *subscribe()*-metodia kutsumalla. (RxJS 2023c.) Observerin voidaan ajatella toimivan tilaajamallissa tilaajana, vaikkakin tarkalleen RxJS:ssä tilaaja (*Subscriber*) implementoi Observer tyyppiä. Takaisinkutsufunktiot määrittelevät sen, miten Observablen iteraattorimetodeihin reagoidaan. Observer siis määrittelee, mitä tehdään kun Observablen *next()*-, *error()*-, ja *complete()*-metodit emittoivat eli lähettävät dataa.

Kuvassa 5 on esitetty, miten Observable ja Observer luodaan ja miten ne sidotaan toisiinsa. Tässä esimerkissä esitetään myös ero kuumien ja kylmien Observablejen välillä konkreettisesti. Kuvan koodissa molemmille Observableille on kaksi tilausta, mutta kylmän Observablen tilaukset saavat keskenään eri numerot ja kuuman Observablen tilaukset saavat keskenään saman numeron. Datatuottajana tässä toimii *Math.random()*-metodi, joka tuottaa sattumanvaraisen numeron nollan ja yhden väliltä.

```typescript
import { Observable, Observer, Subscriber, Subscription } from 'rxjs';

const observerOlio: Observer<number> = {
  next: (x: number) => console.log('Observer sai arvon: ' + x),
  error: (err: any) => console.error('Observer sai virheen: ' + err),
  complete: () =>
    console.log('Observer sai tiedon lähettämisen päättymisestä.'),
};

// Kylmä Observable
const kylmäObservable: Observable<number> = new Observable((tilaaja: Subscriber<number>) => {
  const satunnainenNumero1: number = Math.random(); // Datatuottaja Observablen sisällä
  tilaaja.next(satunnainenNumero1);
  tilaaja.complete();
});

const satunnainenNumero2: number = Math.random(); // Datatuottaja Observablen ulkopuolella

// Kuuma Observable
const kuumaObservable: Observable<number>  = new Observable((tilaaja: Subscriber<number>) => {
  tilaaja.next(satunnainenNumero2);
  tilaaja.complete();
});

// Saavat eri arvot
const tilaus1: Subscription = kylmäObservable.subscribe(observerOlio);
const tilaus2: Subscription = kylmäObservable.subscribe(observerOlio);

// Saavat saman arvon
const tilaus3: Subscription = kuumaObservable.subscribe(observerOlio);
const tilaus4: Subscription = kuumaObservable.subscribe(observerOlio);
```

Kuva 5. Observablen ja Observerin luominen ja niiden sitominen toisiinsa

RxJS:n **operaattorit** (*Operator*) ovat funktioita, joilla voidaan muokata tai luoda Observableja. Operaattoreita on kahta tyyppiä: putkioperaattoreita (*pipeable operators*) ja luontioperaattoreita (*creation operator*). Putkioperaattorit ottavat parametrinään Observablen ja palauttavat uuden – yleensä jollakin tavalla muokatun – Observablen. Ne toteuttavat siis reaktiivisen ohjelmoinnin funktionaalista ulottuvuutta RxJS:ssä. Luontioperaattorit taas ovat funktioita, jotka luovat uusia Observableja joko muista Observableista (esimerkiksi yhdistämällä niitä) tai muista datatyypeistä. Luontioperaattorien avulla voidaan siis tehdä esimerkiksi datatyypistä T (tai vaikkapa listasta tyyppiä *Array<T>*) tyyppiä *Observable<T>* oleva Observable. (RxJS 2023d.)

Subscription on olio, joka yleensä edustaa Observablen suoritusta. Sen tärkein metodi on *unsubscribe()*, jota kutsuttaessa peruutetaan se Observablen suoritus, jota Subscription-olio edustaa. Subscription voi edustaa useiden Observableiden suoritusta ja niitä voidaan lisätä ja poistaa *add()*- ja *remove()*-metodien avulla Subscriptionista. Tällä mallilla voidaan useampi Observablen suoritus peruuttaa kutsumalla yhden Subscriptionin *unsubscribe()*-metodia.

(RxJS 2023d.) Yleisesti ottaen Subscriptionit ovat melko pienessä roolissa Angular ohjelmissa, sillä Observablen suoritusta (eli sen *subscribe()*-metodin palautusta) ei tarvitse tallentaa Subscriptioniin ja niin harvemmin tehdäänkään Angular sovelluksia ohjelmoitaessa. Kuvassa 6. on esitetty Subscription-olion luominen ja siihen tilauksien lisääminen *add()*-metodilla. Lopuksi nämä molemmat tilaukset voidaan peruuttaa kutsumalla Subscription-olion *unsubscribe()*-metodia, kuten kuvan rivillä 21.

```
1  import { Observable, Subscription, interval } from 'rxjs';
2
3  // RxJS:n interval() luontioperaattori luo Observablen,
4  // joka lähettää kasvavan kokonaisluvun sille parametrinä
5  // annetun millisekunti-intervallin välein.
6  const observable1: Observable<number> = interval(1000);
7  const observable2: Observable<number> = interval(2000);
8
9  const subscription1: Subscription = observable1.subscribe(
10   (x) => console.log('ensimmäinen tilaus: ' + x) // Observer-olion next-metodi
11 );
12
13 const subscription2: Subscription = observable2.subscribe((x) =>
14   console.log('toinen tilaus: ' + x)
15 );
16
17 subscription1.add(subscription2);
18
19 // Myöhemmin:
20 // Subscription olion unsubscribe-metodilla voidaan peruuttaa tilaus.
21 subscription1.unsubscribe();
```

Kuva 6. Subscription-olion luominen ja sen *unsubscribe()*-metodin kutsuminen

RxJS:n terminologiassa **Subject** on oliona hieman erilainen, kuin Gamman ym. (1995, 294) tarkkailijamallissa esittelemä Subject (tai Provider) olio. RxJS:n Subject on tyyppi, joka on sekä kuuma Observable että Observer. Subject säilyttää listaa sen tilanneista Observer-olioista, joten siinä mielessä se toimii kuten tarkkailijamallin Subject tai Provider. Koska Subject kuitenkin toimii myös Observer-oliona, on yksi sen pääasiallisista käyttötarkoituksista tehdä kylmistä Observableista kuumia Observableja. (RxJS 2023e.)

Subjectilla on siis Observerin *next()*-metodi ja siihen pystytään tämän avulla lähettämään haluttua dataa sekä saman tien vastaanottamaan tämä data tilaamalla se *subscribe()*-metodissa. Tällaista mallia voidaan kuitenkin pitää antisuunnittelumallina (*antipattern*), vaikka se onkin melko yleisesti käytetty tekniikka. Datan lähettäminen Subjectin ”läpi” tällä tavalla on varsinkin RxJS:ää vähemmän käyttäneiden ohjelmoijien keskuudessa suosittua, mutta sitä tulisi kuitenkin pyrkiä välttämään. Kuvassa 7 on esitetty esimerkki, jossa nappulan painamisesta tehdään Subject tällä antisuunnittelumallilla. Lisäksi kuvassa on esitetty suositellumpi tapa tehdä nappulan painamisesta Observable käyttäen RxJS:n tarjoamaa *fromEvent()*-operaattoria. Tällä operaattorilla voidaan tapahtumista luoda helposti Observableja eikä jouduta turvautumaan edellä mainittuun antisuunnittelumalliin. (Lesh 2016b.)

```
1  import { fromEvent, Observable, Subject } from 'rxjs';
2
3  // Haetaan nappula DOM:ista sen ID:llä
4  const nappula = document.getElementById('nappulanId');
5
6  // Tämä on esimerkki antisuunnittelumallista
7  const nappulanPainallukset1: Subject<Event> = new Subject<Event>();
8  nappula.addEventListener('click', (tapahtuma: Event) ⇒
9    nappulanPainallukset1.next(tapahtuma)
10 );
11
12 nappulanPainallukset1.subscribe((x) ⇒ console.log(x));
13
14 // Tämä on esimerkki RxJS:n fromEvent() -operaattorin käytöstä
15 const nappulanPainallukset2: Observable<Event> = fromEvent(nappula, 'click');
16
17 nappulanPainallukset2.subscribe((x) ⇒ console.log(x));
```

Kuva 7. Subjectin käyttäminen ja sen korvaaminen RxJS:n *fromEvent()*-operaattorilla

Nyt kun tarvittavat käsitteet on esitelty, voidaan tarkastella sitä, millaisia keinoja Angular tarjoaa reaktiivisen ohjelmointitavan tueksi ja miten reaktiivisesti kirjoitettu koodi eroaa imperatiivisesti kirjoitetusta koodista Angular-sovelluksessa.

3 ANGULAR SOVELLUKSEN REFAKTOROINTI REAKTIIVISEKSI

3.1 RxJS:n käyttäminen Angularissa

Angular käyttää Observableja ja RxJS:ää muutamissa olennaisissa toiminnoissa, joiden ansiosta reaktiivinen ohjelmointi on Angularissa luontevaa. Näitä toimintoja ovat:

1. **HttpClient**-luokka, joka palauttaa HTTP-kutsujen vastaukset Observableina
2. **EventEmitter**-luokka, joka on Subject luokan laajennos ja jolla dataa voidaan välittää lapsi ja vanhempi -komponenttien välillä.
3. **AsyncPipe**-templaattimerkintä, jolla voidaan näyttää Observablen sisältämä data HTML-templaatissa.
4. **Reaktiiviset lomakkeet**, joiden avulla lomakkeiden hallinta ja tilamuutokset voidaan tehdä reaktiivisesti.
5. **Router**, eli Angularin reititin, jonka avulla osoitepolkua, navigointitapahtumia ja reittiparametrejä voidaan käsitellä Observableina. (Angular 2022b.)

Edellä mainittuja toimintoja käyttämällä periaatteessa kaikkea Angular-sovelluksen dataa voi käsitellä Observableina. Reaktiivisen ohjelmointiparadigman mantran mukaan kaikki data on virtoja, ja Angular tarjoaakin valmiit työkalut tämän mantran mukaiseen ohjelmointiin. Vaikka Angular ei pakotakaan näitä työkaluja tai RxJS:ää käyttämään, niin mielestäni voidaan perustellusti esittää, että reaktiivinen ohjelmointi on olennainen osa Angular-sovelluksien ohjelmointia ja että sen mukanaan tuomat hyödyt sekä Angularin tarjoamat työkalut tekevät siitä ensisijaisen valinnan ohjelmointiparadigmaksi Angular-sovelluksille.

Reaktiivisen mallin mukaan datan tulisi virrata Observableissa ja sitä voidaan muokata operaattoreilla niin pitkään, kunnes tätä tietoa käytetään. Jossakin vaiheessa Observablen sisältämä tieto siis täytyy ikään kuin saada ”poimittua” tästä datavirrasta käyttöön. Observablen sisältämällä datalla tulisi ihannetilanteessa olla vain kaksi mahdollista päätepistettä: HTML-templaatti, jossa tieto näytetään visuaalisesti käyttäjälle tai HTTP-kutsun parametrinä, kun dataa lähetetään ohjelmasta ulospäin. (Lüdemann 2019.) Yllä esitellyillä työkaluilla esimerkiksi HTML-templaatissa datan näyttäminen voidaan toteuttaa Angularin AsyncPipella, joka tilaa Observablen ja näyttää sen datan HTML-templaatissa. HTTP-kutsujen suorittaminen taas onnistuu HttpClientilla, ja RxJS:n operaattoreiden avulla Observablejen sisältämää dataa voidaan käyttää

HTTP-kutsujen parametreinä. Näin sovelluksen sisällä data säilyy aina Observablejen virtana, vaikka näistä virroista data poimitaankin mainituissa päätepisteissä.

Koska Angular ohjelmointi ei edellytä reaktiivista ohjelmointimallia, onkin yllä mainittujen reaktiivisten työkalujen sekä RxJS:n väärinkäyttö ilmeisen yleinen ongelma Angular sovelluksissa. Muun muassa Lüdemann (2019) esittää, että kaksi yleisintä RxJS:n väärinkäytöstä Angularissa ovat Observablen muuttaminen tilaominaisuudeksi joko käyttämällä *subscribe()*-metodia tai *tap*-operaattoria ja Observablen muuttaminen Promiseksi *toPromise()*-metodilla. Myös Vardanyan (2020b) kirjoittaa, että RxJS:n *subscribe()*-metodia ja *tap*-operaattoria tulisi välttää kaikissa olosuhteissa. Hän jopa ehdottaa, että *subscribe()*-metodia ei tulisi käyttää koskaan ja *tap*-operaattoria olisi vältettävä kaikin mahdollisin keinoin.

Näiden väärinkäytösten lisäksi esimerkiksi Vardanyan (2020a) mainitsee kaksi mielestäni niin ikään yleistä väärinkäytöstä: lomakkeiden ja tapahtumankäsittelijöiden väärinkäytökset. Lomakkeita väärinkäytettäessä esimerkiksi kuunnellaan jonkin lomakkeen kentän *onchange()*-metodia ja reagoidaan kentän muutoksiin imperatiivisesti. Reaktiiviset lomakkeet tarjoavat muun muassa *valueChanges()*- ja *statusChanges()*-metodit, jotka lähettävät Observablen lomakkeen arvoista ja validiudesta aina kun jokin arvo tai lomakkeen validius muuttuu. Tapahtumankäsittelijöillä tarkoitetaan funktioita, jotka sidotaan käyttäjän tuottamiin tapahtumiin. Näitä tapahtumia voidaan kuunnella tapahtumankäsittelijöiden avulla ja niihin voidaan reagoida imperatiivisesti – kuten perinteisesti on tapana ei-reaktiivisissa sovelluksissa. Reaktiivisesti näistä tapahtumista voidaan kuitenkin tehdä Observableja esimerkiksi aiemmin mainitun RxJS:n *fromEvent()*-operaattorin avulla, kuten esitettiin kuvassa 7.

3.2 Refaktoroitava toiminnallisuus ja sen imperatiivinen toteutus

Refaktorointityön kohteeksi valikoitui Angular-sovelluksen yksi palvelu ja ne komponentit, jotka käyttävät kyseistä palvelua. Palvelun nimi on SpecialReasonService ja sen tehtävänä on tarkastaa ja tallentaa käyttäjäkohtaisia syitä asiakkaiden tietojen katselemiseen. Näitä syitä voidaan tallentaa istuntokohtaiseen tallennustilaan (*session storage*) tai niitä voi olla tallennettu pysyvästi

tietokantaan. Tietokantaan voidaan tehdä HTTP-kutsuja asiakkaan ja käyttäjän yksilöivillä tunnuksilla ja vastauksena saadaan tieto, onko käyttäjällä oikeus katsella kyseisen asiakkaan tietoja. Palvelussa tästä oikeudesta tietojen katseluun käytetään termiä **konteksti** (*context*). Mikäli sisään kirjautuneella käyttäjällä ja hänen valitsemallaan asiakkaalla ei ole kyseistä kontekstia tallennettuna istuntokohtaisesti eikä tietokantaan, niin käyttäjän tulee valita se ja valittu konteksti tallennetaan istuntokohtaiseen tallennustilaan. Lisäksi kontekstin puuttuessa on näytettävä alasvetovalikko syyn valitsemiseksi sekä estettävä seuraavalle näytölle siirtyminen, kunnes syy on valittu. Tarkastellaan ensin palvelun ja komponentin toimintaa ennen refaktorointia. Tämän jälkeen esitellään koodiin tehdyt muutokset ja arvioidaan refaktoroinnin onnistumista sekä reaktiivisen ohjelmointiparadigman omaksumisella saavutettuja hyötyjä.

Kuvassa 8 on esitetty yksinkertaistettu koodi SelectClient-komponentin HTML-templaatista, joka edustaa asiakkaan valintaa. Tässä komponentissa näytetään asiakashaku (*client-search*-lapsikomponentti), jossa asiakkaita voidaan hakea. Valittu asiakas välitetään *client-search* -komponentin *selected* -@Outputina SelectClient-komponentin *setValue()*-metodille (rivillä 4). Lisäksi tässä templaatissa näytetään *special-reason-dropdown* -lapsikomponentti, johon välitetään valittu asiakas *selectedClient*-@Inputina (rivillä 8). Asiakasvalintaa ylläpidetään SelectClient-komponentin *value$*-muuttujassa, jota muutetaan *setValue()*-metodilla aina kun valinta muuttuu *client-search* -komponentissa. Lisäksi käyttäjän etenemistä hallitaan Seuraava-nappulalla, joka on pois käytöstä (*disabled*), kun *disableButton$*-muuttujan arvo on tosi (*disableButton$* on Observable, jonka arvo saadaan Angularin AsyncPipen avulla). Nappulaa painettaessa suoritetaan *onConfirm()*-metodi, joka tarvittaessa tallettaa kontekstin ja navigoi seuraavalle näytölle.

```
1  <client-search #search
2    [configType]="configType"
3    [filterByIds]="existingIds"
4    (selected)="setValue($event)">
5  </client-search>
6
7  <special-reason-dropdown
8    [selectedClient]="value$ | async">
9  </special-reason-dropdown>
10
11 <button
12   [disabled]="disableButton$ | async"
13   (click)="onConfirm()">Seuraava</button>
```

Kuva 8. Asiakasvalinta-komponentin HTML-templaatti

Nyt kuvassa 9 esitetään, miten valittua asiakasta säilytetään Asiakasvalinta-komponentin *value$*-BehaviorSubjectissa. BehaviorSubject on yksinkertaistettuna Subject, johon voidaan tallentaa alkuarvo. Käytetään jatkossa BehaviorSubjectista ja ReplaySubjectista yhteistä nimitystä Subject, sillä niiden erot eivät ole työn kannalta merkittäviä. Kuvan 8 rivin 4 *setValue()*-metodi on nyt esitetty Kuvan 9 riveillä 25–26. Kun tämä metodi suoritetaan, niin *value$*-Subject emittoi valitun asiakkaan. Kuten aiemmin huomattiin, niin emittoitu arvo välitettiin *special-reason-dropdown*-komponenttiin *selectedClient*-@Inputina kuvan 8 rivillä 8. Lisäksi komponentissa on määritelty kuvan 8 rivin 15 *disableButton$*-Observable, jolla hallitaan käyttäjän pääsyä kyseiseltä sivulta eteenpäin. Tämän Observablen arvo on tosi ja nappula on pois käytöstä, mikäli asiakasta ei ole valittuna (*!coerceBooleanProperty(value)*) tai mikäli syyn valinta alasvetovalikko näytetään, mutta siihen ei ole valittu syytä (*canShowReason && disableSaveButton*) tai mikäli kontekstien haku tietokannasta on epäonnistunut (*isValidContextFailed*).

```
1  @Component()
2  export class SelectClientComponent {
3
4    public value$: BehaviorSubject<IClient | null> = new BehaviorSubject<IClient | null>(null);
5
6    public disableButton$: Observable<boolean> = combineLatest([
7      this.value$.asObservable(),
8      this._specialReasonService.canShowSpecialReason$,
9      this._specialReasonService.disableSaveButton$,
10     this._specialReasonService.isValidContextFailed$,
11   ]).pipe(
12     map(([value, canShowReason, disableSaveButton, isValidContextFailed]) =>
13       !coerceBooleanProperty(value)
14       || (canShowReason && disableSaveButton)
15       || isValidContextFailed
16     ),
17     debounceTime(0),
18     shareReplay(1),
19   );
20
21   constructor(
22     private _specialReasonService: SpecialReasonService,
23   ) {}
24
25   public setValue(value: IClient): void {
26     this.value$.next(value);
27   }
28
29   public onConfirm(): void {
30     this._specialReasonService.onSaveSpecialReason();
31   }
32 }
```

Kuva 9. Asiakasvalinta-komponentti (SelectClientComponent).

Seuraavassa kuvassa 10 on esitetty SpecialReasonDropdown-komponentin HTML-templaatti. Tässä huomionarvoisina seikkoina ovat seuraavat kohdat: Komponentti näytetään vain, mikäli rivin 1 *canShowReason*-muuttuja on tosi. Templaattiin välitetään Angularin reaktiivinen lomake *formGroup*-direktiivinä rivillä 2 – lomakkeen nimenä on *specialReasonForm*, joka määritellään komponentissa (kuvassa 11). Templaatissa sidotaan alasvetovalikon arvo *specialReasonForm*-lomakkeen kontrolliin nimeltä *specialReason* (rivillä 5) ja tekstisyöttökentän arvo kontrolliin nimeltä *description* (rivillä 18). Näiden lomakkeiden arvojen sitomisen lisäksi molempiin lomakekenttiin sidotaan niiden muutoksia seuraavat tapahtumankäsittelyfunktiot: *onReasonChange()* (rivillä 6) ja *isDecriptionEmpty()* (rivillä 19). Lisäksi alasvetovalikolle välitetään mahdolliset valinnat muuttujassa *specialReasons$* (rivillä 7). Lienee paikallaan muistuttaa, että lomakkeisiin tapahtumankäsittelyfunktioiden sitominen on yksi esimerkki Angularin reaktiivisten lomakkeiden väärinkäytöstä, kuten aiemmin muun muassa Vardanyan (2020a) esitti.

```
<div [hidden]="!canShowReason">
  <form [formGroup]="specialReasonForm">
      <mat-form-field >
          <mat-select name="specialReason"
            formControlName="specialReason"
            (selectionChange)="onReasonChange($event)">
            <ng-container *ngFor="let item of specialReasons$ | async">
              <mat-option [value]="item">{{item.display}}</mat-option>
            </ng-container>
          </mat-select>
          <mat-error *ngIf="specialReasonForm.controls['specialReason'].
errors?.required">
            Pakollinen tieto
          </mat-error>
      </mat-form-field>
    <div *ngIf="showDescription">
      <mat-form-field >
        <textarea matInput
          formControlName="description"
          (ngModelChange)="isDescriptionEmpty()">
        </textarea>
        <mat-error *ngIf="specialReasonForm.controls['description'].
errors?.required">
          Pakollinen tieto
        </mat-error>
      </mat-form-field>
    </div>
  </form>
</div>
```

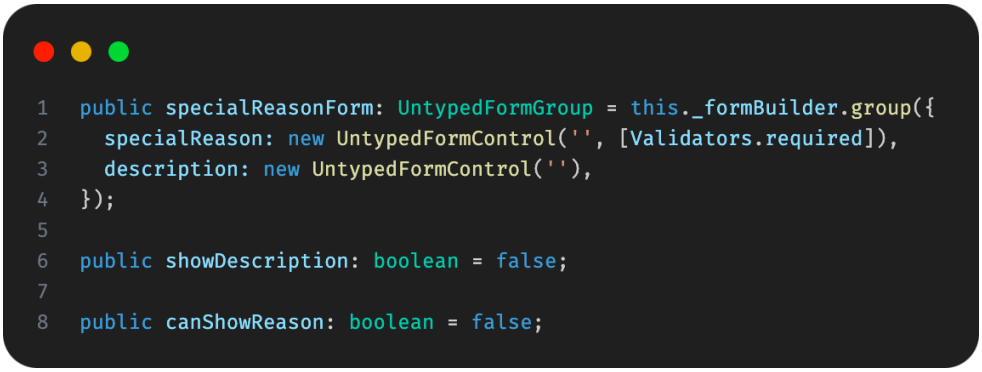
Kuva 10. SpecialReasonDropdown-komponentin HTML-templaatti

Tästä päästään tutkimaan kyseisen SpecialReasonDropdown-komponentin sisältöä. Tässä komponentissa törmätään useampaan aiemmin esillä olleeseen RxJS:n väärinkäytökseen ja se johtaa komponentin vaikeaselkoisuuteen sekä Angularin työkalujen epätavalliseen käyttöön. Aloitetaan komponentin tarkastelu sen muuttujien määrittelyistä kuvissa 11 ja 12. Kuvassa 11 määritellään muuttujat *_destroy*, *_specialReasonCodeSet$* ja *specialReasons$* sekä @Input muuttujat *selectedClient*, *selectedClients* ja *isMultipleClients*.

```
@Component()
export class SpecialReasonDropdownComponent implements OnInit, OnChanges, OnDestroy {
  private _destroy: Subject<void> = new Subject();

  private _specialReasonCodeSet$: Observable<ISpecialReason[]> = this._codeDAO.fetchCodeSet(SPECIAL_REASON).pipe(shareReplay(1),);

  public specialReasons$: BehaviorSubject<ISpecialReason[]> = new BehaviorSubject<ISpecialReason[]>([]);

  @Input()
  public selectedClient: IClient = null;

  @Input()
  public selectedClients: IClient[] | IMetaClient[] = null;

  @Input()
  public isMultipleClients: boolean = false;
```

Kuva 11. SpecialReasonDropdown-komponentin muuttujien määrittelyä (osa 1)

Kuvassa 12 määritellään loput muuttujat eli Angularin FormGroup muuttuja *specialReasonForm* sekä muuttujat *showDescription* ja *canShowReason*.



```
public specialReasonForm: UntypedFormGroup = this._formBuilder.group({
  specialReason: new UntypedFormControl('', [Validators.required]),
  description: new UntypedFormControl(''),
});

public showDescription: boolean = false;

public canShowReason: boolean = false;
```

Kuva 12. SpecialReasonDropdown-komponentin muuttujien määrittelyä (osa 2)

Seuraavaksi tutkitaan tapahtumankäsittelyfunktioiden määrittelyjä kuvassa 13. *onReasonChange()*-funktio suoritetaan, kun valittu syy alasvetovalikossa muuttuu. Tällöin suoritetaan useampi eri toimenpide. Ensiksi asetetaan specialReasonFormin *specialReason*-kontrollin arvoksi tämä valittu arvo. Tämän jälkeen asetetaan *showDescription*-muuttujan arvo sen mukaan, oliko valittu arvo yksi arvoista, jotka vaativat lisätietoja. Mikäli *showDescription* on tosi, niin asetetaan *description*-kenttä lomakkeessa pakolliseksi ja mikäli epätosi tyhjennetään *description*-kentän arvo ja poistetaan sen pakollisuus. Lopuksi välitetään SpecialReasonServicelle tieto lomakkeen validiudesta kuvan 13 rivillä 14. Kutsuttava palvelun *setDisableSaveButton()*-metodi emittoi palvelussa olevan *_disableSaveButton$*-Subjectin välitetyllä arvolla, ja palvelussa oleva *disableSaveButton$*-Observable on vain tämä Subject muutettuna Observableksi *asObservable()*-metodilla. Kuvan 13 rivillä 17 määritelty *isDescriptionEmpty()*-metodi taasen pelkästään välittää lomakkeen validiuden palvelulle.

```
public onReasonChange(event: MatSelectChange): void {
  const specialReason = event.value as ISpecialReason;
  this.specialReasonForm.controls[SPECIAL_REASON].setValue(specialReason);
  this.showDescription = CODES_TO_ENABLE_DESCRIPTION.includes(specialReason.code);
  const description = this.specialReasonForm.controls[DESCRIPTION];
  if (this.showDescription) {
    description.setValidators([Validators.required, Validators.maxLength(
DESCRIPTION_MAX_LENGTH)]);
    description.updateValueAndValidity();
  } else {
    description.clearValidators();
    description.reset();
    description.updateValueAndValidity();
  }
  this._specialReasonService.setDisableSaveButton(this.specialReasonForm.invalid);
}

public isDescriptionEmpty(): void {
  this._specialReasonService.setDisableSaveButton(this.specialReasonForm.invalid);
}
```

Kuva 13. SpecialReasonComponentin tapahtumankäsittelijät

Tarkastellaan vielä, mitä tapahtuu SpecialReason-komponentin lifecycle metodeissa *ngOnInit()* ja *ngOnChanges()*. Näiden metodien sisältö on kuvattu kuvissa 14 ja 15. Angularissa komponentin *ngOnInit()*-metodi suoritetaan, kun komponentti renderöidään, ja kuvasta 14 nähdään, että tällöin asetetaan jälleen palvelussa oleva *disableSaveButton$*-Observable todeksi. Tämän lisäksi tilataan aiemmin määritelty _specialReasonCodeSet$-Observable, joka sisältää HTTP-kutsun vastauksena saadut mahdolliset syyt. Nämä syyt asetetaan *specialReasons$*-Subjectin arvoksi. Tässä huomataan käytettävän Observablen *subscribe()*-metodia, jota esimerkiksi Vardanyan (2020b) ja Lüdemann (2019) varoittivat käyttämästä. Komponenttia refaktoroitaessa tullaan huomaamaan tämä Observablen tilaaminen ja Subjectin käyttäminen muutenkin tarpeettomaksi.

```
ngOnInit(): void {
  this._specialReasonService.setDisableSaveButton(true);

  this._specialReasonCodeSet$.pipe(
    takeUntil(this._destroy),
  ).subscribe((specialreasons) => this.specialReasons$.next(specialreasons));
```

Kuva 14. SpecialReasonComponentin *ngOnInit()*-metodi

Lopulta tarkastellaan vielä kuvassa 15 nähtävää SpecialReasonDropdown-komponentin *ngOnChanges()*-metodia. Muistetaan, että SpecialReasonDropdown-komponentti on ylipäätään näkyvissä silloin, kun muuttuja *canShowReason* on tosi. Kyseinen muuttuja määritellään ensin epätodeksi (kuva 12 rivi 8) ja sen jälkeen sille asetetaan uusi arvo Angularin *ngOnChanges()*-metodissa, joka esitetään kuvassa 15. Tässä metodissa muuttujan *canShowReason* arvo asetetaan SpecialReasonService-palvelusta saadun funktiokutsun vastauksen mukaan. Tässä kohtaa huomionarvoista lienee epätavallinen *ngOnChages()*-metodin määrittely asynkroniseksi, jollaiseen en ole aiemmin törmännyt. Angularin *ngOnChages()*-metodi suoritetaan periaatteessa aina kun komponentin @Input-arvot muuttuvat ja tässä toteutuksessa on tarkoituksena tehdä uusi asynkroninen funktiokutsu SpecialReasonService-palveluun aina kun komponentin *selectedClient*- tai *selectedClients*-@Input-tuit muuttuvat. Kuten myöhemmin tullaan näkemään, niin SpecialReasonService-palvelun vastaukset ovat Promiseja, ja ilmeisesti tästä syystä on jouduttu käyttämään Angularissa epätavallista async-await mallia ja asynkronista *ngOnChanges()*-metodia.

```
1  async ngOnChanges(): Promise<void> {
2    this.canShowReason = this.isMultipleClients
3      ? this._specialReasonService.canShowSpecialReasonForMultipleClients(this.selectedClients)
4      : await this._specialReasonService.canShowSpecialReason(this.selectedClient);
5    this._changeDetectorRef.detectChanges();
6  }
```

Kuva 15. SpecialReasonDropdown-komponentin *ngOnChanges()*-metodi

Siirrytään seuraavaksi tarkastelemaan SpecialReasonService-palvelun toiminnallisuutta koodin kautta. Kuvan 15 koodista huomataan, että mikäli kyseessä on useamman asiakkaan tietojen haku, niin kutsutaan SpecialReasonServicen *canShowSpecialReasonForMultipleClients()*-metodia näillä asiakkailla. Tämä metodi palauttaa boolean-arvon tosi, jos sille välitetään asiakkaita ollenkaan, kuten huomataan kuvan 16 koodista. Samalla muutetaan palvelun *_canShowSpecialReason*-Subjectin arvo todeksi.

```
public canShowSpecialReasonForMultipleClients(selectedClients: IClient[] | IMetaClient[]): boolean {
    const isEnabled = selectedClients.length > 0;
    this._canShowSpecialReason.next(isEnabled);
    return isEnabled;
  }
```

Kuva 16. SpecialReasonServicen *canShowSpecialReasonForMultipleClients()*-metodi

Kuvasta 17 nähdään vielä, miten tämä Subject (ja kolme muuta tärkeää muuttujaa) muutetaan Observableiksi, jotta niiden muutoksiin voidaan reagoida muualla sovelluksessa. Huomataan tässä käytettävän RxJS:n työkaluja kuten Subjecteja ja Observableja, mutta ohjelmointityyli on kuitenkin imperatiivinen. Subjectille asetetaan tietty arvo imperatiivisesti eikä se automaattisesti reagoi muiden arvojen muutoksiin, kuten reaktiivisessa ohjelmoinnissa tulisi tapahtua. Kyseisessä palvelussa ja komponenteissa tämä vaikuttaa olleen yleisesti käytetty ohjelmointimalli, mutta kuten muistamme Leshin (2016b) esittäneen, ei se ole tapa, jolla RxJS:ää on suunniteltu käytettäväksi.

```
private _saveSpecialReason: BehaviorSubject<boolean> = new BehaviorSubject(false);

private _disableSaveButton: BehaviorSubject<boolean> = new BehaviorSubject(true);

private _canShowSpecialReason: BehaviorSubject<boolean> = new BehaviorSubject(false);

private _isValidContextFailed: BehaviorSubject<boolean> = new BehaviorSubject(this.
_canAccessSpecialReason);

public saveSpecialReason$: Observable<boolean> = this._saveSpecialReason.asObservable();

public disableSaveButton$: Observable<boolean> = this._disableSaveButton.asObservable();

public isValidContextFailed$: Observable<boolean> = this._isValidContextFailed.asObservable();

public canShowSpecialReason$: Observable<boolean> = this._canShowSpecialReason.asObservable();
```

Kuva 17. SpecialReasonServicen antisuunittelumalli, jossa Subjectien arvot asetetaan imperatiivisesti ja käännetään sitten Observableiksi

Tarkemmin koodiin tutustuttuani löysin useamman asiakkaan tapausta käytettävän vain, jos jo aiemmin oli haettu useamman asiakkaan kontekstit metodilla *clientsWithOutContext()*, joka esitetään kuvassa 18. Tämä metodi käyttää toista metodia *getClientsWithoutContext()* palauttaakseen parametrinään saamansa asiakaslistan ne asiakkaat, joilla ei ole kontekstia tallennettuna. Mikäli jollakin valituista asiakkaista ei ole kontekstia (eli *getClientsWithoutContext()*-metodin palauttama array ei ole tyhjä), niin avataan Dialog-elementti, jossa

näytetään valitut asiakkaat ja kyseinen SpecialReasonDropdown-komponentti. Palaamme tähän *getClientsWithoutContext()*-metodiin myöhemmin, sillä sitä käytetään myös seuraavaksi tarkasteltavassa yksittäisen asiakkaan kontekstin tarkastuksessa.

```
public async clientsWithoutContext(
  clients: IMetaClient[] | IClient[],
  value: number,
  isCaseType?: boolean,
): Promise<(IMetaClient | IClient)[]> {
  const validClients = isCaseType ? this._getValidClients(clients as IPerson[]) : clients;
  const clientsWithoutContext = this._canAccessSpecialReason && validClients?.length > value
    ? await this.getClientsWithoutContext(validClients)
    : [];
  return clientsWithoutContext;
}
```

Kuva 18. SpecialReasonServicen *clientsWithoutContext()*-metodi

Kuvan 15 koodia tarkasteltaessa huomataan yksittäisen asiakkaan tilanteessa kutsuttavan SpecialReasonServicen *canShowSpecialReason()*-metodia valittu asiakas parametrinä. Kyseinen metodi on esitetty kuvassa 19 ja sen huomataan kutsuvan toista palvelussa olevaa metodia *_enableSpecialReason()* sama asiakas parametrinään ja palauttavan tämän metodin vastauksen sekä asettavan vastauksen *_canShowSpecialReason*-Subjectin uudeksi arvoksi (samalla tavalla imperatiivisesti kuin tehtiin useamman asiakkaan tapauksessa). Tässä kohtaa voidaan huomata kaava, jossa Subjectien arvoja asetetaan imperatiivisesti metodien sivuvaikutuksina, mikä on omiaan tekemään koodin muokkaamisesta haastavaa ja synnyttämään virheitä, joita on hankalaa paikantaa ja korjata.

```
public async canShowSpecialReason(selectedClient: IClient): Promise<boolean> {
  const enabledStatus = await this._enableSpecialReason(selectedClient);
  this._canShowSpecialReason.next(enabledStatus);
  return enabledStatus;
}
```

Kuva 19. SpecialReasonServicen *canShowSpecialReason()*-metodi

Kuvassa 20 on esillä *_enableSpecialReason()*-metodin koodi, jossa asetetaan ensin jälleen imperatiivisesti *_isValidContextFailed()*-Subjectin arvoksi tosi –

tarkemmin *_canAccessSpecialReason*-muuttuja on kytkin (*feature toggle*), mutta tämän työn piirissä kyseinen kytkin on aina päällä eli tosi. Tämän jälkeen kutsutaan jälleen aiemmin mainittua *getClientsWithoutContext()*-metodia sama asiakas parametrinä. Tällä kertaa asiakas muutetaan kuitenkin listaksi asiakkaita, sillä samalla metodilla voidaan hakea useamman asiakkaan tiedot. Metodi palauttaa toden, jos kontekstittomien asiakkaiden määrä ei ole nolla.

```
private async _enableSpecialReason(selectedClient: IClient): Promise<boolean> {
  this._isValidContextFailed.next(this._canAccessSpecialReason);
  const clients = selectedClient && this._canAccessSpecialReason
    ? await this.getClientsWithoutContext([selectedClient])
    : [];
  return clients.length !== 0;
}
```

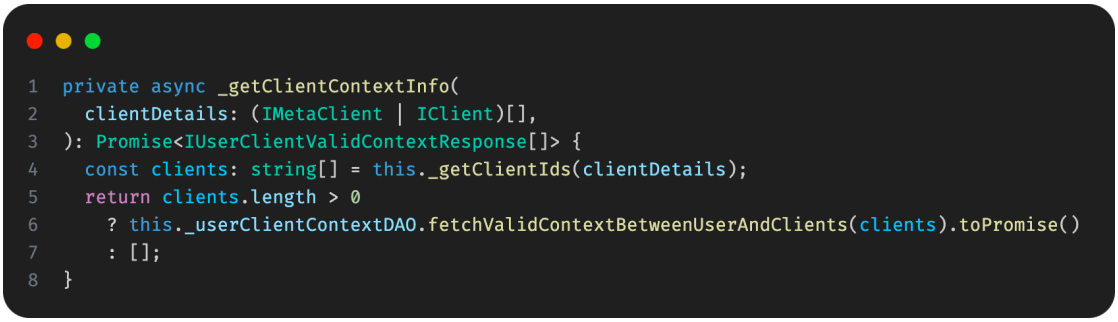
Kuva 20. SpecialReasonServicen *_enableSpecialReason()*-metodi

Viimein kuvassa 21 on esitetty tämän *getClientsWithoutContext()*-metodin koodi, joka tiivistettynä välittää listan asiakkaita jälleen uudelle metodille *_getClientContextInfo(),* joka on esitetty kuvassa 22. Tämä metodi palauttaa parametreinä saamistaan vain ne asiakkaat, joiden tietojen katseluun ei ole tallennettu kontekstia. Lisäksi asetetaan taas imperatiivisesti Subjectille *_isValidContextFailed* uusi arvo sen mukaan onnistuiko *_getClientContextInfo()*-metodin HTTP-kutsu vai ei.

```
public async getClientsWithoutContext(clientDetails: (IMetaClient | IClient)[]):
Promise<(IClient | IMetaClient)[]> {
  let clientsWithoutContext: (IClient | IMetaClient)[] = [];
  return await this._getClientContextInfo(clientDetails).then((_clientResults) => {
    const clientsWithNoContext = _clientResults.filter((client) => !client.context);
    clientsWithoutContext = clientDetails.filter((_client) =>
      clientsWithNoContext.some((val) =>
        val.clientId === _client['id'] || val.clientId === _client['rekisteritunnus']
      )

);  this._isValidContextFailed.next(false);
    return clientsWithoutContext;
  }).catch(() => {
    this._isValidContextFailed.next(true);
    return [];
  });
}
```

Kuva 21. SpecialReasonServicen *getClientsWithoutContext()*-metodi

Huomionarvoista kuvan 22 *_getClientContextInfo()*-metodissa on, että se kutsuu toisen palvelun *_userClientContextDAO* metodia *fetchValidContextBetweenUserAndClient()*, joka tekee HTTP-kutsun palvelimelle. Tämä vastaus on Angularille tyypilliseen tapaan Observable, jonka sisältönä on tieto välitetyn asiakaslistan asiakkaiden konteksteista. Tämä Observable muutetaan jälleen epäreaktiivisesti Promiseksi Observableilla olevalla *toPromise()*-metodilla kuvassa 22 rivillä 6. Esimerkiksi Lüdemann (2019) nimenomaisesti aiemmin varoitti käyttämästä kyseistä *toPromise()*-metodia, sillä Promiset eivät kuulu reaktiiviseen ohjelmointiin.



```
1  private async _getClientContextInfo(
2    clientDetails: (IMetaClient | IClient)[],
3  ): Promise<IUserClientValidContextResponse[]> {
4    const clients: string[] = this._getClientIds(clientDetails);
5    return clients.length > 0
6      ? this._userClientContextDAO.fetchValidContextBetweenUserAndClients(clients).toPromise()
7      : [];
8  }
```

Kuva 22. SpecialReasonServicen *_getClientContextInfo()*-metodi, jossa käytetään Observablen *toPromise()*-metodia

Tästä alkuperäisen koodin käsittelystä lienee käynyt selväksi, miten vaikeasti ymmärrettäväksi ohjelmointikoodi voi pahimmillaan mennä, kun yritetään taivuttaa Angularin reaktiivisia välineitä imperatiiviseen muotoon. Käsitellyssä toiminnallisuudessa oli käytetty Observablen *toPromise()*-metodia, joka johti async-await ketjuun, jonka ymmärtäminen oli ainakin itselleni haastavaa. Lisäksi toiminnallisuudessa viljeltiin antisuunnittelumallia, jossa Subjectien arvoja asetettiin niiden *next()*-metodeilla imperatiivisesti ja sen jälkeen näistä Subjecteista oli tehty Observableja *asObservable()*-metodilla. Tämä tehtiin metodien sivuvaikutuksina, jolloin rikottiin lisäksi funktionaalisen ohjelmoinnin sääntöjä. Observableja oli myös tilattu Angularin *ngOnInit()*-metodissa ja niiden *subscribe()*-metodissa oli asetettu saatu arvo uuden Subjectin arvoksi. Subjectia oli tässäkin käytetty ikään kuin tilamuuttujana, mikä on epätavallista eikä tarkoituksenmukaista. Olisi jopa ollut suotavampaa asettaa tämä Observablen arvo suoraan tilamuuttujan arvoksi, sillä tässä Subjectin käyttämisessä tilamuuttujana ei ole suoraan sanottuna mitään järkeä. Vielä huomattiin, että myös Angularin reaktiivisia lomakkeita oli väärinkäytetty. Lomakkeiden ele-

mentteihin oli sidottu tapahtumankäsittelyfunktioita, joiden avulla imperatiivisesti asetettiin itse lomakkeen (*FormGroup* tai *FormControl*) arvoja ja suoritettiin muitakin sivuvaikutuksia, kun lomakkeen arvot muuttuivat. Tavoitteena refaktoroinnilla on korjata nämä mainitut ongelmakohdat koodissa, jotta palvelusta ja komponenteista saataisiin ymmärrettävämpiä, yksinkertaisempia ja muokattavampia.

3.3 Reaktiiviseksi refaktoroitu palvelu ja komponentit

Aloitetaan refaktorointi tarkastelemalla niitä tekijöitä, jotka vaikuttavat syytä kysyvän SpecialReasonDropdown-komponentin näyttämiseen. Huomataan, että ainoa vaikuttava tekijä on valittu asiakas tai asiakkaat. Jos joltakin valitulta asiakkaalta puuttuu konteksti, niin alasvetovalikko näytetään. Toinen refaktoroinnin kannalta tärkeä seikka on se, milloin asiakashakukomponentin Seuraava-nappula on painettavissa. Tämä nappula disabloidaan silloin, kun jokin seuraavista on tosi: valittuja asiakkaita ei ole, alasvetovalikko näytetään eikä syytä ole valittu tai HTTP-kutsu kontekstien hakemiseksi ei onnistu.

Refaktoroidaan ensin SpecialReasonDropdownin HTML-templaatti kuvan 23 mukaiseksi. Käytetään komponentin näyttämiseksi Observablea nimeltä *showSpecialReasonDropdown$*, joka tilataan Angularin AsyncPipen avulla templaatissa. Lisäksi poistetaan lomakkeen tapahtumankäsittelyfunktiot, sillä niitä ei tarvita käytettäessä Angularin reaktiivisia lomakkeita. Lisäksi hallitaan lisätietokentän näyttämistä niin ikään Observablen *showDescription$* avulla, joka tilataan AsyncPipen avulla. Myös alasvetovalikon valinnat saadaan Observablesta *specialReasons$* jälleen AsyncPipen avulla.

```
1  <div *ngIf="showSpecialReasonDropdown$ | async">
2    <form [formGroup]="form">
3        <mat-form-field >
4            <mat-select name="specialReason"
5              formControlName="specialReason">
6              <ng-container *ngFor="let item of specialReasons$ | async">
7                <mat-option [value]="item">{{item.display}}</mat-option>
8              </ng-container>
9            </mat-select>
10           <mat-error *ngIf="form.controls['specialReason'].errors?.required">
11             Pakollinen tieto
12           </mat-error>
13       </mat-form-field>
14     <div *ngIf="showDescription$ | async">
15       <mat-form-field >
16         <textarea matInput
17           formControlName="description">
18         </textarea>
19         <mat-error *ngIf="form.controls['description'].errors?.required">
20           Pakollinen tieto
21         </mat-error>
22       </mat-form-field>
23     </div>
24   </form>
25 </div>
```

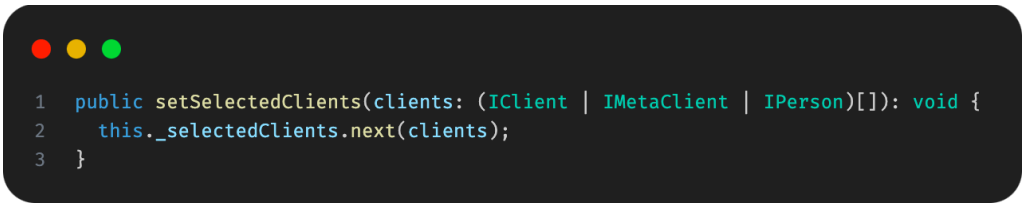
Kuva 23. Refaktoroitu SpecialReasonDropdown-komponentin HTML-templaatti

Nyt kuvassa 24 on esitetty refaktoroitu SpecialReasonDropdown-komponentin koodi. Kuvan 24 rivillä 4 huomataan, että alasvetovalikossa näytettävät vaihtoehdot saadaan nyt suoraan *_codeDAO*-palvelusta, eikä tätä Observablea tarvitse tilata komponentissa, vaan se tehdään templaatissa AsyncPipellä. Lisäksi *showDescription$*-Observable rivillä 13 määritellään Angularin lomakkeiden *valueChanges()*-metodin avulla suoraan lomakkeen arvojen mukaan. Itse lomake on määritelty SpecialReasonService-palvelussa ja tässä komponentissa vain viitataan siihen. Samalla tavalla rivin 19 *showSpecialReasonDropdown$*-Observable on pelkkä viittaus palvelussa määriteltyyn Observableen. Oleellista on myös huomata, että komponenttiin annetaan @Inputina tällä kertaa asiakaslista, sillä toiminnallisuuden kannalta ei ole oleellista onko asiakkaita useita vai yksi. Tämä lista voi tietenkin sisältää vain yhden asiakkaan, mutta useamman ja yhden asiakkaan tapauksia voidaan käsitellä täysin identtisesti. Nyt *selectedClients*-@Input kuvassa 24 riveillä 6–9 on itse asiassa setter-funktio, joka lähettää valitut asiakkaat SpecialReasonService-palveluun.

```
export class SpecialReasonDropdownComponent implements OnInit, OnDestroy {
  private _destroy: Subject<void> = new Subject();

  public specialReasons$: Observable<ISystemCode[]> =  this._codeDAO.fetchCodeSet(SPECIAL_REASON).pipe(shareReplay(1),);

  @Input('selectedClients')
  set __selectedClients(value: InputType<(IClient | IMetaClient)[]>) {
    if (value?.[0]) { this._service.setSelectedClients(value); }
  }

  public form: FormGroup<ISpecialReasonForm> = this._service.form;

  public showDescription$: Observable<boolean> = this.form.controls.specialReason.valueChanges.pipe(
    map((specialReason) => !!specialReason && CODES_TO_ENABLE_DESCRIPTION.includes(specialReason.code)),
    tap((showDescription) => this._setDescriptionValidators(showDescription)),
    shareReplay(1),
  );

  public showSpecialReasonDropdown$: Observable<boolean> = this._service.showSpecialReasonDropdown$;

  ngOnInit(): void {
    this._service.saveSpecialReasonInSession$.pipe(
      takeUntil(this._destroy),
    ).subscribe((success) => { if (success) { this._service.afterSaving(); } });
  }

  ngOnDestroy(): void {
    this._destroy.next();
    this._destroy.complete();
  }
```

Kuva 24. Refaktoroitu SpecialReasonDropdown-komponentin TypeScript-koodi

Kuvassa 25 näytetään, miten komponentilta lähetetyt asiakkaat otetaan vastaan SpecialReasonService-palvelussa *setSelectedClients()*-metodilla, joka asettaa ne Subjectin *_selectedClients* arvoksi.



```
public setSelectedClients(clients: (IClient | IMetaClient | IPerson)[]): void {
  this._selectedClients.next(clients);
}
```

Kuva 25. SpecialReasonServicen *setSelectedClients()*-metodi

Seuraavaksi kuvassa 26 esitetään, miten näistä asiakkaista saadaan johdettua RxJS:n työkalujen avulla tarvittava *showSpecialReasonDropdown$*-Observable reaktiivisesti. Tässä valitut asiakkaat sisältävää Subjectia käsitellään RxJS:n operaattorien avulla funktionaalisesti askeleittain. Ensin päästetään läpi ne asiakkaat, joiden syytä ei ole tallennettu istuntokohtaiseen tallennustilaan *(_clientsWithoutSpecialReason$)* ja sitten ne, joilla ei ole kontekstia tietokannassa *(_clientsWithoutContexts$)*. Lopulta rivillä 18 määritellään *showSpecialReasonDropdown$*-Observable todeksi, mikäli asiakkaita, joilla ei ole kontekstia on useampi kuin nolla. Valittuja asiakkaita on käsitelty koko ajan listana, joten nyt sama *showSpecialReasonDropdown$*-Observable kertoo kaikissa tapauksissa, milloin SpecialReasonDropdown-komponentti tulisi näyttää.

```
1  private _selectedClients: ReplaySubject<(IClient | IMetaClient)[]> = new ReplaySubject(1);
2
3  private _clientsWithoutSpecialReason$: Observable<(IClient | IMetaClient)[]> = this._selectedClients.pipe(
4    map((clients) => this._getClientsWithNoSpecialReason(clients)),
5    shareReplay(1),
6  );
7
8  private _clientContextsResponse$: Observable<IContextResponse> = this._clientsWithoutSpecialReason$.pipe(
9    switchMap((clients) => this._getClientContextResponse$(clients)),
10   shareReplay(1),
11 );
12
13 private _clientsWithoutContexts$: Observable<(IClient | IMetaClient)[]> = this._clientContextsResponse$.pipe(
14   map(({ clients, contexts }) => this._filterOnlyClientsWithInvalidContexts(clients, contexts)),
15   shareReplay(1),
16 );
17
18 public showSpecialReasonDropdown$: Observable<boolean> = this._clientsWithoutContexts$.pipe(
19   map((clients) => clients.length > 0),
20   shareReplay(1),
21 );
```

Kuva 26. SpecialReasonServicen reaktiivinen asiakkaiden vaihtumista kuunteleva Observable-virta

Kuvan 26 rivillä 8 määritelty *_clientContextsResponse$*-Observable on vastuussa HTTP-kutsusta palvelimelle tai tietokantaan. Kutsun vastauksena on Observable, joten sitä voidaan käsitellä RxJS:n operaattoreilla ja liittää siihen tieto HTTP-kutsun onnistumisesta. Tämä HTTP-kutsu tehdään kuvan 27 metodissa *_getClientContextsResponse()*, jonka vastaus asetetaan *_clientContextsResponse$*-Observablen arvoksi käyttäen RxJS:n *switchMap*-operaattoria. Huomataan, että tässä kohtaa ei ole mitään syytä muuttaa tätä saatua vastausta Promiseksi, kuten oli tehty aiemmassa toteutuksessa. Lisäksi voimme käyttää RxJS:n *catchError*-operaattoria hallitsemaan HTTP-kutsujen virheet. Nyt virhe voidaan tallentaa osaksi vastausta, josta se voidaan myöhemmin hakea ja määritellä *isValidContextFailed$*-Observablen arvoksi kuvassa 28 rivillä 12. Myöskään tässä kohtaa ei ole tarpeen turvautua imperatiiviseen ohjelmointiin ja arvojen tallentamiseen Subjecteihin niiden *next()*-metodeilla. Mainittu *isValidContextFailed$*-Observable on nyt määritelty reaktiivisesti ja se päivittyy automaattisesti aina kun asiakaslista päivittyy.

```
1  private _getClientContextResponse$(
2    clients: (IMetaClient | IClient)[],
3  ): Observable<IContextResponse> {
4    const clientIds: string[] = clients.map((client) => this._isClient(client) ? client.id : client.rekisteritunnus)
5      .filter((clientId) => !!clientId);
6
7    return clientIds.length > 0
8      ? this._userClientContextDAO.fetchValidContextBetweenUserAndClients(clientIds).pipe(
9        map((contexts) => ({
10         clients,
11         contexts,
12         failed: false,
13       })),
14       catchError(() => {
15         return of<IContextResponse>({
16           clients,
17           contexts: [],
18           failed: true,
19
20 });   }),
21       shareReplay(1),
22     )
23     : of<IContextResponse>({
24       clients,
25       contexts: [],
26       failed: false,
27
28 });
```

Kuva 27. SpecialReasonServicen *_getClientContextResponse()*-metodi

Esitetään vielä kuvassa 28, miten hallitaan asiakashaku-komponentissa olevan Seuraava-nappulan disabloimista. Kuvassa ensin määritellään Angularin reaktiivinen lomake, jossa ylläpidetään alasvetovalikon valintaa *specialReason*-kontrollissa sekä lisätietokentän tekstisisältöä *description*-kontrollissa. Nyt määritellään rivillä 6 *specialReasonFormIsInvalid$*-Observable siten, että se on tosi, kun lomake ei ole validi. Validiutta hallitaan Angularin reaktiivisten lomakkeiden Validators-ominaisuuksilla, joilla voidaan asettaa lomakkeen arvoja pakollisiksi tai antaa niille rajoituksia. Nyt voidaan käyttää RxJS:n *combineLatest*-luontioperaattoria ja määritellä nappulaa hallinnoiva *canNotContinue$*-Observable kuvan 28 rivillä 17. Tämä Observable tarkastaa samat asiat, mitä aiempi kuvan 9 *disableButton$*-Observable, mutta tällä kertaa tämä Observable voidaan määritellä SpecialReasonService-palvelussa, jolloin sitä voidaan käyttää useammassa komponentissa, joissa asiakashakuja suoritetaan (näitä komponentteja oli itse asiassa useampia, mutta tämän työn raportoinnin piirissä ei ollut mielekästä esitellä niitä kaikkia, vaan käytettiin yhtä asiakashaku-komponenttia yleisenä esimerkkinä).

```
1  public form: FormGroup<ISpecialReasonForm> = new FormGroup<ISpecialReasonForm>({
2    specialReason: new FormControl<ISystemCode | null>(null, [Validators.required]),
3    description: new FormControl<string | null>(null),
4  });
5
6  public specialReasonFormIsInvalid$: Observable<boolean> = this.form.statusChanges.pipe(
7    startWith(this.form.status),
8    map((status) => status === 'INVALID'),
9    shareReplay(1),
10 );
11
12 public isValidContextFailed$: Observable<boolean> = this._clientContextsResponse$.pipe(
13   map(({ failed }) => failed),
14   shareReplay(1),
15 );
16
17 public canNotContinue$: Observable<boolean> = combineLatest([
18   this._selectedClients,
19   this.showSpecialReasonDropdown$,
20   this.specialReasonFormIsInvalid$,
21   this.isValidContextFailed$,
22 ]).pipe(
23   map(([selectedClients, canShowReason, specialReasonFormIsInvalid, isValidContextFailed]) =>
24     !selectedClients[0] || (canShowReason && specialReasonFormIsInvalid) || isValidContextFailed
25   ),
26   debounceTime(0),
27   shareReplay(1),
28 );
```

Kuva 28. SpecialReasonServicen form-lomakkeen määrittely sekä Seuraava-nappulan hallinnoimiseen käytettävä *canNotContinue$*-Observable

Tämä *canNotContinue$*-Observable voidaan jälleen tilata Angularin AsyncPipellä asiakashakukomponentin HTML-templaatissa kuten kuvassa 29 näytetään. Huomataan, että asiakashakukomponentissa välitetään SpecialReasonDropdown-komponentille tällä kertaa *selectedClients*-@Inputina lista asiakkaita kuvan 29 rivillä 8, sillä tavoitteena oli käsitellä useampaa ja vain yhtä asiakasta täysin samalla tavalla.

```
<client-search #search
  [configType]="configType"
  [filterByIds]="existingIds"
  (selected)="setValue($event)">
</client-search>

<special-reason-dropdown
  [selectedClients]="[value$ | async]">
</special-reason-dropdown>

<button
  [disabled]="canNotContinue$ | async"
  (click)="onConfirm()">Seuraava</button>
```

Kuva 29. Asiakashaku-komponentin refaktoroitu HTML-templaatti

Näin ollaan onnistuneesti refaktoroitu SpecialReasonService-palvelu ja SpecialReasonDropdown-komponentti lähes täysin reaktiivisiksi. Refaktoroinnissa onnistuttiin poistamaan useita aiemmin mainittuja RxJS:n ja Angularin työkalujen väärinkäytöksiä ja epäloogisuuksia. Ainakin omasta mielestäni refaktoroinnin tuloksena syntynyt uusi koodi on huomattavasti yksinkertaisempaa, luettavampaa ja muokattavampaa kuin aiemmin imperatiivisesti toteutettu koodi. Tässä uudessa refaktoroidussa toiminnallisuudessa valittu asiakas tai asiakkaat toimivat datavirran alkulähteenä, josta kaikki muut tarvittavat Observablet johdetaan. Tästä yhdestä alkulähteestä saadaan reaktiivisesti johdettua *showSpecialReasonDropdown$*-Observable, jonka mukaan näytetään alasvetovalikko kontekstin valitsemiseksi. Lisäksi saadaan johdettua reaktiivisesti myös isValidContextFailed$-Observable, jossa on tieto HTTP-kutsun onnistumisesta. Alasvetovalikkoon sidotaan Angularin reaktiivinen lomake, joka tarjoaa Observablet sen arvosta ja validiudesta. Näistä voidaan reaktiivisesti johtaa Seuraava-nappulaa hallitseva *canNotContinue$*-Observable. Kaikkien Observablejen tilaaminen tapahtuu templaatissa AsyncPipellä ja missään kohtaa ei ole tarvetta turvautua imperatiiviseen ohjelmointiin tai Promiseihin, jotka tekivät aiemmasta koodista vaikeaselkoista.

4 PÄÄTÄNTÖ

Tässä opinnäytetyössä on pyritty selventämään reaktiivisen ohjelmoinnin määritelmää ja sitä, miten se vertautuu perinteiseen imperatiiviseen ja proseduraaliseen ohjelmointiin. Lisäksi työssä on avattu sitä, miten reaktiivinen ja funktionaalinen ohjelmointi suhtautuvat toisiinsa ja sitä miten ne kulkevat käsi kädessä muun muassa RxJS-kirjastoa käytettäessä. Koodin luettavuutta ja ymmärrettävyyttä on pyritty parantamaan deklaratiivisemmalla koodilla, jollainen saavutetaan yhdistämällä reaktiivinen ohjelmointi ja kuvaavampi muuttujien nimeäminen. Reaktiiviseen ohjelmointiin ja erityisesti RxJS:ään on sisäänrakennettuna ajatus kuvaavista Observableista, joita pystytään rakentamaan toisia Observableja funktionaalisesti muokkaamalla ja yhdistämällä. Observablen määrittelystä pitäisi pystyä aina lukemaan, mistä se saa arvonsa ja milloin se muuttuu. Tällä tavalla järkeily koodin sisällöstä ja sen suorittamista toiminnoista on tarkoitus tehdä mahdollisimman ilmeiseksi.

Kehittämistavoitteena oli refaktoroida reaktiivisen ohjelmointityylin mukaiseksi yksi Angular-sovelluksen palvelu ja sitä käyttävät komponentit. Tässä suoriuduttiin mielestäni oikein hyvin ottaen huomioon, että refaktoroinnissa tulisi pyrkiä säilyttämään olemassa oleva toiminnallisuus mahdollisimman samanlaisena. Tämä on usein haastavaa, jos komponentit tai palvelut eivät ole erityisen hyvin toisistaan eriytettyjä. Tässäkin tapauksessa suurimmaksi haasteeksi muodostui se, että aiempaa koodia oli käytetty useammassa paikassa sovellusta ja esimerkiksi muuttujien nimien tai sisällön pienikin muuttaminen saattoi aiheuttaa ennalta arvaamattomia vaikutuksia muualla sovelluksessa. Tämän takia olikin tarpeellista tehdä raportoidun työn lisäksi jonkin verran muutoksia muuhun ohjelman koodiin, jotta SpecialReason-palveluun ja SpecialReasonDropdown-komponenttiin tehdyt muutokset eivät rikkoneet aiempaa toiminnallisuutta.

Haasteelliseksi osoittautui myös aiemman koodin lukeminen ja sen toiminnallisuuden järkeily. Tämä lienee ainakin yksi osoitus siitä, miten reaktiivisella ohjelmoinnilla voidaan parantaa ohjelmoijien tehokkuutta jatkossa, kun he joutuvat tekemään muutoksia olemassa olevaan koodipohjaan. Mielestäni pystyin kuitenkin tuomaan ilmiselviä parannuksia koodiin ja löytämään vanhasta koo-

dista elementtejä ja antisuunnittelumalleja, joita teoreettisessa käsittelyssä varoitettiin käyttämästä. Lopputuloksena on mielestäni merkittävästi yksinkertaisempi sekä paremmin Angularin ja RxJS:n työkaluja hyödyntävä tuotos kuin aiempi epäjohdonmukaisuuksia, toistoa ja luvalla sanoen kummallisia ratkaisuja sisältänyt vanha koodi.

Parhaista pyrkimyksistäni huolimatta aivan jokaisessa kohdassa uutta koodia en kuitenkaan pystynyt (tai nähnyt tarpeelliseksi) noudattaa täysin puhdasta reaktiivista ohjelmointimallia. Esimerkiksi Leshin (2016b) esittämässä mallissa nappulan klikkauksista tulisi tehdä Observableja *fromEvent()*-metodilla, mutta valmiissa refaktoroidussa koodissa säilytin aiemman mallin, jossa nappulan painaminen emittoi Subjectin sen *next()*-metodin avulla. Tässä ratkaisussa vastapainona toimi Angularin suhteellisen työläs button-elementtien haku HTML-templaatista ja niihin *fromEvent()*-metodien sitominen. Ohjelmoinnissa yleensäkin saman asian voi saada aikaan useammalla eri tavalla, ja aina edes reaktiivisempi tapa ei ole yksinkertaisin ja ymmärrettävin. Toivon mukaan kokemuksen ja tietämyksen karttuessa osaan jatkossa vieläkin paremmin arvioida eri paradigmoja kriittisesti valitessani käyttämiäni ohjelmointimalleja.

Toimeksiantona ollut refaktorointi suoritettiin siis onnistuneesti ja vanhan koodin perimmäinen toiminnallisuus saatiin säilytettyä, joskin tämä vaati hieman koodimuutoksia muuallekin sovellukseen. Toisaalta nämä koodimuutokset olivat melko pieniä ja ainakin omasta mielestäni tekivät myös näistä muista komponenteista yksinkertaisempia ja helpommin ymmärrettäviä. Toimeksiantajalle työ toimii esimerkkinä reaktiivisesta ohjelmoinnista sekä osoituksena siitä, miten sama toiminnallisuus voidaan saavuttaa yksinkertaisemmin reaktiivisesti kuin imperatiivisesti. Toivon mukaan työn tuloksista on apua tulevaisuudessa muillekin tiimin jäsenille, jotka voivat ottaa mallia tästä reaktiivisesta koodista ohjelmoidessaan omia toiminnallisuuksiaan. Reaktiivinen ohjelmointi oli työtä aloittaessa itselleni melko uusi ohjelmointimalli, mutta huomasin nopeasti sen tuottamat hyödyt. Angularissa on valmiiksi paketoituna RxJS-kirjasto sekä useita reaktiivisia työkaluja, joita oikeaoppisesti hyödyntämällä reaktiivista koodia on helppo kirjoittaa. Itse tulen varmasti jatkossakin suosimaan reaktiivista ohjelmointimallia erityisesti Angular-sovelluksissa, joissa sen käyttäminen on hyvin luontevaa ja mielestäni ehdottomasti mielekkäin vaihtoehto.

LÄHTEET

Angular. 2022a. Introduction to Angular concepts. WWW-dokumentti. Päivitetty 28.2.2022. Saatavissa: https://angular.io/guide/architecture [viitattu: 6.8.2023].

Angular. 2022b. Observables in Angular. WWW-dokumentti. Päivitetty 28.2.2022. Saatavissa: https://angular.io/guide/observables-in-angular [viitattu: 18.9.2023].

Angular. 2023a. Angular Roadmap. WWW-dokumentti. Päivitetty 3.5.2023. Saatavissa: https://angular.io/guide/roadmap [viitattu 6.8.2023].

@angular. 2023. Tviitti 12.1.2023. Twitter-mikroblogipalvelu. Tilapäivitys. Saatavissa: https://twitter.com/angular/status/1613611774678384640?lang=fi [viitattu 6.8.2023].

Bainomugisha, E., Carreton, A., van Cutsem, T., Mostinckx, S. & de Meuter, W. 2013. A Survey on Reactive Programming. *ACM Computing Surveys* 52. Verkkolehti. Saatavissa: http://dl.acm.org/citation.cfm?id=2501666 [viitattu 9.9.2023].

Carniato, R, 2021. What the hell is Reactive Programming anyway? WWW-dokumentti. Päivitetty 23.3.2021. Saatavissa: https://dev.to/this-is-learning/what-the-hell-is-reactive-programming-anyway-31p5 [viitattu 7.9.2023].

Elliott, C. 2011. Specification for a Functional Reactive Programming language. Puheenvuoro Stack Overflow -keskusteluyhteisössä 4.5.2011. Saatavissa: https://stackoverflow.com/questions/5875929/specification-for-a-functional-reactive-programming-language/5878525#5878525 [viitattu 7.9.2023].

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley.

Gavigan, D. 2018. The History of Angular. WWW-dokumentti. Päivitetty 3.4.2018. Saatavissa: https://medium.com/the-startup-lab-blog/the-history-of-angular-3e36f7e828c7 [viitattu 6.8.2023].

Krouse, S. 2019. The Misunderstood Roots of FRP Can Save Programming. WWW-dokumentti. Saatavissa: https://futureofcoding.org/essays/dctp [viitattu 5.11.2023].

Lesh, B. 2016a. Hot vs Cold Observables. WWW-dokumentti. Päivitetty 29.3.2016. Saatavissa: https://benlesh.medium.com/hot-vs-cold-observables-f8094ed53339 [viitattu 9.9.2023].

Lesh, B. 2016b. On The Subject Of Subjects (in RxJS). WWW-dokumentti. Päivitetty 10.12.2016. Saatavissa: https://benlesh.medium.com/on-the-subject-of-subjects-in-rxjs-2b08b7198b93 [viitattu 9.9.2023].

Lüdemann, C. 2019. Refactoring Angular Apps To Reactive Architecture. WWW-dokumentti. Päivitetty 24.12.2019. Saatavissa: https://christianlydemann.com/refactoring-angular-apps-to-reactive-architecture/ [viitattu 18.9.2023].

Mota, M. 2016. Getting Started with RxJS. WWW-dokumentti. Päivitetty 22.4.2016. Saatavissa: https://miguelmota.com/blog/getting-started-with-rxjs/ [viitattu 24.9.2023].

RxJS. 2023a. Introduction. WWW-dokumentti. Saatavissa: https://rxjs.dev/guide/overview [viitattu 9.9.2023].

RxJS. 2023b. Observable. WWW-dokumentti. Saatavissa: https://rxjs.dev/guide/observable [viitattu 9.9.2023].

RxJS. 2023c. Observer. WWW-dokumentti. Saatavissa: https://rxjs.dev/guide/observer [viitattu 9.9.2023].

RxJS. 2023d. RxJS Operators. WWW-dokumentti. Saatavissa: https://rxjs.dev/guide/operators [viitattu 9.9.2023].

RxJS. 2023e. Subscription. WWW-dokumentti. Saatavissa: https://rxjs.dev/guide/subscription [viitattu 9.9.2023].

RxJS. 2023f. Subject. WWW-dokumentti. Saatavissa: https://rxjs.dev/guide/subject [viitattu 9.9.2023].

Staltz, A. 2014. The introduction to Reactive Programming you've been missing. WWW-dokumentti. Päivitetty: heinäkuu 2014. Saatavissa: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754 [viitattu 7.9.2023].

Staltz, A. 2015. Why I cannot say FRP but I just did. WWW-dokumentti. Päivitetty 27.7.2015. Saatavissa: https://medium.com/@andrestaltz/why-i-cannot-say-frp-but-i-just-did-d5ffaa23973b [viitattu 7.9.2023].

Stovell, P. 2010. What is Reactive Programming? WWW-dokumentti. Päivitetty 4.1.2010. Saatavissa https://paulstovell.com/reactive-programming/ [viitattu 7.9.2023].

Taylor, W. 2019. RxJS: Hot, Cold, Finite, Infinite, Unicast and Multicast Observables explained. WWW-dokumentti Päivitetty 27.5.2019. Saatavissa: https://www.willtaylor.blog/rxjs-observables-hot-cold-explained/ [viitattu 10.9.2023].

Thakur, I. 2021. Understanding Angular. WWW-dokumentti. Päivitetty 1.8.2021. Saatavissa: https://medium.com/analytics-vidhya/understanding-angular-2775383eac99 [viitattu 6.8.2023].

Vardanyan, A. 2020a. RxJS in Angular: Part I. WWW-dokumentti. Päivitetty 27.2.2020. Saatavissa: https://indepth.dev/posts/1162/rxjs-in-angular-part-1 [viitattu 20.9.2023].

Vardanyan, A. 2020b. RxJS in Angular: Part II. WWW-dokumentti. Päivitetty 18.8.2020. Saatavissa: https://indepth.dev/posts/1316/rxjs-in-angular-part-ii [viitattu 23.9.2023].