Sampo Osmonen

Graafisten käyttöliittymien määrittelyn käsitteet sekä suora ja inkrementaalinen määrittely

Tietotekniikan pro gradu -tutkielma

11. kesäkuuta 2024

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Sampo Osmonen

Yhteystiedot: sampo.e.osmonen@student.jyu.fi

Ohjaajat: Jonne Itkonen ja Ville Tirronen

Työn nimi: Graafisten käyttöliittymien määrittelyn käsitteet sekä suora ja inkrementaalinen määrittely

Title in English: Concepts of graphical user interface definitions as well as direct and incremental definition styles

Työ: Pro gradu -tutkielma

Opintosuunta: Informaatiotekniikka

Sivumäärä: 90+15

Tiivistelmä: Tässä tutkimuksessa perehdytään graafisten käyttöliittymien määrittämisen komponenttikirjastojen avulla. Tutkimusmenetelmänä käytetään käsiteanalyysiä, jonka avulla tunnistetaan käsitteitä, joita käyttöliittymän määrittelyyn minimissään liittyy. Erityisen kiinnostavia ovat termit suora ja inkrementaalinen näkymämäärittely, jotka on tunnistettu aiemmin ohjelmointiyhteisössä, mutta joita ei juuri ole ennen tätä akateemisesti tutkittu. Näille käsitteille esitetään tässä tutkimuksessa funktionaaliset määritelmät, ja määritelmien pätevyyttä tutkitaan vertaamalla niitä useaan eri ohjelmistokirjastooon ja akateemiseen tutkimukseen. Lopuksi verrataan näiden kahden määrittelytyylin eroja käsiteanalyysillä, ja todetaan, että suora näkymämäärittely voi usein olla inkrementaalista määrittelyä joustavampi valinta käyttöliittymäkehityksessä.

Avainsanat: GUI, suora määrittely, inkrementaalinen määrittely, välitön piirtäminen, tilapohjainen piirtäminen, käsiteanalyysi

Abstract: This study explores defining graphical user interfaces using component libraries. The study uses concept analysis as its research method to identify the core concepts that are needed for defining user interfaces. Two especially interesting concepts are those of direct and incremental view definitions, both of which have been recognized earlier in the

programming community, but haven't been researched much academically before. This study proposes functional definitions for these concepts, and the validity of these definitions is analyzed by comparing them to several different programming libraries and research articles. Finally these two definition styles are compared using concept analysis and it's found that direct view definition can often be a more flexible choice than incremental definition for user interface programming.

Keywords: GUI, direct definition, incremental definition, immediate rendering, retained-mode rendering, conceptual analysis

Termiluettelo

GUI	Graafinen käyttöliittymä – tietoteknisen järjestelmän käyttöliittymä, joka sisältää visuaalisen esityksen ja interaktiiviset osiot, joita käyttäjä voi ohjata.
näkymä	Graafisen käyttöliittymän visuaalinen osa.
hetkellinen näkymä	Näkymän hetkellinen esitys, joka piirretään näytölle.
komponenttikirjasto	Ohjelmointikirjasto, jolla voidaan määrittää käyttöliittymiä koostamalla komponentteja komponenttipuiksi.
komponentti	Näkymän osa, joka piirretään tietyllä tavalla näytölle, ja joka voi sisältää omaa toimintalogiikkaansa. Engl. *component, widget*.
komponenttityyppi	Joukko komponentteja, joilla on tietyt ominaisuudet.
parametri	Komponentin vastaanottama tieto, joka muokkaa sen toimintaa.
argumentti	Komponentin vastaanottama tieto, joka muokkaa sen toimintaa.
staattinen	Asia, joka ei muutu ajan kuluessa. Esimerkiksi liikkumaton kuva näytöllä.
dynaaminen	Asia, joka muuttuu ajan kuluessa. Esimerkiksi käyttöliittymä, joka mukautuu käyttäjän syötteisiin.
deklaratiivinen	Määrittelytyyli, joka suoraan kuvaa haluttua tulosta.
imperatiivinen	Määrittelytyyli, jonka kuvaama tulos muodostetaan suorittamalla lista ohjeita.
suora määrittely	Näkymän määrittely yhtenä lausekkeena, jonka arvo riippuu ohjelman suorituksen aikaisista tapahtumista.
inkrementaalinen määrittely	Näkymän määrittely staattisena alkuversiona ja siihen kohdistuvina muutoksina tapahtumien yhteydessä.
käsite	Abstrakti malli, joka kuvaa jotain asiaa tai ilmiötä.
intensio	Jonkin käsitteen määritelmä, joka kuvaa, mitä asioita käsitteen sisältöön yleisesti kuuluu.

ekstensio Jonkin käsitteen kuvaama joukko kaikkia asioita, jotka käsitteeseen kuuluvat.

semantiikka Jonkin esityksen kuvaama abstrakti malli, joka ei suoraan riipu esityksestään. Esimerkiksi jokin algoritmi voidaan kuvata erilaisilla esityksillä eri ohjelmointikielillä, mutta sen toiminta ja semantiikka pysyvät silti samoina.

Kuviot

Kuvio 1. Muistilistanäkymä web-selaimessa 4
Kuvio 2. Mitä käsitteitä graafisen käyttöliittymän määrittelyyn liittyy? 11
Kuvio 3. Motif-komponenttikirjastolla toteutettu listanäkymä 31
Kuvio 4. Syöte- ja ulostulovirtojen kulku komposiitti-fudgetin *f1* >==< *f2* osien välillä 35
Kuvio 5. Komposiittikäyttöliittymä piirrettynä Fudgets-kirjastolla 35

Sisällys

1 JOHDANTO 1

2 TEORIATAUSTA SEKÄ TUTKIMUSKYSYMYKSET JA METODI 3
2.1 Graafiset käyttöliittymät 3
2.2 Tutkimusmenetelmä 10
2.3 Kirjallisuuskatsaus 13

3 GRAAFISTEN KÄYTTÖLIITTYMIEN MÄÄRITTELY 14
3.1 Näkymän käsitteet 14
3.2 Dynaamisen näkymän määrittely 20

4 GRAAFISTEN KÄYTTÖLIITTYMIEN HISTORIAA 28
4.1 1970- ja 1980-luvut – inkrementaaliset komponenttikirjastot 28
4.2 1990-luku – funktionaaliset komponenttikirjastot 32
4.2.1 Fudgets 33
4.2.2 Fruit 40
4.2.3 Muut kirjastot 44
4.3 2000-luvusta nykyhetkeen – suorat komponenttikirjastot 47

5 MÄÄRITTELYTAPOJEN EROT 58
5.1 Inkrementaalisen määrittelyn edut 58
5.2 Määrittelytapojen neutraalit erot 61
5.3 Suoran määrittelyn edut 62

6 POHDINTA JA YHTEENVETO 69
6.1 Tutkimuksen taustat 69
6.2 Tutkimuksen tulokset 71
6.3 Tutkimuksen luotettavuus 72

LÄHTEET 74

VERTAISARVIOIMATTOMAT LÄHTEET 80

LIITTEET 85
A Koodinäytteitä 85

1 Johdanto

Graafiset käyttöliittymät (GUI) ovat yleinen tapa kommunikoida tietoteknisten järjestelmien kanssa, ja niistä on tullut kiinteä osa monen arkea. Graafisen käyttöliittymän kehittäminen on kuitenkin usein monimutkainen prosessi, johon on tarjolla on monia erilaisia lähestymistapoja ja teknologioita (Lumsden 2002; Valaer ja Babb 1997). Valinnat näiden välillä vaikuttavat väistämättä kehitysprosessiin ja sitä kautta saattavat vaikuttaa myös ohjelmistoprojektin onnistumiseen. Ohjelmistokehitysalalla valintoja GUI-teknologioiden välillä tehdään usein intuitiivisesti, mutta myös analyyttisemmästä lähestymisestä voi olla etua (Lumsden 2002). GUI-teknologian ominaisuuksien ymmärtäminen on väistämättä lähtökohta sille, että sen soveltuvuutta käyttötarkoitukseen voidaan arvioida, ja että sitä osataan käyttää tehokkaasti. Tästä huolimatta erilaisia GUI-kehityksen lähestymisiä ymmärtämään ja vertailemaan pyrkiviä tutkimuksia on suhteellisen vähän. Ohjelmistoprojektin GUI-teknologioita valitessa onkin helppo verrata eri ratkaisujen teknisiä ominaisuuksia, kuten tarjolla olevien rakennusosien määrää tai alustatukea, mutta syvällisemmät ohjelmistoarkkitehtuuriin vaikuttavat tekijät voivat jäädä tiedostamatta, tai päätökset tehdään intuition varassa (Lumsden 2002).

Tämän tutkimuksen tarkoitus on kehittää ymmärrystä GUI-teknologioihin liittyvistä käsitteistä. Erityisesti se keskittyy GUI-komponenttikirjastoihin, eli ohjelmointikirjastoihin, jotka tarjoavat ohjelmoijalle valikoiman peruskomponentteja käyttöliittymän rakentamiseen (Jansen 1998). Lisäksi tämä tutkimus keskittyy erityisesti kahteen GUI-arkkitehtuurin lähestymisvalintaan, joille esitän termejä *suora määrittely* ja *inkrementaalinen määrittely* luvussa 3. Suorassa määrittelyssä käyttöliittymä esitetään kuvauksena, joka kertoo, miten käyttöliittymä pitäisi piirtää näytölle kunakin ajan hetkenä riippuen siitä, mitä tapahtumia ohjelman suorituksen aikana on esiintynyt. Inkrementaalisessa määrittelyssä puolestaan käyttöliittymä esitetään alkuversiona ja muutoksina, joita siihen myöhemmin tehdään. Tutkimuksessa sivutaan myös muita GUI:n määrittelyyn liittyviä arkkitehtuurivalintoja kuten imperatiivista ja deklaratiivista määrittelytapaa – imperatiivisessa määrittelyssä käyttöliittymä koostetaan antamalla käskyjä sen muodostamiseen pienemmistä osista, deklaratiivisessa määrittelyssä puolestaan annetaan kuvaus valmiista lopputuloksesta. Nämä erilaiset lähestymistavat ovat

olleet kauan rinnakkain käyttöliittymien ohjelmoinnissa, ja siksi onkin hyödyllistä ymmärtää niitä tarkemmin, jotta niitä voidaan tutkia lisää.

Tämä tutkimus käsittelee GUI-arkkitehtuurin menetelmiä käsiteanalyysin keinoin. Aluksi pyritään analysoimaan GUI-kehityksen käsitteitä, sitten keskitytään tarkemmin suoran ja inkrementaalisen GUI-määrittelyn ominaisuuksiin, osin myös käytännön kannalta. Tutkimuskysymykset ovat:

1. Mitä käsitteitä graafisen käyttöliittymän ohjelmalliseen määrittelyyn liittyy?
2. Mitä ovat suora ja inkrementaalinen käyttöliittymän näkymän määrittely?
3. Mitä etuja suoralla ja inkrementaalisella näkymän määrittelyllä on toisiinsa nähden?

Luvussa 2.1 esittelen taustakäsitteet, joita tämän tutkimuksen aiheen ymmärtäminen edellyttää. Esittelen myös tutkimuksen pääaiheena olevat suoran ja inkrementaalisen määrittelyn käsitteet alustavasti. Luvussa 3 suoritetaan varsinainen käsiteanalyysi, jossa tutkitaan suoran ja inkrementaalisen määrittelyn käsitteitä ja muodostetaan niille funktionaaliset määritelmät. Luvussa 4 luodaan katsaus graafisten komponenttikirjastojen historiaan sekä akateemisiin tutkimuksiin käyttöliittymien määrittämisestä ja verrataan, ovatko edellisessä luvussa esitetyt suoran ja inkrementaalisen määrittelyn käsitteet päteviä niiden yhteydessä. Luvussa 5 verrataan suoraa ja inkrementaalista määrittelyä keskenään käsiteanalyysin keinoin ja pyritään myös löytämään eroja, jotka voidaan laskea jommallekummalle määrittelytavalle eduksi. Luvussa 6 kootaan yhteen tämän tutkimuksen tulokset, verrataan niitä aiempaan kirjallisuuteen sekä esitellään mahdollisia jatkotutkimuskohteita.

2 Teoriatausta sekä tutkimuskysymykset ja metodi

Tässä luvussa esitellään ensin graafisten käyttöliittymien ohjelmointiin liittyviä peruskäsitteitä, sitten tutkimuksen metodivalintaa ja tehtyä kirjallisuuskatsausta.

2.1 Graafiset käyttöliittymät

Graafinen käyttöliittymä on visuaalinen esitys, jonka on tarkoitus mahdollistaa käyttäjän ja tietoteknisen järjestelmän vuorovaikutus (Jansen 1998; Martinez 2011). Esitys koostuu osista, jotka voivat olla interaktiivisia tai ei-interaktiivisia. Käyttäjä tunnistaa interaktiiviset osat ja hyödyntää niitä suorittamaan haluamansa toiminnot järjestelmässä, jolloin käyttöliittymä tavallisesti myös muuttuu vasteena käyttäjän syötteeseen. Graafisessa käyttöliittymässä tarvittavia toimintoja ovat siis visuaalisen esityksen muodostus, käyttäjäsyötteiden vastaanotto ja visuaalisen esityksen kyky muuttua ajan ja käyttäjäsyötteiden funktiona. Näillä elementeillä käyttäjä voi saada käyttöliittymän kautta päivittyvää tietoa ja syöttää komentoja järjestelmälle.

Koska nykyiset näyttölaitteet koostuvat yleensä pikseleistä, on niiden tai vastaavan näyttöteknologian suora manipulaatio suoraviivaisin tapa toteuttaa käyttöliittymän visuaalinen esitys. Moderneille käyttöjärjestelmille on kuitenkin kehitetty *GUI-komponenttikirjastoja*, jotka abstrahoivat tämän prosessin korkeammalle tasolle (Jansen 1998). Näiden kirjastojen toiminta perustuu ajatukseen, että jonkin ohjelmiston käyttöliittymän visuaalinen esitys, *näkymä*, koostuu itsenäisistä osista, *komponenteista* (engl. *component, widget*) (Adelsberger, Setzer ja Walkingshaw 2018; Carlsson ja Hallgren 1993; Hanus ja Kluß 2009). Tarjoamalla valikoiman peruskomponentteja ja mahdollisesti tavan tyylitellä niitä, erilaiset komponenttikirjastot mahdollistavat joustavan ja helpon tavan määritellä käyttöliittymän näkymiä. Esimerkkejä komponenttikirjastoista ovat muun muassa GTK, Qt ja Flutter (ks. “The GTK Project - A free and open-source cross-platform widget toolkit” 2023; “Qt | Tools for Each Stage of Software Development Lifecycle” 2023; “Flutter - Build apps for any screen” 2023).

Komponenttikirjasto antaa käyttäjälleen mahdollisuuden lisätä näkymään yhden tai useampia komponentteja saatavilla olevista eri *komponenttityypeistä*. Komponenteille voidaan mah-

dollisesti asettaa myös muokattavissa olevia ominaisuuksia, *parametreja*, jolloin yksi komponenttityyppi voi kattaa useampia käyttötarkoituksia (esim. “QLineEdit Class | Qt Widgets 5.15.13” 2023; “TextField class - material library - Dart API” 2023). Jonkin parametrin tietystä arvosta käytetään sanaa *argumentti*. Joidenkin komponenttien sisälle voidaan asettaa toisia komponentteja, mikä pätee erityisesti niin sanottujen *asemointikomponenttien* (engl. *layout component*) tapauksessa (esim. “Layouts in Flutter | Flutter” 2023). Komponentteja sisäkkäin ja rinnakkain asettelemalla voidaan muodostaa puumainen rakenne, joka kuvaa halutunlaista näkymän koostumusta ja ulkoasua (Hanus ja Kluß 2009).

Yksi ensimmäisiä asioita näkymää komponenttikirjaston avulla määritettäessä on *staattisen* eli muuttumattoman näkymän esittäminen komponenteista koostamalla. Tähän voidaan käyttää lähestymistapana imperatiivista tai deklaratiivista määrittelyä. Imperatiivisessa määrittelytavassa kirjaston tarjoamista komponenttityypeistä luodaan komponentit, joihin saadaan sitten viite, esimerkiksi funktiokutsun paluuarvona (Overton 1997). Viitteiden avulla komponentteja voidaan manipuloida ja asettaa toistensa sisälle. Mahdollisten viimeistelyvaiheiden jälkeen komponenttikirjasto piirtää komponentit näytölle. Tästä määrittelytavasta esimerkin antaa listaus 2.1, jossa JavaScript-kielellä määritetään HTML-DOM-rajapinnan(“Introduction to the DOM” 2020) kautta muistilistasovellusta kuvaava näkymä.

Listauksessa 2.1 kuvattu koodi tuottaa web-selaimen käsittelemänä kuvion 1 kaltaisen näkymän.

Kuvio 1: Muistilistanäkymä web-selaimessa

Näkymä voidaan määrittää myös deklaratiivisesti. Tällöin määritettäviin komponentteihin ei saada viitteitä manipulointia varten, vaan valmiin näkymän komponenteista ja niiden argumenteista sekä komponenttien puumaisesta hierarkiasta annetaan kuvaus, jonka perusteella komponenttikirjasto muodostaa näkymän esityksen näytölle (Silva ja Campos 2012). Tästä esimerkkinä toimii HTML-esityksellä määritetty deklaratiivinen versio aiemmasta muistilis-

```
const root = document.createElement("div");
const item1 = document.createElement("div");
item1.textContent = "Maito";
root.appendChild(item1);
const item2 = document.createElement("div");
item2.textContent = "Vehnäjauho";
root.appendChild(item2);
const input = document.createElement("input");
input.type = "text";
root.appendChild(input);
const button = document.createElement("button");
button.textContent = "Lisää";
root.appendChild(button);
```

Listaus 2.1: Imperatiivinen muistilistanäkymän määritelmä (JavaScript, DOM)

tanäkymästä listauksessa 2.2.

On perusteltua väittää, että nämä kaksi määrittelytapaa ovat ainoat suoraviivaiset tavat näkymän esittämiseen ohjelmointikielen avulla. Kun esitetään jotain näkymärakennetta, tai yleisemmin mitä tahansa algoritmia, voidaan kuvata haluttu lopputulos, mutta myös tapa, joka lopputuloksen saavuttamiseksi halutaan valita (Fahland ym. 2009). Deklaratiivinen määrittely kertoo suoraan ohjelmakoodin kielellä miltä näkymän tulisi näyttää, ja imperatiivinen määrittely puolestaan kuvaa paitsi epäsuorasti lopullisen näkymän, myös työvaiheet sen

```
<div>
  <div>Maito</div>
  <div>Vehnäjauho</div>
  <input type="text" />
  <button>Lisää</button>
</div>
```

Listaus 2.2: Deklaratiivinen muistilistanäkymän määritelmä (HTML)

muodostamiseksi.

Edelliset esimerkit kuvasivat staattisia näkymiä. Staattisuus tarkoittaa, etteivät näkymät muutu ajan suhteen vaan pysyvät aina samanlaisina. Graafisten käyttöliittymien on kuitenkin käytännössä aina pystyttävä *dynaamisuuteen*, eli muutokseen ajan ja tapahtumien myötä, kuten aiemmin jo esitin. Staattinen käyttöliittymä ei kykene muuttumaan vastauksena käyttäjän syötteisiin tai muihin tapahtumiin, minkä vuoksi se voi ainoastaan vastaanottaa tietoa käyttäjältä, mutta ei voi antaa käyttäjälle interaktiivista palautetta tai käyttäjän tarpeiden mukaan päivittyvää tietoa. Tällainen käyttöliittymä on periaatteessa mahdollinen, mutta ei yleensä kovin hyödyllinen minkäänlaisen tietoteknisen järjestelmän käyttöön. Tämän vuoksi käytännössä kaikki käyttöliittymät ovat jollain tavalla dynaamisia. Dynaamisuuden tasot voivat käyttöliittymissä toki vaihdella – joissain käyttöliittymissä on hyvin rajallinen määrä interaktiomahdollisuuksia ja erilaisia tiloja, joihin käyttöliittymä voi päätyä, toisissa taas on monia interaktiivisia elementtejä, erilaisia syötetapoja ja lukemattomia mahdollisia käyttöliittymän tiloja.

Dynaamisen käyttöliittymän määrittelyyn tarvitaan siis staattisen näkymän määrittelyn rinnalle myös tapa kuvata käyttöliittymän muutosta. Muutoksen voidaan ajatella tapahtuvan käyttäjäsyötteiden ja muiden tapahtumien, kuten ajastinten, vaikutuksesta (Martinez 2011). Tämän vuoksi käyttöliittymän tilasiirtymien esittämisessä yleensä viitataankin tavalla tai toisella ohjelman suorituksen aikaisiin tapahtumiin.

Eräs ratkaisu käyttöliittymän dynaamisuuden esittämiseen on *inkrementaalinen määrittely* – näkymän alkutila kuvataan staattisesti aivan kuten aiemmin, mutta tämän lisäksi määritetään ohjeet siihen, miten näkymän tilaa tulee muokata kunkin tapahtuman yhteydessä. Esimerkki tästä voidaan rakentaa aiemman imperatiivisen määrityksen esimerkin pohjalle. Esimerkissä näkymän komponentteihin on saatavilla viitteet, ja komponenttikirjasto voi tarjota käskyt, joilla näitä instansseja voidaan nyt muokata, lisätä tai poistaa näkymästä ohjelman suorituksen edetessä. Tämä on tehty listauksessa 2.3. Muistilistan käyttöliittymään on nyt lisätty interaktiivinen toiminto ”Lisää”-painikkeen aktivointiin. Tapahtuman yhteydessä ajetaan siihen jollain tavalla liitetty funktio, niin sanottu takaisinkutsufunktio (engl. *callback function*), esimerkissä nimellä `addItem` (ks. Carlsson ja Hallgren 1993; Overton 1997). Esimerkin funktio lukee tekstikentän senhetkisen tekstin, luo uuden komponentin vastaavalla tekstillä

ja lisää sen näkymään. Funktio määrittää siis muutoksen, joka käyttöliittymään tehdään lisäystapahtuman yhteydessä. Muutosta kuvataan käskyillä, jotka muokkaavat näkymän osia.

Toinen lähestyminen dynaamisuuden toteuttamiseen on *suora määrittely*. Siinä näkymä kuvataan yhtenä kokonaisuutena, joka voi riippua ohjelman suorituksen aikaisista tapahtumista, ja tämän kuvauksen avulla näkymän visuaalinen esitys voidaan muodostaa minä vain ohjelman suorituksen hetkenä. Eräs tapa tähän on *reaktiivisen* ohjelmointityylin hyödyntäminen näkymän määritykseen (Salvaneschi ym. 2014). Tästä esimerkki on listaus 2.4, jossa JavaScriptiin perustuvalla Svelte-ohjelmointikielellä määritellään aiempi lisäystoiminto muistilistanäkymään. Svelteä käytettäessä `script`-osiossa määritellään JavaScript-muuttujat ja käyttäjäsyötteisiin reagoivat takaisinkutsufunktiot, ja sen jälkeen määritellään näkymä deklaratiivisesti HTML:ää muistuttavalla *sapluunakielellä* (engl. *templating language*). Sapluunakielessä `each`-direktiivi laajentuu koodia tulkitessa peräkkäiseksi jonoksi komponentteja, jotka vastaavat määritelmälle annetun listan alkioita `each`-määritelmän sisältä löytyvän määritelmän mukaisiksi komponenteiksi muunnettuina. Aaltosulkujen sisältö laajentuu JavaScript-arvoiksi, tässä tapauksessa tekstiksi, joka esitetään näkymässä. Svelte yhdistää nämä sapluunakielen ominaisuudet reaktiivisuuteen – kun jotain JavaScript-muuttujaa päivitetään, muutokset heijastetaan automaattisesti myös käyttöliittymän näkymään sapluunan määrittämällä tavalla. Sapluunakielen `bind`-määritelmä kytkee halutun komponentin parametrin kaksisuuntaisesti JavaScript-muuttujaan niin, että muuttujan muokkaus päivittää komponenttia, ja toisaalta komponentti voi käyttäjäsyötteen saatuaan päivittää muuttujaa. Kielen reaktiivisuuden ansiosta näkymäsapluunan laajennettu arvo, ja siten käyttäjälle esitettävä näkymä, päivittyvät automaattisesti vastaamaan tilatietoa, jota sapluuna käyttää. Näin näkymä voidaan määrittää yhdessä yhtenäisessä osassa. Tämä eroaa edellisestä esimerkistä, jossa osa käyttöliittymän määritelmästä oli esitetty muutoksina tapahtumien laukaisemien takaisinkutsufunktioiden sisällä.

Tämä tutkimus pyrkii erityisesti tarkastelemaan suoran ja inkrementaalisen määrittelyn semantiikkoja. Useat perinteiset komponenttikirjastot perustuvat inkrementaaliseen malliin, esimerkiksi Motif ja Qt, mutta uudempien komponenttikirjastojen keskuudessa suora malli on alkanut saavuttaa suosiota (ks. "A First Motif Program" 2023; "Trail: Creating a GUI With Swing (The Java™ Tutorials)" 2023; "React" 2023; "Flutter - Build apps for any sc-

```
const root = document.createElement("div");
const item1 = document.createElement("div");
item1.textContent = "Maito";
root.appendChild(item1);
const item2 = document.createElement("div");
item2.textContent = "Vehnäjauho";
root.appendChild(item2);
const input = document.createElement("input");
input.type = "text";
root.appendChild(input);
const button = document.createElement("button");
button.textContent = "Lisää";
root.appendChild(button);

const addItem = () => {
  const item = document.createElement("div");
  item.textContent = input.value;
  root.insertBefore(item, input);
  input.value = "";
};
button.onclick = addItem;
```

Listaus 2.3: Inkrementaalisesti määritelty dynaaminen muistilistanäkymä (JavaScript, DOM)

```
<script>
  let input = "";
  let items = [];

  const addItem = () => {
    items = [...items, input];
    input = "";
  };
</script>

<div>
  <div>Maito</div>
  <div>Vehnäjauho</div>
  {#each items as item}
    <div>{item}</div>
  {/each}
  <input type="text" bind:value={input} />
  <button on:click={addItem}>Lisää</button>
</div>
```

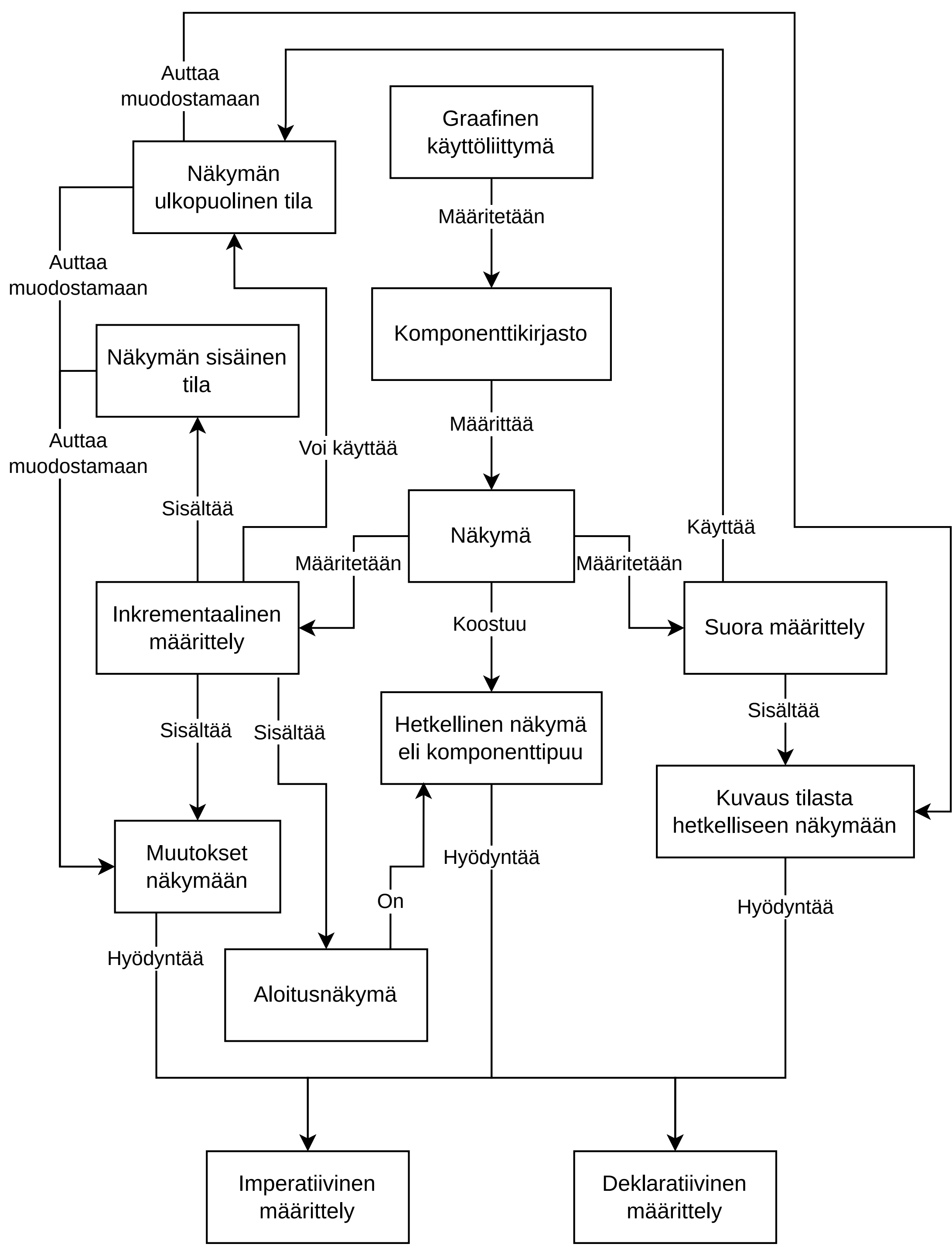
Listaus 2.4: Suorasti määritelty dynaaminen muistilistanäkymä (Svelte)

reen” 2023), mitä esitellään tarkemmin luvussa 4. Mitkä tekijät vaikuttavat kummankin määrittelytyylin suosioon, ja mitä etuja tyyleillä on toisiinsa nähden? Tämä tutkimus pyrkii tutkimaan aihetta selventämällä suoran ja inkrementaalisen määrittelytyylin suhteita.

Tässä luvussa kuvattuja käyttöliittymämäärittelyn käsitteitä ja niiden suhteita on esitetty kuviossa 2. Yläkäsitteenä on graafinen käyttöliittymä, jonka määrittelyyn kaikki muut termit liittyvät. Käytännössä ohjelmistokehityksessä komponenttikirjasto näyttäytyy rajapintana, jonka kautta käyttöliittymä määritetään kirjaston komponentteja hyödyntäen. Komponenttikirjaston avulla esitetään käyttöliittymän visuaalinen osa, näkymä. Koko suorituksen aikainen dynaaminen näkymä koostuu hetkellisistä näkymistä, joita voidaan kuvata komponenttikirjaston komponenttipuina. Tämän tutkimuksen dynaaminen näkökulmasta näkymä voidaan määrittää kahdella tavalla, suorasti tai inkrementaalisesti. Inkrementaalisessa määrittelyssä hyödynnetään näkymän sisäistä tilaa, staattista aloitusnäkymää ja mahdollisesti näkymän ulkoista tilaa. Suorassa määrittelyssä puolestaan annetaan lauseke, joka kuvaa komponenttipuun eli hetkellisen näkymän muodostamista muodostamista sovelluksen senhetkisen tilan perusteella. Näitä määrittelytapoja esitellään tarkemmin luvussa 3. Lisäksi imperatiivinen ja deklaratiivinen määrittely ovat osina useita muita näkymän osien määrityksiä ja niiden semantiikkaa, mikä näkyy hyvin tästä käsitekartasta.

2.2 Tutkimusmenetelmä

Tämän tutkimuksen keskeinen tutkimusmenetelmä on käsiteanalyysi. Käsiteanalyysiä on varsinkin filosofiassa käytetty paljon erilaisten käsitteiden luonteen tarkempaan ymmärtämiseen (Kaijanaho 2017; Järvinen 2004; Margolis ja Laurence 2020). Se soveltuu kuitenkin myös tietotekniikan alalle, ja sitä onkin käytetty monissa tutkimuksissa onnistuneesti. Turingin laskennallisuusmääritelmän (Turing 1937) voidaan katsoa olevan käsiteanalyysi (Kaijanaho 2017). Liebermanin suorittama objektien perintämekaniikkojen vertailu (Lieberman 1986) taas on klassinen filosofinen käsiteanalyysi. Lisäksi alalta löytyy monia muitakin tutkimuksia, jotka ovat onnistuneesti tuottaneet uutta tietoa ja teorioita pääosin käsiteanalyysin keinoin, kuten Cook (2009), Wand ja Wang (1996), Cook, Hill ja Canning (1989), Koskela ym. (2013), J. Hughes (1989) ja Stefik ja Hanenberg (2014).



Kuvio 2: Mitä käsitteitä graafisen käyttöliittymän määrittelyyn liittyy?

Käsiteanalyysi todettiin tähän tutkimukseen sopivaksi, sillä tutkimuksen aihealueen käsitteitä ei ole aiemmin tutkittu käsiteanalyysin keinoin akateemisesti kovinkaan paljoa ainakaan tekemäni kartoituksen perusteella. Varsinkin inkrementaalista ja suoraa näkymämäärittelyä vertailevasta julkisesta keskustelusta suurin osa löytyi akateemisen keskustelun ulkopuolelta (esim. “About the IMGUI paradigm · ocornut/imgui Wiki” 2023a; “Immediate Mode Model/View/Controller” 2023a; “Moxie: Incremental Declarative UI in Rust | Hacker News” 2020; “Virtual DOM is pure overhead (2018) | Hacker News” 2020; Levien 2020; Harris 2020). Toisaalta käyttöliittymien määrittelyn yleisistä formalismeista tutkimusta oli tehty akateemisestikin (esim. Courtney ja Hudak 2004; Czaplicki ja Chong 2013). Joka tapauksessa suoran ja inkrementaalisen käyttöliittymämäärittelyn termien selventäminen ja niiden suhteiden määrittäminen muihin käyttöliittymän käsitteisiin voi tukea jatkotutkimuksen tekemistä ja myös muuta keskustelua aiheesta. Lisäksi sen kautta voidaan löytää uusia teorioita tai alikäsitteitä (Kaijanaho 2017; Järvinen 2004).

Käsiteanalyysin käyttöön liittyy myös haasteita (Kaijanaho 2017; SCHROETER 2004; Margolis ja Laurence 2020). Ei ole varmaa, onko käsitteille lopulta edes mahdollista antaa täydellisiä määritelmiä (D. Chalmers 2015; Gettier 1963; Margolis ja Laurence 2020), ja jopa niiden luonne on jossain määrin väittelyn kohteena (D. Chalmers 2015; Gettier 1963; Margolis ja Laurence 2020). Käsitteen määritelmän hyväksyminen perustuukin argumentaatioon ja intuitioon vetoamiseen (Kaijanaho 2017). Käsiteanalyysi vaatii myös vuorovaikutusta saadakseen vahvistusta muiltakin kuin kirjoittajalta (Kaijanaho 2017). Tässä tutkimuksessa tärkeässä roolissa ovat sen ohjaajat, joiden kanssa voidaan haastaa kirjoittajan ajatuksia. Lopuksi käsiteanalyysin tuloksia olisi usein syytä tutkia myös empiirisesti niiden käytännöllisen arvon todentamiseksi (Kaijanaho 2017). Toisaalta käsitteiden määrittely ja käsiteanalyysi ovat välttämättömiä osia tutkimuksen tekemisessä – muuten ei voida saada kaikkea tietoa eikä edes tietää, mitä tutkitaan (Kaijanaho 2017; Margolis ja Laurence 2020; D. J. Chalmers 1996; Jackson 1998). Tästä syystä koin, että tutkimuksen aiheen tarkempi määrittely voi olla hyödyllistä ennen sen tutkimista yksityiskohtaisemmin empiirisellä menetelmällä. Käsiteanalyysi voi muuttaa aiheen ymmärrystä tai sitä, mitä aiheesta kannattaa tutkia. Tässä tutkimuksessa käsite-termiä käytetään klassisen määritelmän mukaisesti.

2.3 Kirjallisuuskatsaus

Tätä tutkimusta varten keräsin tietoa GUI-komponenttikirjastoista kartoittamalla sekä olemassa olevista käytännön ratkaisuja että akateemisia tutkimuksia. Lisäksi perehdyttiin komponenttikirjastojen kehitykseen ja historiaan. Tämä oli tarpeellista, jotta voitiin muodostaa kattava ja mahdollisimman objektiivinen ymmärrys aiheesta ja erilaisista komponenttikirjastoissa esiintyvistä käsitteistä.

Hakulähteinä käytin ensisijaisesti ACM Digital Library, IEEE Xplore ja Google Scholar -sivustoja. Hyödynsin myös Wikipedia-tietosanakirjaa ja Google-hakuja perehtyessäni useisiin aiheisiin. Käytin seuraavaksi listaamiani hakutermejä kaikissa näissä palveluissa, ja hakutermeissä käytin yleisesti toimivia hakusemantiikkoja, joissa lainausmerkit (tässä korvattu heittomerkeillä hakutermien sisällä) merkitsevät, että kyseisten sanojen pitäisi esiintyä tarkalleen annetussa muodossa osana tekstiä, "OR-avainsana määrittää, että kumpi vain sen puolisista hakusanoista kelpaa, ja sulut ryhmittelevät usean hakusanan yhdeksi. Hakutermejä olivat muun muassa seuraavat: "gui OR graphical programming", "gui architecture", ”graphical user interface’ OR gui declarative", "reactive graphical OR gui programming", ”graphical user interface’ OR gui semantics", "graphical OR gui programming concepts", "gui OR ’graphical user interface’ concepts", "(’graphical user interface’ OR gui) (framework OR method OR pattern OR architecture)", "imperative OR declarative gui", "(declarative OR reactive) programming empirical", "gui frameworks", "gui libraries", "platform-indenpendent gui", "declarative gui", "functional gui", ”graphical user interface’ OR gui (definition OR analysis OR components)", "gui history".

3 Graafisten käyttöliittymien määrittely

Luvussa 2.1 esiteltiin ohjelmallisen GUI-määrittelyn toimintaa ja pohjustettiin loppututkimuksessa käytettävät graafisten käyttöliittymien peruskäsitteet. Tässä luvussa perehdytään GUI-määrittelyyn tarkemmin käsiteanalyysin keinoin. Aluksi tarkennetaan yleisiä käsitteitä ja erityisesti imperatiivisen ja deklaratiivisen määrittelyn ominaisuuksia, sitten tarkastellaan suoran ja inkrementaalisen määrittelyn käsitteitä.

3.1 Näkymän käsitteet

Analyysin alussa on tarpeen määrittää tarkemmin joitain luvussa 2.1 alustavasti esitettyjä käsitteitä. Kuten mainittu, graafinen käyttöliittymä muodostuu visuaalisesta esityksestä, näkymästä. Näkymä on näytöllä esitettävä kuva, joka näyttää käyttäjälle tietoa ja käyttöliittymän toimintoja. Joillain sovelluksilla voi olla useita toisistaan irrallisia näkymiä, esimerkiksi useita työpöytäympäristön ikkunoita (Martinez 2011), mutta tässä tutkimuksessa keskitytään yksittäisen näkymän ominaisuuksiin. Kuten aiemmin todettiin, näkymä tavallisesti muuttuu ajan myötä käyttäjän syötteiden ja muiden tapahtumien vaikutuksesta. Näkymä on siis yleensä dynaaminen objekti, jolla voi olla useita hetkellisiä versioita. Tässä tutkimuksessa termiä *näkymä* käytetään tarkoittamaan koko käyttöliittymän esitystä sen käytön aikana ja termiä *hetkellinen näkymä* tarkoittamaan yhtä hetkellistä versiota käyttöliittymästä. *Näkymän määritelmä* on käsite, jolla viitataan ohjelmakoodiin, joka määrittää näkymän. Se ilmoittaa kaikki mahdolliset hetkelliset versiot näkymästä, eli se määrittää *näkymätyypin*. Useita näkymätyypin mukaisia näkymiä, instansseja, on mahdollista suorittaa eri järjestelmissä tai saman järjestelmän sisällä. Näkymän määritelmä siis ilmaisee näkymätyypin *intension* ja suoritettavat näkymäinstanssit sen *ekstension*.

Kuten luvussa 2.1 esitettiin, komponenttikirjastolla luotu näkymä muodostuu komponenteista. Komponentti on näkymän aliosa, joka piirretään näytölle tietyllä tapaa sen tyypin ja mahdollisten argumenttien mukaan. Komponentteja voivat olla esimerkiksi tekstiä esittävät komponentit, painikkeet, tekstisyötekentät ja asemointikomponentit. Joidenkin komponenttien sisälle voidaan asettaa yksi tai useampia muita komponentteja eli lapsikomponentteja, jol-

loin ne tavallisesti esitetään visuaalisesti jossain vanhempikomponentin sisällä (esim. "Layouts in Flutter | Flutter" 2023). Komponenttien suhteista syntyy tällöin puurakenne, joka on nähtävissä myös näkymän visuaalisessa esityksessä (Hanus ja Kluß 2009). On mahdollista, että vanhempi- tai lapsikomponentti tai molemmat voivat vaikuttaa siihen, miten komponentit asemoidaan toistensa suhteen. Komponenttikirjastot tarjoavat yleensä myös neutraaleja asemointikomponentteja, joiden tärkein tehtävä on niille annettujen alikomponenttien asemointi jollain tietyllä tavalla, esimerkiksi ylhäältä alas tai vasemmalta oikealle. Komponentit voivat vastaanottaa myös parametreja, jotka muokkaavat niiden toimintaa ilman, että pienen muutoksen vuoksi tarvitsee käyttää erillistä komponenttityyppiä. Lopuksi myös käyttäjäkohtaisten komponenttien määrittely on usein mahdollista (esim. "Quick Start – React" 2023). Tällöin komponentti toimii eräänlaisena modulaarisena alinäkymänä, jota voidaan käyttää useassa eri paikassa, ja jolle voidaan mahdollisesti syöttää sen toimintaa muokkaavia argumentteja.

Kuten luvussa 2.1 esitettiin, komponenttikirjaston avulla tehtäviin näkymämääritelmiin vaikuttavat myös kirjaston tukemat ohjelmointikielet sekä kirjaston tarjoamat rajapinnat komponenttien määrittelyyn ja hallintaan. Luonnollisesti myös kirjaston tarjoamat komponentit sekä mahdollisuudet mukautettujen komponenttien määrittämiseen voivat vaikuttaa siihen, miten kirjaston avulla määritelmiä tehdään. Kirjasto voi tukea useampaa ohjelmointikieltä, ja näkymän eri osia voidaan määrittää eri kielillä – esimerkiksi inkrementaalisessa mallissa alkunäkymä voidaan määrittää eri kielellä kuin näkymään tapahtumien yhteydessä tehtävät muutokset, kuten esimerkissä 2.4. Yleiskäyttöinen Turing-täydellinen kieli voi tukea käytännössä mitä vain ominaisuuksia kirjastossa halutaan tarjota, mutta toisaalta tällainen kieli ei välttämättä sovellu tarjoamaan käytännöllisintä rajapintaa kirjastolle – triviaalina esimerkkinä tästä voi toimia x86 assembly-kieli, jossa ei juuri ole korkean tason ominaisuuksia, jotka mahdollistaisivat GUI-näkymien käytännöllisen määrittelyn, mutta jolla silti olisi mahdollista määrittää mikä tahansa haluttu näkymä. Toisaalta useat suositut ohjelmointikielet tukevat puumaisten tietorakenteiden ja tietueiden käytännöllisen esittämisen, mikä tekee myös näkymien komponenttipuiden esittämisestä vaivatonta. Ei-yleiskäyttöiset kielet GUI-ohjelmoinnissa puolestaan ovat usein *täsmäkieliä* (engl. domain-specific language), jotka on suunniteltu nimenomaan käyttöliittymien esittämiseen. Tällaisen kielen etuna voi olla sen selkeys ja käytännöllisyys, sekä myös se, että kielen ei tarvitse olla Turing-täydellinen, mil-

lä voi olla etuja esimerkiksi tietoturvan tai ohjelman suorituskyvyn kannalta. Esimerkkinä tällaisesta kielestä toimii HTML (ks. “HTML Standard” 2020).

Komponenttikirjasto voi myös mahdollistaa esimerkiksi animaatioiden määrittelyn, käyttäjäsyötteiden simuloinnin, vapaamuotoisen datan tallentamisen komponentteihin tai tietojen hakemisen ja kopioimisen toisilta komponenteilta. Kirjasto voi tukea käyttäjän omien mukautettujen komponenttien määrittämistä, mikä on tärkeää varsinkin, jos näkymien määrittelykieli ei tue jotain muuta abstrahoinnin metodia kuten aliohjelmia tai funktioita. Mukautetut komponentit tai muu abstrahointimenetelmä mahdollistavat mielivaltaisen monimutkaisten komponenttirakenteiden yksinkertaisen esityksen näkymän määritelmässä sekä niiden uudelleenkäytön.

Komponenttikirjastoihin liittyy siis lukuisia piirteitä, jotka voivat vaikuttaa näkymien määrittämiseen. Toisaalta myös komponenttikirjaston tukemien ohjelmointikielten valinta voi vaikuttaa valintaan imperatiivisen ja deklaratiivisen näkymän esityksen välillä. Esimerkiksi jos käytetty ohjelmointikieli ei tarjoa mitään käytännöllistä tapaa esittää puumaisia tietorakenteita, voi näkymän komponenttipuun kuvailu deklaratiivisesti tällöin haastavaa, ja toisaalta jos kieli on kokonaan deklaratiivinen, voi sillä olla haastavaa kuvata imperatiivisia käskyjä näkymän muodostamiseksi. Muilta osin kirjaston tukemat ohjelmointikielet eivät juuri voi vaikuttaa näkymän määritelmän semantiikkaan, sillä mikäli kielessä ei ole joitain erityisiä rajoitteita, pystyy se kuvaamaan kaikki kirjaston tarjoamat rakenteet joko imperatiivista tai deklaratiivista lähestymistä käyttäen. Loput erot kielten välillä ovat siten syntaktisia. Tässä tutkimuksessa erilaisten ohjelmointikielten vaikutusta esitellään jonkin verran esimerkkien kautta.

Komponenttikirjastojen tai ohjelmointikielten erityisominaisuuksien sijaan tässä tutkimuksessa keskitytään tarkimmin näkymämäärityksen peruskäsitteisiin, käsitteisiin, joita minimissään tarvitaan näkymän määrittämiseen. Aluksi tässä luvussa perehdytään imperatiiviseen ja deklaratiiviseen määrittelyyn, sillä valinta niiden välillä vaikuttaa keskeisesti hetkellisen näkymän esityksen semantiikkaan, kuten luvussa 2.1 todettiin. Tämän jälkeen keskitytään suoran ja inkrementaalisen näkymämäärittelyn analyysiin, mikä on tämän tutkimuksen keskeinen aihe.

Luvussa 2.1 esiteltiin jo varsin kattavasti imperatiivisen ja deklaratiivisen näkymän määrityksen semanttisia eroja. Tiivistetysti todettuna imperatiivisessa määrittelyssä keskeistä on näkymän komponenttien manipulointi käskyillä niin, että haluttu rakenne saadaan lopulta muodostettua, deklaratiivisessa määrittelyssä taas näkymän komponenttirakenne kuvataan yhtenä esityksenä käyttäen kirjaston ja ohjelmointikielen tarjoamaa syntaksia. Listaukset 2.1 ja 2.1 antavat varsin havainnollistavan esimerkin siitä, millaisia eroja käytetty ohjelmointikieli voi tuoda näkymän määritelmään – deklaratiivinen esimerkki käyttää näkymän määrittämiseen HTML-syntaksia, joka on näkymien määrittämiseen suunniteltu täsmäkieli. Imperatiivinen esimerkki puolestaan käyttää yleiskäyttöistä JavaScript-kieltä. Esimerkeistä voidaan helposti nähdä, että tässä tapauksessa deklaratiivisen esimerkin esitys pystyy kuvailemaan saman näkymän suoraviivaisemmin kuin imperatiivinen. Toisaalta imperatiivisen esimerkin käyttämä kieli soveltuu näkymien määrittämisen lisäksi yleisiin laskennallisiin tehtäviin, kun taas deklaratiivisen esimerkin syntaksi ei sellaisenaan vaikuta tähän kovin käytännölliseltä. Ohjelmointikielen valinta voi siis vaikuttaa huomattavasti näkymän määritelmän syntaksiin. Mutta jotta imperatiivisen ja deklaratiivisen määrittelyn eroja voitaisiin vielä tarkastella lähemmin, on syytä pyrkiä vähentämään ohjelmointikielten luomia eroja.

Imperatiivisen ja deklaratiivisen esimerkin eroja voidaan tasoittaa esittämällä ne vaihtoehtoisissa muodoissa. Imperatiivinen esimerkki voidaan muuttaa listauksen 3.1 esittämästi kuvitteelliseen komponenttikirjastoon, jolloin esimerkkiin jäävät vain imperatiivisen mallin välttämättömät piirteet – komponenttien luonti, komponenttien viitteiden tallennus nimettyihin muuttujiin, ja komponenttien välisten suhteiden määritys. Komponentti-instanssien luonti jonkinlaisen käskyn osana on välttämätön osa imperatiivista lähestymistä, sillä se on ainoa tapa saada tietyn komponentin viite tallennettua muuttujaan. Viitteet taas tarvitaan, että komponentteja on myöhemmin tarvittaessa mahdollista manipuloida käskyillä. Esimerkissä `component`-funktion kolmas parametri määrittää vanhempikomponentin, jonka sisälle luotava uusi komponentti lisätään mahdollisten aiempien lapsikomponenttien perään. Mikäli komponenttien viitteitä ei ollenkaan säilöttäisi muuttujiin, olisi niiden suhteet ilmoitettava jollain deklaratiivisella puumaisella datarakenteella, jolloin esimerkki ei enää olisi imperatiivinen ja muistuttaisi enemmän seuraavaa esimerkkiä 3.2. Tämän vuoksi viitteiden tallennus katsotaan olennaiseksi osaksi imperatiivista näkymän rakennusta. Toisaalta taas kuvitteellisessa esimerkissä tekstisisältöjen ja komponenttien parametrien määritys tapahtuvat deklara-

```
root = component("div", {}, null);
   item1 = component("div", {"text": "Maito"}, root);
   item2 = component("div", {"text": "Vehnäjauho"}, root);
   input = component("input", {"type": "text"}, root);
   button = component("button", {"text": "Lisää"}, root);
// END root
```

Listaus 3.1: Vaihtoehtoinen imperatiivinen muistilistanäkymän määritelmä (JavaScript, kuvitteellinen komponenttikirjasto)

```
const view = {component: "div", params: {}, children: [
   {component: "div", params: {}, children: "Maito"},
   {component: "div", params: {}, children: "Vehnäjauho"},
   {component: "input", params: {type: "text"}, children: []},
   {component: "button", params: {}, children: "Lisää"}
]}
```

Listaus 3.2: Vaihtoehtoinen deklaratiivinen muistilistanäkymän määritelmä

tiivisesti, tietueita hyödyntäen `component`-funktion toisena parametrina. Näihin tietoihin ei tarvita myöhempää viitettä, joten niiden ilmoittaminen deklaratiivisesti poistaa tarpeen ylimääräisille muuttujille ja käskyille. Lopuksi voidaan huomata, että ohjelmakoodia sisentämällä voidaan ilmaista lukijalle komponenttien hierarkiaa samalla tavalla kuin aiemmassa deklaratiivisessa näkymän määrittelyssä oli tehty.

Aiempi deklaratiivinen esimerkki puolestaan voidaan esittää ilman erillistä täsmäkieltä JavaScriptin tarjoamilla taulukko- ja tietuerakenteilla listauksen 3.2 mukaisesti. Näkymän kuvaus muodostuu tällöin tietueista, jotka kuvaavat jotain komponenttityyppiä, komponentin parametreja ja komponentin lapsikomponentteja tai tekstisisältöä. Samoin kuin HTML-esimerkissä, näkymän konkreettiset komponentit luodaan kuvauksen pohjalta automaattisesti komponenttikirjaston toimesta.

Kahta edellistä esimerkkiä vertailemalla voidaan nähdä, että imperatiivisen mallin pelkistetty ja deklaratiivisen mallin vaihtoehtoinen ratkaisu ovat aiempia käytännön esimerkkejä

lähempänä toisiaan. Esitettyjen versioiden merkkimäärät ovat melko samanlaisia, ja käytettyjen funktioiden ja tietueavainten nimiä sekä deklaratiivisen esityksen syntaksia vaihtamalla kumpi vain esitys voi olla lyhyempi. Ainoa varsinainen ero on, että imperatiivisessa mallissa komponenttityypeistä luodaan komponenteista instanssit, joiden viitteet on tallennettava muuttujiin, jos niitä myöhemmin halutaan manipuloida. Imperatiivisessa mallissa siis vaaditaan yksi semanttinen käsite enemmän näkymien määrittämiseksi, mutta tämä ei väistämättä nosta esityksen merkkimäärää. Toisaalta imperatiivisesta esitystavasta voidaan todeta, että näkymä esitetään suorana jonona erilaisia käskyjä, kun taas deklaratiivinen esitys muodostaa syntaktisesti samanlaisen puumaisen rakenteen kuin sen kuvaama näkymä. Ero voitiin kuitenkin luettavuuden näkökulmasta kuroa sisentämällä imperatiivisen esityksen rivit niin, että ne muistuttavat näkymän puumaista rakennetta, ja myös lisäämällä kommenttirivi merkitsemään juurikomponentin päättymistä. Silti ohjelmointikielen semantiikan näkökulmasta imperatiivinen näkymän määritelmä koostuu ei-hierarkkisista peräkkäisistä käskyistä, mikä saattaa vaikeuttaa optimointien tekoa määritelmää tulkattaessa – komponenttikirjasto ei voi esimerkiksi tulkata ja piirtää komponenttihierarkiassa rinnakkain olevia komponentteja samanaikaisesti, sillä ohjelmointikielen käskyillä on määrätty järjestys, ja siitä poikkeaminen voi tuottaa sivuvaikutuksia, esimerkiksi aliohjelmien tai funktiokutsujen kautta, niin, että järjestyksestä poikkeaminen tuottaisi väärän lopputuloksen. Mikäli käskyjen järjestys kuitenkin rajoitetaan kirjaston tai ohjelmointikielen toimesta siten, että komponentteja voidaan muuttaa vain silloin kun ne luodaan ensimmäistä kertaa tai kun niiden sisälle lisätään lapsikomponentteja heti luonnin jälkeen, on imperatiivinen lähestyminen jo huomattavan yhdenmukainen deklaratiivisen vastineensa kanssa.

Voidaan siis todeta, että imperatiivisella ja deklaratiivisella lähestymistavalla voidaan erilaisista semantiikoista huolimatta päästä lähestulkoon identtiseen esitysmuotoon. Käytännön kannalta lähestymistavan valintaa merkittävämmät tekijät staattisen näkymän määritelmän kannalta ovatkin siis käytetty ohjelmointikieli sekä komponenttikirjaston rajapinta – niistä riippuen näkymän määritelmän syntaksi voi olla hyvin erilainen, kuten listauksista 2.2 ja 2.1 nähdään. Näistä ensimmäinen muistuttaa enemmän äskettäin esitettyjä pelkistettyjä näkymän esityksiä (listaukset 3.1 ja 3.2). Keskeistä sille ja pelkistetyille esityksille on, että niissä turhien syntaktisten elementtien ja merkkien määrä on suhteellisen vähäinen, ja että komponenttien hierarkiat näkymässä ilmenevät esityksestä syntaktisesti sisennyksinä. Tur-

hien syntaktisten elementtien välttäminen lieneekin etu staattisia näkymiä kuvatessa – useiden hierarkkisia dokumentteja kuvaavien *merkintäkielten* (engl. *markup language*) voidaan nähdä noudattavan näitä periaatteita ja käyttävän deklaratiivista määrittelyä, esimerkkeinä TeX-sukuiset kielet ja SGML-vaikutteiset kielet kuten XML ja HTML (ks. Silva ja Campos 2012).

Valinta imperatiivisuuden ja deklaratiivisuuden välillä esiintyy myös dynaamisten näkymien muutoksia määritettäessä, mitä käsitellään seuraavassa luvussa.

3.2 Dynaamisen näkymän määrittely

Edellisessä luvussa analysoitiin staattisten näkymien esitystapoja, mutta kuten aiemmin todettiin, pelkkä staattinen näkymä harvoin riittää käyttöliittymäksi. Näkymien dynaamisen muutoksen mallintaminen vaatii ajan käsitteen huomioimista suorasti tai epäsuorasti näkymän määritelmässä pelkän yhden hetkellisen näkymän lisäksi. Näkymän voidaan katsoa olevan ajan funktio, sillä se muuttuu ajan edetessä ja sovellusta käytettäessä. Tämä ei kuitenkaan ole mahdollinen tapa määrittää näkymää, sillä näkymän on pystyttävä reagoimaan ennalta määräämättömiin käyttäjäsyötteisiin ja muihin tapahtumiin – muuten se muistuttaisi enemmän esimerkiksi videota, joka etenee ennalta määrätysti tilasta toiseen. Tapahtumalla tarkoitan tietuetta, joka sisältää tapahtuman tyypin, tapahtuman ajankohdan jossain muodossa sekä mahdollista muuta tapahtumaan liittyvää tietoa. Täten näkymän määritelmä voidaan ajatella eräänlaisena funktiona, jonka parametrina on lista ohjelman tähänastisista suorituksenaikaisista tapahtumista, ja joka palauttaa hetkellisen näkymän. Näkymäfunktion yleinen muoto on siis:

$$[\text{tapahtuma}] \rightarrow \text{hetkellinen näkymä} \tag{3.1}$$

Tällaisella funktiolla voidaan määrittää näkymän käyttäytyminen kutsumalla funktiota iteratiivisesti jokaisen tapahtuman jälkeen kronologisella listalla siihenastisia tapahtumia ja päivittämällä näkymän visuaalinen esitys vastaamaan funktion tulosta. Voidaan väittää, että diskreetti tapahtumalista ei esitä yhtäjaksoisia muutoksia näkymässä optimaalisesti, esimerkiksi tapauksessa, jossa jokin kuvio liikkuu käyttöliittymässä tasaisesti ajan edetessä. Kuitenkin

tapahtumien tiheyttä lisäämällä voidaan nostaa käyttöliittymän päivitystaajuutta ja saavuttaa haluttu tarkkuus yhtäjaksoisten tapahtumien kuvaamiseen, joten diskreetti lista on riittävä malli käyttöliittymän tapahtumien esittämiseen.

Näkymäfunktio vastaanottaa listan kaikista ohjelman tapahtumista, sillä muodostettava näkymä saattaa riippua mistä vain aiemmista tapahtumista. Esimerkki siitä, miten tällainen näkymäfunktio voisi toimia, on ulkoasultaan edellisiä esimerkkejä muistuttava muistilistanäkymä liitteen A listauksessa 1. Siinä kuvitteellisen komponenttikirjaston komponenttien tapahtumiin linkitetään `eventBindings`-direktiivin avulla tyyppinimet, joiden perusteella suorituksenaikaiset tapahtumat voidaan erottaa toisistaan. Tapahtumat käydään näkymää muodostettaessa läpi järjestyksessä vanhimmasta uusimpaan, ylläpitäen samalla prosessia tukevaa tietoa eli tilaa. Esimerkiksi tapahtuman kautta saatu syötekentän viimeisin arvo on oltava tiedossa, kun halutaan käsitellä tapahtuma, joka lisää syötekentän arvon muistilistaan. Näin tapahtumat käsittelemällä voidaan muodostaa toivotunlainen hetkellinen näkymä.

Myös kaksi liitteen A listauksesta 1 poikkeavaa lähestymistä näkymän muodostamiseen tapahtumalistasta ovat mahdollisia – lista voidaan käsitellä viimeisimmästä tapahtumasta uusimpaan, tai osia siitä voidaan iteroida monta kertaa. Uusimmasta vanhimpaan iteroidessa voidaan jättää huomiotta vanhemmat tietyn tyyppiset tapahtumat, kun tiedetään, että ne eivät vaikuta luotavaan näkymään, sillä jokin myöhempi tapahtuma on yliajanut ne. Toisaalta taas näkymän muodostaminen näin voi olla vähemmän intuitiivista, sillä näkymän osat joudutaan päättelemään ja muodostamaan käänteisessä järjestyksessä kuin missä niihin vaikuttavat tapahtumat tapahtuivat. Esimerkiksi edellisessä esimerkissä lista-alkio pitäisi lisätä, kun löydetään ensimmäinen lisäystapahtumaa edeltävä syötekentän muutostapahtuma, ja se pitäisi lisätä listan loppuun, ei alkuun. Kolmas lähestymistapa on iteroida tapahtumalistaa useammin kuin kerran, jolloin tapahtumien välillä ei tarvitse säilöä mitään tilaa kuten annetussa esimerkissä, vaan jos tapahtumaan reagointi vaatii tietoa myös toisista tapahtumista, voidaan kaikki merkitykselliset tapahtumat hakea ja muodostaa näkymä niiden perusteella. Toisaalta tämä vaatii tapahtumalistan jatkuvaa iterointia, mikä voi olla paitsi työlästä toteuttaa, myös ohjelman suorituskyvyn näkökulmasta tehottomampaa.

Edellä esitelty tapa näkymän muodostamiseen täyden tapahtumalistan kautta jokaisen tapahtuman yhteydessä ei kuitenkaan ole käytännöllinen ratkaisu. Ensinnäkin lista voi kasvaa ra-

joittamattomasti ohjelman suorituksen edetessä, mikä asettaa haasteita ohjelman muistinkäytölle. Mikään ohjelma ei tarvitse täyttä historiaa kaikista aiemmista tapahtumista näkymänsä muodostamiseen. Tämä on havainnollistettavissa esimerkillä: jos liitteen A esimerkin 1 muistilistanäkymään lisättäisiin listan tyhjennyspainike, riittäisi listan alkioita pääteltäessä huomioida lisäystapahtumista vain ne, jotka tulivat viimeisimmän tyhjennystapahtuman jälkeen, joten täyden tapahtumalistan säilöminen olisi tarpeetonta. Lisäksi voidaan todeta, että tapahtumalistaa ei tarvita ollenkaan hetkellistä näkymää muodostettaessa, kun näkymään vaikuttavat hyödylliset tiedot säilötään tapahtumien välillä ulkopuoliseen, muokattavissa olevaan tilaan. Tämä nähdään liitteen A esimerkissä 1, jossa tapahtumalistaa iteroidessa tieto siihenastisista lista-alkioista ja viimesimmästä syötekentän arvosta pidetään erillisenä tilana. Tällöin tämä tila ja tieto nykyisestä tapahtumasta riittävät näkymän seuraavan vaiheen sekä seuraavan tilan muodostamiseen. Tällaisen muokattavan tilan hyödyntäminen mahdollistaa aiempien tapahtumien suodattamisen ja tiivistämisen muotoon, joka on tapahtumalistaa käytännöllisempi seuraavan näkymän muodostamisessa. Kun uusi näkymä ja tila muodostetaan jokaisen tapahtuman yhteydessä, muuttuu näkymän määritelmä muotoon:

$$(\text{tila}, \text{tapahtuma}) \rightarrow (\text{hetkellinen näkymä}, \text{tila}) \tag{3.2}$$

Näkymä muodostetaan tällöin kutsumalla näkymäfunktiota iteratiivisesti jokaisen tapahtuman jälkeen nykyisellä tilalla ja viimeisimmällä tapahtumalla. Funktio palauttaa uuden hetkellisen näkymän sekä uuden tilan, joka toimii syötteenä seuraavaa hetkellistä näkymää laskettaessa.

Tämä tilaa hyödyntävä ratkaisu muistuttaa toiminnaltaan hyvin paljon liitteen A esimerkkiä 1, jossa tapahtumat iteroitiin yksi kerrallaan läpi muokaten tilaa, jonka pohjalta näkymä lopulta muodostettiin. Ainoa ero on, että uudentyyppinen näkymäfunktio ei tarvitse täyttä tapahtumalistaa jokaisen hetkellisen näkymän muodostamiseen, sillä tapahtumien ja tilan iteraatio saavutetaan näkymäfunktiota toistuvasti tapahtumien yhteydessä kutsumalla. Aiemmin esitetyistä kolmesta tapahtumien iterointitavasta ensimmäinen, eli iterointi kerran vanhimmasta uusimpaan tapahtumaan, on siis käytännössä hyödyllisin, sillä se mahdollistaa tilan käyttämisen ja tapahtumien käsittelyn yksi kerrallaan, kun taas käänteinen tai toistuva

iterointi vaatisivat jokaisen hetkellisen näkymän muodostamiseen täyttä siihenastista tapahtumalistaa. Tapahtumista koostetun tilan hyödyntäminen ja uusimpien tapahtumien käsittely niiden tapahtuessa on siis muistinkäytön asettamien rajoitteiden kannalta paras ratkaisu. Toisaalta se on myös intuitiivinen tapa käsitellä näkymän muokkautumista esimerkiksi käyttäjävuorovaikutusten myötä – jokaisen tapahtuman yhteydessä muodostetaan uusi versio tilasta ja näkymästä, mahdollisesti aiempaa tilaa hyödyntäen.

Ehdotan siis kaavaa 3.2 näkymän yleiseksi määritelmäksi. On nähtävissä, että mikä vaan näkymä voidaan määritellä sen avulla, ja se on myös suoraviivainen ymmärtää sekä tietokoneen muistinkäytön kannalta tehokas ratkaisu.

Näkymän yleisestä määritelmästä on johdettavissa myös kaksi käytännön muunnosta:

1. Tilan ja näkymän määritelmien erottaminen toisistaan.
2. Näkymän määrittely hyödyntäen edellisiä hetkellisiä näkymiä.

Esitän näille muunnoksille termejä *suora määrittely* ja *inkrementaalinen määrittely*, vastaavassa järjestyksessä. Seuraavaksi kuvaan nämä määrittelytavat tarkemmin.

Suora määrittely perustuu oivallukseen, että jos uusi tila johdetaan edellisen tilan ja viimeisimmän tapahtuman perusteella, voidaan uusi hetkellinen näkymä johtaa tästä uudesta tilasta sen sijaan, että käytettäisiin edellistä tilaa ja viimeisintä tapahtumaa. Käytännössä hetkellinen näkymä voidaan siis muodostaa pelkän tietynhetkisen tilan perusteella. Samalla tilan päivitys voidaan erottaa näkymän päivityksestä. Tällöin näkymän määritelmä koostuu seuraavista funktioista:

$$
\begin{aligned}
(\text{tila}, \text{tapahtuma}) &\rightarrow \text{tila} \\
\text{tila} &\rightarrow \text{hetkellinen näkymä}
\end{aligned}
\tag{3.3}
$$

Näkymä muodostetaan laskemalla uusi tila jokaisen tapahtuman yhteydessä edellisen tilan (tai alkutilan) ja uuden tapahtuman pohjalta, sitten laskemalla uusi haluttu hetkellinen näkymä tämän tilan perusteella. Tämä määrittely on käytännössä suoraviivaisempi versio aiemmasta näkymän yleisestä määritelmästä, ja siitä toimii esimerkkinä liitteen A listaus 2. Aiempiin ostoslistanäkymän versioihin verrattuna näkymään on lisätty teksti, joka ilmoit-

taa, kuinka monta kertaa alkioita on lisätty. Tämä voitaisiin toki tässä laskea myös alkioiden määrästä, mutta ei esimerkiksi, jos alkioita olisi mahdollista myöhemmin poistaa.

Inkrementaalinen määrittely puolestaan perustuu oivallukseen, että hetkellisen näkymän muodostamisessa voidaan hyödyntää muun tilan lisäksi myös edellistä hetkellistä näkymää. Täten se voidaan nähdä näkymän yleisen määritelmän versiona, jossa muodostettavia hetkellisiä näkymiä käytetään myös syötetilana. Tällöin on luontaista määritellä näkymä muutosten kautta edellisiä versioita muokkaamalla sen sijaan, että jokainen hetkellinen näkymä muodostettaisiin alusta alkaen jostain eri tyyppisestä tilasta. Edellistä hetkellistä näkymää voidaan hyödyntää halutulla tavalla tai olla hyödyntämättä sitä lainkaan, jolloin näkymän muodostus toimii enemmän suoraa määrittelyä muistuttavalla tavalla. Inkrementaalinen näkymän määritys muodostuu seuraavanlaisesta funktiosta:

$$(\text{hetkellinen näkymä}, \text{tila}, \text{tapahtuma}) \rightarrow (\text{hetkellinen näkymä}, \text{tila}) \tag{3.4}$$

Näkymä muodostetaan tällöin laskemalla jokaisen tapahtuman yhteydessä uusi hetkellinen näkymä edellisen näkymän ja tilan (tai alkutilan ja alkunäkymän) sekä tapahtuman pohjalta. Samalla määritetään seuraavana syötteenä toimiva tila. Uusi hetkellinen näkymä voidaan määrittää muokkauksina edellisen näkymän pohjalta. Voidaan huomata, että edellinen näkymä saattaa sisältää valtaosan tilasta, jota vaaditaan uuden hetkellisen näkymän muodostamiseen. On kuitenkin tilanteita, joissa kaikkea näkymän muodostamiseen vaadittavaa tilaa ei voida säilöä suoraan itse näkymään, ellei sallita mielivaltaisen datan tallentamista näkymään ei-visuaalisesti, mitä käsitellään myöhemmin tässä tutkimuksessa luvussa 5.1. Huomionarvoista on, että näkymän ja tilan laskentaa ei inkrementaalisessa määrittelyssä ole hyödyllistä erottaa toisistaan kuten suorassa määrittelyssä, sillä sekä edellistä että uutta tilaa voidaan potentiaalisesti hyödyntää hetkellistä näkymää muodostettaessa. Molempien tilan versioiden tarvitsee siis tarvittaessa olla saatavilla näkymänmuodostusvaiheessa, ja siten uusi tila voidaan myös laskea samassa yhteydessä. Inkrementaalisesta määrittelystä toimii esimerkkinä muistilistanäkymä liitteen A listauksessa 3. Esimerkissä komponenteille on lisätty yksilöivä `id`-tieto, jotta niihin voidaan helpommin viitata, ja näkymän muokkaus tapahtuu deklaratiivisesti ja muuttumattomasti (engl. *immutably*) funktioita hyödyntämällä. Syötekentän arvo

asetetaan tapahtumien yhteydessä, joskin tässä esimerkissä olisi helppoa antaa syötekentän päivittää oman arvonsa itse, jolloin sitä riittäisi manuaalisesti muokata vain silloin, kun syötekenttä halutaan palauttaa tyhjään arvoonsa alkion lisäämisen yhteydessä.

Esittelin tässä suoran ja inkrementaalisen määrittelyn kahtena erillisenä päävaihtoehtona dynaamisen näkymän määrittämiseen, ja on perustettua väittää, että ne ovat myös ainoat vaihtoehdot lajissaan. Suora määrittely vastaa kaavassa 3.2 esitettyä näkymän yleistä määritelmää, mutta vain kahteen osaan jaettuna. Inkrementaalinen määrittely voi vaikuttaa siltä, että se eroaa merkittävämmin näkymän yleisestä määritelmästä, mutta sekin on lopulta vain yleisen määritelmän variaatio – siinä edellinen hetkellinen näkymä on osa tilaa, jonka pohjalta seuraava hetkellinen näkymä muodostetaan. Riittää, että komponenttikirjaston avulla edellisestä hetkellistä näkymää voidaan hakea tietoja, kuten esimerkiksi syötekentän senhetkinen arvo, jolloin osa uuden hetkellisen näkymän muodostamiseen tarvittavasta tilasta voidaan pitää näkymän itsensä sisällä. Suora ja inkrementaalinen määrittely ovat siis kaksi versiota näkymän määrittelystä, joista toisessa tilaa ei säilötä näkymän komponenttirakenteeseen, ja toisessa taas säilötään.

Voidaan kuitenkin huomata, että suora ja inkrementaalinen määrittely ovat teknisesti varsin lähellä toisiaan, ja niiden ero on ennen kaikkea ajattelutavassa. Kunhan komponenttikirjasto tukee näkymän datarakenteen käsittelyä, myös näkymän yleisessä tai suorassa määrittelyssä voidaan hyödyntää inkrementaalista lähestymistä kahdentamalla näkymä osaksi tilaa ja sitten hyödyntämällä sitä seuraavan näkymän muodostamisessa. Toisaalta taas inkrementaalisessa määrittelyssä voidaan halutessa olla käyttämättä edellistä hetkellistä näkymää seuraavan näkymän muodostamisessa mikäli edellisen hetkellisen näkymän komponentit poistetaan näkymästä ja uusi hetkellinen näkymä piirretään tyhjästä lähtötilanteesta. Tällöin kaikki näkymän muodostamiseen tarvittava tila säilötään suoran määrittelyn tavoin jonain näkymän sisäisestä datarakenteesta eroavana tyyppinä. Siispä suora ja inkrementaalinen määrittely ovat teknisesti tarkasteltuna yhteneviä, ja ero näkymän määrittelyssä tulee käytetystä arkkitehtuurivalinnasta – määritetäänkö näkymä mahdollisuuksien mukaan edellisiä versioita hyödyntäen, vai johdetaanko se aina kokonaisuudessaan eri tyyppisestä tilasta. Viittaan jatkossakin kyseisillä termeillä tähän arkkitehtuurivalintaan, en mihinkään tiettyyn näkymämääritelmän tekniseen muotoon.

```
const view = state => {
  const root = document.createElement("div");
  for (let itemName of state.items) {
    const item = document.createElement("div");
    item.textContent = itemName;
    root.appendChild(item);
  }
  return root;
};
```

Listaus 3.3: Yleisen listanäkymän määrittely tilan funktiona imperatiivisesti (JavaScript, DOM)

Aiemmin todetusti suoran ja inkrementaalisen määrittelyn rinnalla eräs merkittävä lähestymisvalinta on näkymän määrittely imperatiivisesti tai deklaratiivisesti. Näitä määrittelytapoja voidaan hyödyntää myös osana suoran ja inkrementaalisen määrittelyn eri vaiheita. Inkrementaalisessa lähestymisessä staattinen alkunäkymä voidaan määrittää molemmilla tavoilla, mistä luvussa 2.1 on nähtävissä useita esimerkkejä. Näkymän muutokset ovat myös esitettävissä molemmin tavoin – listauksessa 2.3 muutokset esitettiin antamalla komentoja, jotka muokkasivat näytöllä esitettäviä objekteja, liitteen A listauksessa 3 puolestaan muutokset ilmoitettiin funktioiden tuloksina, eikä mitään dataobjekteja muokattu. Toisaalta taas suorassa määrittelyssä näkymä voidaan esittää tilan parametrina joko deklaratiivisesti, kuten listauksessa 2.4, tai imperatiivisesti luomalla näkymän komponentit ja muokkaamalla niitä käsky käskyltä vastaamaan annettua tilaa, kuten esimerkissä 3.3. Inkrementaalinen ja suora näkymän määrittely sekä deklaratiivisen ja imperatiivisen ohjelmointitavan käyttö ovat siis toisistaan toisistaan riippumattomia käsitteitä, ja niitä voidaan yhdistää halutulla tavalla käytetyn komponenttikirjaston, ohjelmointikielen sekä myös ohjelmoijan tarpeiden mukaan.

Valinta inkrementaalisen ja suoran näkymän mallinnuksen välillä on hyvin keskeinen käyttöliittymämäärityksen semantiikkaan vaikuttava tekijä. Lähestymiset johtavat keskenään erilaisiin loogisiin polkuihin, joiden myötä käyttöliittymän dynaaminen käyttäytyminen mallinnetaan. Myös imperatiivinen ja deklaratiivinen ohjelmointityyli vaikuttavat määrityksen

semantiikkaan eli siihen, muodostetaanko hetkellinen näkymä vaiheittaisilla käskyillä vai suoraan yhtenäisenä kuvauksena. Toisaalta kuten aiemmin nähtiin, molempien lähestymisten ohjelmakoodi on jäsennettävissä muistuttamaan rakenteeltaan sitä hierarkkista komponenttipuuta, josta hetkellinen näkymä koostuu, joten voidaan väittää, ettei tämän valinnan merkitys ei ole yhtä suuri kuin dynaamisen käyttäytymisen mallinnustavan valinnalla. On perusteltua väittää, että nämä kaksi ovat merkittävimmät näkymän semantiikkaan vaikuttavat tekijät – hetkellisen näkymän määritys ja käyttöliittymän dynaamisen käytöksen määritys kattavat koko käyttöliittymän toiminnallisuuden. Muut valinnat, kuten esimerkiksi ohjelmointikieli, eivät usein vaikuta siihen abstraktiin käyttöliittymän intensioon, jota ohjelmakoodilla kuvataan, ainoastaan kyseisen koodin syntaksiin. Toisaalta nämä tekijät voivat vaikuttaa myös siihen, millä semantiikoilla käyttöliittymä on luontaista mallintaa kyseisessä ympäristössä. Näiden tekijöiden vaikutuksiin näkymän määrittelyssä perehdytään seuraavassa luvussa käytännön tasolla tarkemmin.

4 Graafisten käyttöliittymien historiaa

Tässä luvussa esitellään GUI-komponenttikirjastojen kehitystä edellisissä luvuissa esitettyihin käsitteisiin peilaten. Näiden esimerkkien avulla pyritään osoittamaan edellisen luvun käsiteanalyysin pätevyys. Erityisesti komponenttikirjastoja tarkastellaan suoran ja inkrementaalisen näkymämäärittelyn näkökulmasta. Komponenttikirjastot rajataan kirjastoihin, joilla voidaan interaktiivisesti hallinnoida jonkin näyttöpäätteen pikseleitä, eli esimerkiksi kokonaan tekstipohjaiset tai staattiset käyttöliittymät on rajattu tästä joukosta ulos. Esitellyt komponenttikirjastot ovat aihesanahakujen perusteella löytämiäni jollain tapaa merkittäviä kirjastoja. Merkittäväksi olen katsonut kirjaston, jos se on ensimmäisenä tai ensimmäisten joukossa esitellyt uusia käsitteitä käyttöliittymän määrittämiseen, tai jos se on saavuttanut poikkeuksellista suosiota laajan käyttöasteen muodossa. Joitain huomionarvoisia komponenttikirjastoja siis ei välttämättä mainita tai käsitellä kovin tarkasti tässä työssä, mutta olen parhaani mukaan luetellut kaikki tämän työn kannalta kiinnostavia kehityksiä esitelleet komponenttikirjastot. Kirjastot esitellään kronologisessa järjestyksessä niin, että katsaus kuvaa komponenttikirjastojen kehittymistä mahdollisimman tarkasti. Tarkemmin aineistohakua on esitelty kappaleessa 2.3.

4.1 1970- ja 1980-luvut – inkrementaaliset komponenttikirjastot

Ensimmäinen julkista huomiota saanut pikselipohjainen GUI-ratkaisu oli Xeroxin PARC-tutkimuskeskuksessa kehitetty ja 1973 julkaistu Alto-tietokone (ks. Wadlow 1981; "Graphical user interface (GUI) | Britannica" 2023; "A History of the GUI | Ars Technica" 2023). PARC:ssä kehitettiin myös Smalltalk-ohjelmointikieli ja sen kehitysympäristö, joka oli ajettavissa Alto-koneella (ks. "Introducing the Smalltalk Zoo - CHM" 2023; "A History of the GUI | Ars Technica" 2023; "The Xerox Alto, Smalltalk, and rewriting a running GUI" 2023). Tämä kehitysympäristö sisälsi oman ohjelmoitavan GUI-ympäristönsä, johon sisältyi monia sittemmin suosituksi tulleita ominaisuuksia, kuten vapaasti asemoitavat ohjelmaikkunat (ks. "Introducing the Smalltalk Zoo - CHM" 2023; "A History of the GUI | Ars Technica" 2023; "The Xerox Alto, Smalltalk, and rewriting a running GUI" 2023). Myöhempi julkista huomiota saavuttanut ja jatkokehitystä tuottanut julkaisu on esimerkiksi Xerox Altosta

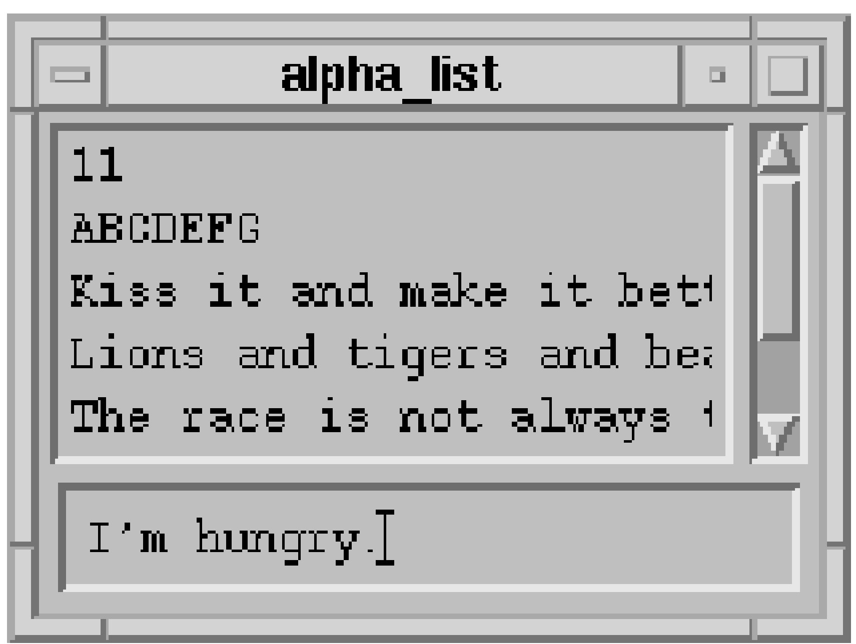
vaikutteita saanut 1983 julkaistu Apple Lisa -tietokone, jonka yhteydessä kehitettiin esimerkiksi tiedostojen siirtäminen raahaamalla (ks. "Graphical user interface (GUI) | Britannica" 2023; "A History of the GUI | Ars Technica" 2023). Lisa ei saavuttanut kaupallista menestystä, mutta toimi pohjana suositulle Apple Macintosh -tietokoneelle, ja sitä myötä Mac OS -käyttöjärjestelmälle, joka Macintoshissa tunnettiin nimellä System 1 (ks. "The Lisa: Apple's Most Influential Failure - CHM" 2023; "Macintosh System 1: What Was Apple's Mac OS 1.0 Like?" 2023). Vuonna 1984 julkaistiin Microsoft Windows 1.0 -käyttöjärjestelmä, joka sisälsi myös graafisen käyttöliittymän ja toimi edeltäjänä Microsoftin myöhemmille Windows-käyttöjärjestelmille (ks. "A History of the GUI | Ars Technica" 2023). Vuonna 1984 Massachusetts Institute of Technology -laitoksessa kehitettiin X Window System -niminen ikkunointijärjestelmä (ks. "A History of the GUI | Ars Technica" 2023), joka oli suunniteltu käyttöjärjestelmäriippumattomaksi (ks. "A History of the GUI | Ars Technica" 2023; Scheifler ja Gettys 1986). X on ikkunointijärjestelmä, eli se huolehtii siitä, että usean eri ohjelman käyttöliittymät voidaan jakaa itsenäisiin ikkunoihin näyttöpäätteellä, ja ikkunoita voidaan siirrellä ja niiden välillä voidaan siirtyä käyttäjän toimesta (Myers 1995; Scheifler ja Gettys 1986). X myös huolehtii ikkunoiden piirtämisestä näytölle (engl. *rendering*) sekä käyttäjäsyötteiden vastaanottamisesta tietokoneen laitteistolta ja niiden välittämisestä sovelluksille (Myers 1995; Scheifler ja Gettys 1986). X tarjoaa yksinkertaisia primitiivisiä piirtomahdollisuuksia kuten tekstin tai kuvioiden piirtämistä, mutta se ei varsinaisesti sisällä komponenttikirjastoa, jolla voitaisiin korkeamman tason määrityksillä piirtää suoraan esimerkiksi nappuloita tai syötekenttiä (Scheifler ja Gettys 1986). X on kuitenkin mielenkiintoinen lähtökohta GUI-kehityksen tarkasteluun siitä syystä, että se on vielä nykyäänkin käytössä monissa Unix-tyylisessä käyttöjärjestelmässä (ks. "A History of the GUI | Ars Technica" 2023; Carew ja Damodaran 2008), ja sen päälle on rakennettu sen historian aikana useita erilaisia komponenttikirjastoja. Toisaalta myös Mac OS ja sen myöhemmät versiot OS X ja macOS (ks. "Macintosh System 1: What Was Apple's Mac OS 1.0 Like?" 2023) ovat mielenkiintoisia tarkastelukohteita, sillä ne edustavat kaupallista käyttöliittymäkehitystä ja ovat kehittyneet aina 1980-luvulta tähän päivään asti. Käytännön vuoksi seuraavassa tarkastelussa keskitytään näihin kahteen ympäristöön sekä muihin kiinnostavia arkkitehtuurillisia ratkaisuja sisältäneisiin komponenttikirjastoihin.

1988 julkaistiin Motif, josta tuli X-ekosysteemin hallitseva komponenttikirjasto 1990-luvun

lopulle asti, kun se valittiin Unix-käyttöjärjestelmiin suunnitellun Common Desktop Environment -standardin osaksi (ks. “Motif (software) - Wikipedia” 2023; “Common Desktop Environment - Wikipedia” 2023). Motifin rajapinta on käytettävissä esimerkiksi C-ohjelmointikielellä, ja siitä on helposti tunnistettavissa imperatiivinen näkymäkomponenttien määrittely ja muokkaus sekä inkrementaalinen näkymän dynaaminen päivitys. Liitteen A listauksessa 4 on kirjasta Heller, Ferguson ja Brennan 1994 mukaillen kopioitu esimerkki Motif-käyttöliittymästä, jossa käyttäjä voi lisätä listakomponenttiin uusia tekstikomponentteja. Kuviossa 3 esitetään käyttöliittymä piirrettynä. Listauksessa komponentti-instansseja luodaan Motif-kehyksen `XtVaCreateWidget`-aliohjelmalla, joka palauttaa viitteen luotuun komponenttiin. Tälle käskylle voidaan syöttää kolmantena argumenttina yläkomponentti, jonka sisälle luotava uusi komponentti asemoidaan. Alkunäkymä siis muodostetaan peräkkäisillä käskyillä imperatiivisesti. Kun taas reagoidaan käyttäjän syötteeseen ja halutaan päivittää näkymää, tehdään se Motifin tarjoamilla aliohjelmilla, joilla voidaan hakea näkymän komponenttien tiloja tai päivittää niitä. Aliohjelma `XtVaGetValues` hakee listakomponentilta sille asetettuja parametreja, tässä tapauksessa listan alkioiden määrän. `XmListAddItemUnselected` puolestaan muokkaa olemassa olevaa listakomponenttia ja lisää siihen uuden alkion, jolloin komponenttikirjasto huolehtii käyttöliittymän päivittymisestä näytöllä. Näkymän päivitys siis tapahtuu inkrementaalisesti muokkaamalla näkymää sen edeltävää hetkellistä tilaa käyttäen, ja muokkaukset on määritetty imperatiivisesti peräkkäisillä käskyllä.

Motifin rinnalla toinen merkittävä varhainen X-pohjainen komponenttikirjasto oli Sun Microsystemsin kehittämä XView (ks. “XView - Wikipedia” 2023). XView edusti Open Look -nimistä Unix-GUI-määrittelyä, ja sen lähdekoodi julkaistiin 1990-luvulla (ks. “XView - Wikipedia” 2023). XView:n kehitys lopetettiin kuitenkin lopulta, kun Motif valittiin Common Desktop Environment -ympäristön osaksi. XView:n rajapinnasta on Motifin tavoin nähtävissä, että se noudatti imperatiivista ohjelmointityyliä komponenttien näkymään lisäämiseen ja muokkaamiseen, ja näkymän päivittämiseen käytettiin inkrementaalista arkkitehtuuria (ks. “XView, an OPEN LOOK Toolkit for X Window download | SourceForge.net” 2023; “XView & OpenLook” 2023).

Käyttöjärjestelmäriippuvaisten komponenttikirjastojen puolella merkittävä tekijä 1980-luvun loppupuolella puolestaan oli Next-yhtiön kehittämä 1989 julkaistu NeXTSTEP (ks. “A Brief



Kuvio 3: Motif-komponenttikirjastolla toteutettu listanäkymä

History of Mac OS X" 2023), jota Apple myöhemmin hyödynsi Mac OS X -käyttöjärjestelmässä (ks. Linzmayer 1999). NeXTSTEP-kehityksessä saatavilla oli graafinen työkalu käyttöliittymien rakentamiseen, mutta varsinainen näkymän ohjelmakoodi oli edellä mainittujen komponenttikirjastojen tavoin imperatiivista ja näkymän päivittäminen tehtiin inkrementaalisesti (ks. "CalculatorApp" 2023). Myös Applen 1990-luvun lopulla julkaisema Cocoa-rajapinta, joka perustui NeXTSTEP-kirjastoon (ks. Linzmayer 1999), toimi samoilla periaatteilla.

1990-luvulla julkaistiin monia käyttöjärjestelmäriippumattomia komponenttikirjastoja, jotka ovat edelleen suhteellisen suosittuja – huomattavia esimerkkejä näistä ovat Qt, GTK, wxWidgets ja Tk. Käytän näitä kirjastoja esimerkkeinä, sillä Qt ja GTK ovat tälläkin hetkellä käytössä monissa ohjelmistoprojekteissa, varsinkin avoimen lähdekoodin ohjelmistoissa. QT on käytössä erityisesti KDE-yhteisön projekteissa ja GTK puolestaan GNOME-yhteisön projekteissa. wxWidgets ja Tk eivät nykyään ole yhtä laajalti käytettyjä kuin kaksi aiempaa, mutta niitäkin käytetään eri projekteissa, ja niille on saatavilla internetissä ja kirjallisuudessa paljon harjoittelumateriaalia. Nämä kaikki kirjastot toimivat julkaisuhetkellään ja myös kirjoitushetkellä varsin samanlaisilla arkkitehtuuriperiaatteilla – käyttöliittymän alkunäky-

mä voidaan määrittää imperatiivisesti komponentti-instansseja manipuloimalla, ja käyttöliittymän muutokset määritetään inkrementaalisesti näkymän aiemman version pohjalta (ks. “Calculator Form | Qt Designer Manual” 2023; “Gtk – 4.0: Getting Started with GTK” 2023; “wxWidgets: Samples Overview” 2023; “Tk examples” 2023). Toisaalta näissä kirjastoissa on kuitenkin havaittavissa poikkeamia tähän malliin. Esimerkiksi Tk-kirjastossa tekstikenttä-komponenttien sisältötekstit määräytyvät reaktiivisesti niin, että tekstikentän sisältö on sidottu johonkin muuttujaan. Muuttujan arvo päivittyy käyttäjän muokatessa kentän tekstiä, ja toisaalta muuttujan arvon muuttaminen päivittää käyttöliittymässä näkyvän tekstin vastaavaksi. Tästä on esimerkkinä yksinkertainen yksikkömuunninohjelma, jossa `textvariable`-direktiivillä määritetään tekstinsyöttökenttään tai tekstiotsikkoon sidottu muuttuja (ks. “TkDocs Tutorial - A First (Real) Example” 2023). Näkymää ei siis tarvitse tällöin muokata vastaamaan johonkin muuttujaan säilöttyä tilaa erillisellä käskyllä, vaikka näkymän muuttaminen edelleen tapahtuukin inkrementaalisesti edellisen näkymätilan pohjalta. Tämän voidaan katsoa helpottavan käyttöliittymän sisäisen tilan ja sen ulkoisen tilan synkronointia. Toisaalta Tk:ssa on myös `config`-metodi, jolla komponentti-instanssien ominaisuuksia voidaan muokata inkrementaalisesti (ks. “TkDocs - ttk.Entry” 2023). Toinen kiinnostava havainto edellä mainituista kirjastoista on Qt:n tarjoama QML-määrittelykieli, joka tarjoaa deklaratiivisen tavan näkymien määrittämiseen. Myös näkymän muutokset määritellään siinä deklaratiivisesti listaamalla näkymän erilaisia tiloja, mutta näkymän päivittäminen kuitenkin on inkrementaalista ja esitetään osittaisina muutoksina näkymän edelliseen tilaan (ks. “Qt Quick States | Qt Quick 6.6.0” 2023).

Näiden esimerkkien pohjalta voidaan havaita, että että 1980-luvun ja 1990-luvun alun GUI-komponenttikirjastot pääsääntöisesti toimivat imperatiivista ohjelmointityyliä ja inkrementaalista arkkitehtuuria käyttäen.

4.2 1990-luku – funktionaaliset komponenttikirjastot

1990-luvulla kehitettiin myös joitain GUI-ratkaisuja, jotka poikkesivat aiemmin esitetystä imperatiivisen ja inkrementaalisen näkymämäärittelyn yhdistelmästä. Luultavasti laajimmalle levinnyt tällainen ratkaisu ovat olleet HTML- ja JavaScript-kielet. HTML on web-sivujen esittämiseen tarkoitettu merkintäkieli, mutta sellaisenaan se pystyy esittämään vain staatti-

sia käyttöliittymiä sekä käyttöliittymiä, joissa voidaan siirtyä näkymien välillä hyperlinkkejä aktivoimalla, jolloin web-selain lataa näytettäväksi linkin osoittaman uuden web-sivun. HTML:stä tekee mielenkiintoisen se, että sen rinnalle kehitettiin varsin varhain JavaScript-ohjelmointikieli (ks. “Netscape and Sun announce JavaScript, the Open, Cross-platform Object Scripting Language for Enterprise Networks and the Internet” 2023), jonka tarkoitus oli toimia web-selaimissa ja mahdollistaa HTML-sivujen dynaaminen muuttaminen ajan tai käyttäjäsyötteiden mukana ilman navigointia toiselle web-sivulle. Myöhemmin julkaistu DOM-rajapinta määritti web-selaimille ja muille sovelluksille standardisoidut operaatiot, joiden kautta JavaScript-koodilla tai muulla ohjelmointikielellä voidaan manipuloida HTML:llä määritettyjä käyttöliittymiä (ks. “Introduction to the DOM - Web APIs | MDN” 2023). Tämän myötä HTML ja JavaScript yhdessä muodostavat kokonaisuuden, jossa kukin web-sivu edustaa eräänlaista itsenäistä käyttöliittymän näkymää, ja JavaScript-koodilla voidaan määrittää näkymän dynaaminen käyttäytyminen. Tällöin näkymän alkutila on määritetty deklaratiivisesti HTML-kielellä, ja toisaalta näkymän muutokset määritetään inkrementaalisesti muokkaamalla näkymää DOM-rajapinnan imperatiivisilla käskyillä (ks. “Introduction to the DOM - Web APIs | MDN” 2023).

Toisaalta kenties kiinnostavampia semanttisia muutoksia esittelivät 1990-luvulla ja 2000-luvun alussa jotkin funktionaalisille ohjelmointikielille kehitetyt komponenttikirjastot ja GUI-ratkaisut. Tällaisia olivat esimerkiksi Fudgets, Haggis, TkGofer, Fran, FranTk, Concurrent Clean -ohjelmointikielen standardikirjasto ja Fruit (ks. Carlsson ja Hallgren 1993; Finne ja Jones 1995; Vullinghs, Tuijnman ja Schulte 1995; Elliott ja Hudak 1997; Sage 2000; Noble ja Runciman 1995; Courtney ja Elliott 2001). Näistä kenties kiinnostavin on Fudgets.

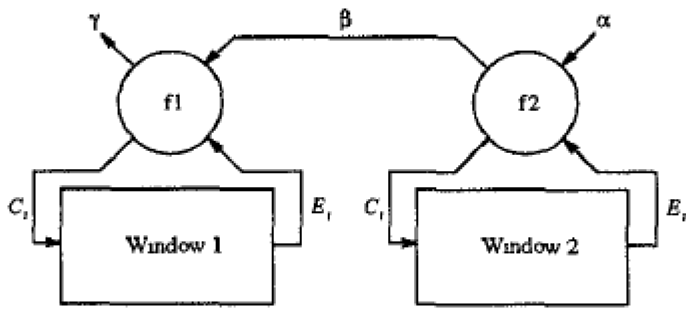
4.2.1 Fudgets

Fudgets esiteltiin 1993, ja siitä on saatavilla Xlib-kirjaston päälle rakennettu toteutus (Carlsson ja Hallgren 1993). Fudgetsista kiinnostavan tekee se, että se on uskoakseni ensimmäinen tunnettu GUI-kirjasto, joka ei määritä käyttöliittymän dynaamisia muutoksia inkrementaalisten muutosten kautta, eli se tämän tutkimuksen termein hyödyntää näkymän suoraa määrittelyä. Tähän Fruit-kirjaston suunnittelijat Courtney ja Hudak viittaavat kutsumalla Fudgetsia ensimmäiseksi ”aidosti funktionaaliseksi” GUI-kirjastoksi (Courtney ja Hudak 2004). Fud-

gets perustuu *virtoihin* (engl. *stream*) diskreettejä tapahtumia ja komentoja, joita käsitellään ja yhdistellään kirjaston tarjoamilla toiminnoilla niin, että voidaan määrittää dynaamisesti päivittyviä käyttöliittymiä. Fudgetsin tarjoamia käyttöliittymäkomponentteja kutsutaan *fudgeteiksi* (engl. *functional widget*), ja ne toimivat konkreettisten näkymän osien lisäksi funktioina, jotka vastaanottavat arvoja syötevirrastaan ja palauttavat ulostulovirtaansa vastaavasti jonkin arvon (Carlsson ja Hallgren 1993). Esimerkki fudgetista on `textF`, joka vastaa tekstinsyöttökenttäkomponenttia (Carlsson ja Hallgren 1993). `textF`-arvon esittäminen saa Fudgets-kirjaston piirtämään näytölle interaktiivisen tekstikentän. `textF` voi vastaanottaa virrasta syötteenä tekstijonon, joka sen sisällöksi pitäisi asettaa, tai se voi palauttaa virtaan tekstijonon, joka vastaa käyttäjän siihen kirjoittamaa uutta tekstiä. Carlsson ja Hallgren ehdottavat tämän perusteella, että käyttöliittymäkomponentin ja siten fudgetien tyypin tulisi olla $F\alpha\beta$, missä F on fudgetin tyyppi ja α ja β ovat tyypin parametrit syötevirran arvoille ja ulostulovirran arvoille (Carlsson ja Hallgren 1993). Tällöin `textF`-fudgetin tyyppi olisi $FStringString$, koska sen syötearvot ovat merkkijonoja samoin kuin sen palauttamat käyttäjän tekstikenttään kirjoittamat arvot.

Koska käytännön käyttöliittymät koostuvat useammasta kuin yhdestä komponentista, Fudgets esittelee joitain perustason kombinaattoreita ja rakenteita, joilla fudgeteista voidaan muodostaa toimiva dynaaminen käyttöliittymä. Eräs keskeinen näistä kombinaattoreista on $>==<$, jonka tyyppi on $F\beta\gamma \rightarrow F\alpha\beta \rightarrow F\alpha\gamma$, mikä muistuttaa funktiokompositiota (Carlsson ja Hallgren 1993). Tämä kombinaattori ottaa vastaan kaksi fudgetia, jolloin molemmat piirretään näytölle, ja lisäksi se ohjaa oikeammaisen fudgetin ulostulovirran arvot vasemmaisen fudgetin sisääntulovirran arvoiksi. Syöte- ja ulostulovirtojen kulku fudgettien välillä on tällöin kuvion 4 mukainen.

Esimerkki Fudgets-käyttöliittymästä, joka koostuu useasta komponentista, on kaksi tekstikenttää, joista toisen ulostulovirta on kytketty toisen syötevirtaan (Carlsson ja Hallgren 1993). Tällöin käyttäjän ensimmäiseen tekstikenttään kirjoittamat arvot päätyvät toisen tekstikentän syötevirtaan, jolloin molemmat tekstikentät päätyvät näyttämään samaa tekstiä. Näin paitsi koostetaan useampi komponentti samaan näkymään, myös välitetään tietoa komponenttien välillä. Käyttöliittymän määritelmä Fudgets-kirjaston uusimmalla versiolla on seuraavanlainen:

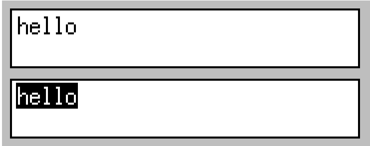


Kuvio 4: Syöte- ja ulostulovirtojen kulku komposiitti-fudgetin *f1* >==< *f2* osien välillä (Carlsson ja Hallgren 1993). C ja E ovat matalan tason virtoja, joita komponenttikirjaston käyttäjä ei suoraan käytä.

```
module Main (main) where
import Fudgets

main = fudlogue (shellF "" (stringInputF >==< stringInputF))
```

Kuviossa 5 nähdään, miltä käyttöliittymä näyttää, kun ohjelmakoodi suoritetaan. Teksti ”hello” kirjoitettiin vain alempaan tekstikomponenttiin, jolloin enter-painalluksen jälkeen ylempi tekstikenttä päivittyi sisältämään saman tekstin. Komponentit asemoidaan oletuksena allekkain määrittelyjärjestyksessä.



Kuvio 5: Komposiittikäyttöliittymä piirrettynä Fudgets-kirjastolla

Fudgets-kirjasto sisältää joitain muitakin peruskombinaattoreita, jotka näkymän määrittämisessä ovat tarpeellisia, kuten esimerkiksi asemointi-fudgeteja, joilla voidaan hallita esimerkiksi sitä, mihin suuntaan fudgetit toisiinsa nähden sijoitetaan, kun ne piirretään näytölle. Myös fudgetien virtojen yhdistämiseen ja silmukkaan kytkentään tarjotaan kombinaattorit. Fudget-käyttöliittymän rakenteen dynaaminen muokkaus on puolestaan huomioitu listakombinaattorilla, jolle voidaan syötevirran arvoina syöttää uusia fudgeteja esitettäväksi. Lisäk-

```
module Main(main) where
import Fudgets

main = fudlogue (shellF "Counter" counter_f)

startstate = 0

counter_f = intDispF startstate >==< absF counter >==< buttonF "Inc"

counter = count startstate
  where count n = getSP $ \ _ ->
    putSP [n+1] $
    count (n+1)
```

Listaus 4.1: Laskurinäkymä Fudgets-kirjastolla (Haskell) (Carlsson ja Hallgren 1993)

si Carlsson ja Hallgren toteavat, että käyttöliittymän pitää voida käsitellä myös tilaa, joka riippuu aiemmista syötevirtojen tapahtumista (Carlsson ja Hallgren 1993). Tätä varten Fudgetsiin on lisätty tuki myös abstrakteille virtafunktioille (SP, engl. *stream processor*), jotka muistuttavat tyypiltään fudgeteja, mutta niitä ei piirretä missään vaiheessa näytölle. Virtafunktion tyyppi on $SP\ \alpha\ \beta$, ja niistä voidaan suoraan muodostaa fudgetien kanssa yhdistettävissä oleva fudget `absF`-funktiolla, jonka tyyppi on $SP\ \alpha\ \beta \rightarrow F\ \alpha\ \beta$. Fudgets tarjoaa `getSP` ja `putSP` -funktiot, joiden tyypit ovat:

```
getSP :: (a -> SP a b) -> SP a b
putSP :: [a] -> SP b a -> SP b a
```

Ne voivat jatkonvälityksen (engl. *continuation passing*) avulla saada syötevirrasta aiemman arvon käyttöönsä, tai voivat syöttää seuraaville virtafunktioille uusia arvoja. Näin voidaan puhtaasti ilman sivuvaikutuksia kuljettaa fudget-näkymässä mitä tahansa näkymän ulkopuoleista tilaa. Tästä nähdään esimerkki listauksessa 4.1.

Esimerkissä rekursiivinen `count`-funktio määrittää rakenteen, jossa virtafunktio reagoi aina

mihin vain sille annettuun arvoon ja syöttää sen jälkeen virtaan aiempaa yhtä suuremman arvon. Virtafunktion syötteet tulevat siihen kytketyltä nappi-fudgetilta, jolloin se syöttää ulostulovirtaansa kasvaneen arvon, jonka Fudgets-kirjasto sitten välittää lukuarvoa näyttävälle komponentille syötteeksi. Abstrakteja virtafunktioita voidaan jatkonvälityksen kautta käyttää myös näkymään liittymättömien sivuvaikutusten suorittamiseen deterministisessä järjestyksessä (Overton 1997). Näin voidaan toteuttaa esimerkiksi tietokantakyselyjen suorittaminen halutussa järjestyksessä GUI-ohjelmakoodin sisältä.

Fudgets on hyvin mielenkiintoinen esimerkki, sillä ensisilmäyksellä sen toimintamalli voi vaikuttaa hyvin erilaiselta kuin tässä tutkimuksessa kuvatut esimerkit näkymän määrittämisestä. Siitä on kuitenkin havaittavissa käsitteet, jotka luvussa 3.2 esiteltiin. Jos katsotaan esimerkiksi listausta 4.1, nähdään, että Fudgets-näkymä määritetään deklaratiivisesti komponentteja yhdistelemällä. Vastaavasti kuin deklaratiivisessa HTML:ssä, Fudgets-komponentteja voidaan asemoida rinnakkain (esimerkiksi >==<-operaattorilla) tai sisäkkäin (esimerkiksi `shellF`-funktio ottaa parametrinaan sisemmän fudgetin), joskin Fudgets-koodin syntaksi on hieman erilainen esimerkiksi siksi, että rinnakkaisten komponenttien esittämiseen tarvitaan erilaisia operaattoreita, kun taas HTML-koodissa komponenttien peräkkäin kirjoittaminen tekee niistä rinnakkaisia, ja toisen komponentin sisälle tulevat komponentit määritetään tapauskohtaisesti funktion argumentteina, kun taas HTML-koodissa sisäkkäiset komponentit tulevat aina ylemmän komponentin alku- ja loppuavainsanojen sisälle. Fudgets poikkeaa HTML:stä merkittävimmin siinä, että näkymäkoodi ei kuvaa ainoastaan tietyn hetkellisen näkymän hierarkkista komponenttipuuta, vaan Fudgetsin kombinaattoreilla ja virtafunktioilla voidaan määrittää myös komponenttien välistä kommunikaatiota, virtaan tulevien tapahtumien käsittelyä sekä komponenttien dynaamista mukautumista tapahtumien myötä. Tätäkin voidaan tarkastella listauksen 4.1 kautta. Siinä `counter_f` määrittää fudget-rakenteen, jossa numeronäyttö ja nappi on asetettu allekkain näkymään. Nappi-fudgetin syötevirta kytkeytyy Fudgets-kirjaston kautta käyttäjän syötteisiin niin, että jokainen piirretyn napin aktivointi lisää syötevirtaan arvon. Monimutkaisempi käyttöliittymä voi sisältää useita tällaisia fudgeteja, joiden syötevirta on kytketty käyttäjän syötteisiin. Lisäksi käyttöliittymässä voi olla virtafunktioita, joiden syötteet vastaavasti tulevat näkymärakenteen ulkoisista lähteistä, kuten ajastimista tai järjestelmätason tapahtumista. Tapahtumien syötevirtoja voidaan kombinaattorien avulla välittää komponenttien välillä, ja virtafunktioilla, kuten `counter`-funktiolla,

voidaan käsitellä virtoja ja säilöä tilaa syötteiden välillä. Fudgetsin näkymämääritelmä siis on näin ollen täysi kuvaus siitä, mitä tapahtumia käyttöliittymä ottaa vastaan, ja miten käyttöliittymä kehittyy ja mihin tilaan se päätyy tietyn tapahtumasarjan jälkeen. Näin ollen nähdään, että Fudgets-määritelmä siis muistuttaa kaavaa 3.1 – fudgetien muodostama virtojen kompositio on eräänlainen funktio, joka muuntaa sille annetun listan tapahtumia (syötevirtojen arvot) konkreettiseksi hetkelliseksi näkymäksi.

Fudgets-määritelmä ei kuitenkaan vastaa suoraan kaavaa 3.1, sillä abstraktit virtafunktiot voivat jatkonvälityksen avulla säilöä tilaa, joka perustuu aiempiin tapahtumiin pelkän viimeisimmän tapahtuman sijasta. Tällöin Fudgets-näkymän määritelmä virtafunktion osalta muistuttaa enemmän kaavan 3.3 ylempää riviä. Virtafunktiot voivat muodostaa Fudgets-näkymään eräänlaisia lokeroita, jotka säilövät tilaa ja palauttavat sen perusteella arvoja ulostulovirtaan muiden fudgetien käytettäväksi, kun taas muut näkymän komponentit riippuvat suoraan niille annetuista syötevirtojen arvoista ilman arvojen välistä tilan käsittelyä.

Koska Fudgets määrittää näkymän dynaamisen käyttäytymisen osin koko tapahtumahistorian funktiona (kaava 3.1) ja osin erillisten näkymärakenteeseen säilöttyjen tilojen funktiona (kaava 3.3), voidaan sen katsoa noudattavan suoraa näkymämäärittelyä – Fudgets-näkymää dynaamisesti päivitettäessä ei ikinä viitata näkymän komponentti-instanssien aiempaan sisäiseen tilaan ja muokata sitä, vaan näkymä muodostetaan aina viimeisimmän syötearvon ja mahdollisen virtafunktioihin säilötyn tilan perusteella. Näkymän komponentti-instanssit voivat myös säilöä sisäistä tilaa itseensä, esimerkiksi tekstikenttä voi pitää sisäisesti tietoa nykyisestä tekstistään, mutta toisin kuin inkrementaalisessa näkymämäärittelyssä, näkymän komponenttien tilaa ei ikinä hyödynnetä näkymän seuraavan tilan muodostamisessa. Fudgets oli tiettävästi ensimmäinen konkreettinen GUI-kirjasto, joka pohjautuu täysin suoraan näkymämäärittelyyn.

Fudgets ei nykyään ole kovin laajassa käytössä. Syitä tähän on haastava selvittää varmuudella, mutta niitä voidaan pohtia käyttöliittymämäärittelyn käsitteiden pohjalta. Eräs ominaisuus, joka Fudgetsin toimintamallista seuraa, on, että näkymän syntaksipuu määrittää sekä komponenttien fyysisen sijainnin näytöllä että komponenttien tapahtumavirtojen lähekkäisyyden. Vaikka asemointikomponenteilla voidaan muuttaa esimerkiksi komponenttien etäisyyksiä toisistaan, rinnakkain syntaksipuussa olevat komponentit ovat silti saman yläkompo-

nentin sisällä. Näkymän haluttu ulkoasu siis jossain määrin määrittää sitä, millainen fudget-syntaksipuun pitää olla. Tämä ei kuitenkaan välttämättä ole optimaalista fudgetien väliseen tapahtumavirtojen käsittelyyn. Toistensa tapahtumavirroista kiinnostuneiden fudgetien olisi käytännöllistä sijaita syntaksipuussa mahdollisimman lähellä toisiaan, koska muuten virtojen kytkeminen haluttujen fudgetien välille vaatii virtojen kuljettamista toisten fudgetien läpi, mikä voi vaatia eri tapahtumavirtojen yhdistämistä ja myöhemmin eriyttämistä, sillä kullakin fudgetilla on vain yksi syöte- ja ulostulovirta. Lisäksi tapahtumavirtoja saatetaan joutua välittämään pitkien etäisyyksien päähän syntaksipuussa mikäli toistensa tapahtumista kiinnostuneet fudgetit sijaitsevat näkymässä kaukana toisistaan. Tällaiset monimutkaiset ja pitkät virtakulut saattavat olla ohjelmoijalle haastavia kirjoittaa tai hahmottaa. Lisäksi näkymän visuaalisen rakenteen muuttaminen vaatii myös fudgetien välisten virtojen käsittelyn muuttamista. Näkymän visuaalinen komponenttipuu ja komponenttien väliset datan riippuvuudet eivät useinkaan vastaa toisiaan suoraan (Hanus ja Kluß 2009). Voidaan siis väittää, että tapahtumavirtojen ja näkymän visuaalisen rakenteen kytkeminen toisiinsa on Fudgets-kirjaston käyttöä hankaloittava tekijä. Jos Fudgetsin mallia verrataan perinteisempiin imperatiivisiin ja inkrementaalisiin komponenttikirjastoihin, voidaan havaita, että kun käyttöliittymä niissä reagoi tapahtumiin, käsittelijäfunktiot voivat tämän seurauksena muokata dynaamisesti mitä tahansa käyttöliittymän komponentteja, joihon komponenttikirjastosta pystytään viittaamaan – komponenttien etäisyydellä esimerkiksi tapahtuman lähdekomponenttiin ei ole väliä. Tästä nähdään esimerkki listauksessa 4, jossa `add_item`-takaisinkutsufunktiolle välitetään `list_w`-muuttujaa, joka on viite näkymän listakomponenttiin, jota funktio kutsuttaessa muokkaa. Periaatteessa mihin vain tapahtumaan kytketystä takaisinkutsufunktiosta voidaan muokata kyseistä listakomponenttia, kunhan listakomponentin viite on takaisinkutsufunktiota määritettäessä saatavilla. Fudgetsin tapauksessa voitaisiin miettiä, olisiko komponenttien syötevirtojen käsittely ja näkymän komponenttipuun määrittäminen hyödyllistä erottaa toisistaan – käsiteltävät syöte- ja ulostulovirrat voitaisiin tällöin kohdistaa näkymän komponenttipuun komponentteihin esimerkiksi antamalla kullekin näkymän komponentille uniikki nimi.

Toisaalta Fudgetsin mallissa on etuja, joita muut komponenttikirjastot ennen sitä eivät vielä tarjonneet. Virtojen kytkentä komponentista toiseen voi tuottaa pienellä koodimäärällä ja suoraviivaisella semantiikalla verrattain monimutkaisiakin näkymiä. Käyttöliittymän dy-

naaminen toiminta, kuten Haskell-isäntäkielen johdosta myös muukin koodi, puolestaan on määritetty täysin deklaratiivisesti, mikä helpottaa käyttöliittymäkoodin staattista analysointia (Courtney ja Hudak 2004). Kenties osa syistä Fudgetsin verrattain pieneen käyttöasteeseen ei liity myöskään sen teknisiin ansioihin tai puutteisiin, vaan esimerkiksi siihen, ettei se ole saavuttanut yhtä suurta tunnettuutta kuin jotkin vaihtoehtoiset ratkaisut.

4.2.2 Fruit

Toinen kiinnostava funktionaalinen komponenttikirjasto on 2000-luvun alussa esitelty Fruit. Se perustuu hieman Fudgetsin tapaisesti malliin, jossa näkymä mallinnetaan reaktiivisesti aikapohjaisten arvojen kompositiona. Fudgetsissa näinä arvoina toimivat diskreetit virrat, joihin tiettyinä hetkinä voi esimerkiksi käyttäjän interaktion seurauksena syntyä hetkellinen arvo, joka käsitellään virtafunktioiden ja fudgetien kautta niin, että käyttöliittymä mahdollisesti päivittyy. Fruitissa sen sijaan aikapohjaisina arvoina toimivat *signaalit*, jotka ovat jatkuvia arvoja, joilla on jatkuvasti jokin hetkellinen arvo (Courtney ja Elliott 2001). Fruit esittää signaalit funktiona ajasta arvon: Signal $\alpha = Time \rightarrow \alpha$. Signaaleja ei käsitellä suoraan arvoina isäntäkielessä Haskellissa, vaan signaalien arvoja pystytään muokkaamaan signaalifunktioilla (engl. *signal transformer*), joiden tyyppi on: ST α β = Signal $\alpha \rightarrow$ Signal β. Lisäksi Fruit käyttää *nuolia* (engl. *arrows*) signaalifunktioiden kompositioiden määrittämiseen (Courtney ja Elliott 2001; John Hughes 2000). Nuolten avulla voidaan mallintaa signaalifunktioita abstrakteina laskentaoperaatioina tyypistä a tyyppiin b, ja näitä laskentaoperaatioita voidaan kytkeä toisiinsa »>-operaattorilla. »> muistuttaa funktiokompositiota niin, että se ottaa kaksi signaalifunktiota ja palauttaa uuden signaalifunktion, joka vie syötteensä ensimmäiselle signaalifunktiolle, syöttää tuloksen toiselle signaalifunktiolle, ja palauttaa toisen signaalifunktion tuloksen. Nuolten tyyppiluokka Haskell-kielessä on seuraavanlainen:

```
class Arrow a where
  arr :: (b –> c) –> a b c
  (>>>) :: a b c –> a c d –> a b d
  first :: a b c –> a (b, d) (c, d)
```

Signaalien, signaalifunktioiden ja nuolien ansiosta Fruit:lla voidaan esittää deklaratiivisina rakenteina laskutoimituksia, jotka riippuvat reaktiivisesti syötesignaalien arvoista kunakin

ajan hetkenä, ja Fruit-kirjasto voi piirtää laskutoimituksen tulokset kunakin hetkenä näytölle. Esimerkki Fruit-ohjelmasta nähdään listauksessa 4.2 (mukailtu tutkimuksesta Courtney ja Elliott 2001). Kyseinen ohjelma piirtää suoritettaessa näytölle ympyrän, joka seuraa hiiren osoitinta. Ohjelma koostuu signaalifunktioiden koostetusta rakenteesta, jossa lopullinen koostettu signaalifunktio vastaanottaa `GUIInput`-tyyppisiä arvoja, jotka kuvastavat kaikkia ohjelman syötteitä, kuten näppäimenpainalluksia tai hiiren osoittimen liikkumista, ja palauttaa `Picture`-tyyppisiä arvoja, joiden perusteella käyttöliittymä piirretään kunakin hetkenä. Ohjelmassa `mouseST`-signaalifunktion avulla voidaan purkaa ohjelman kaikista syötteistä hiiren sijainti, ja tämä arvo voidaan nuolen avulla välittää toiselle signaalifunktiolle, joka palauttaa annettuun sijaintiin piirretyn ympyrän. Ohjelma muodostuu näiden funktioiden koosteesta.

Fruit mahdollistaa myös näkymäkomponenteista erillään määritettävän tilan käsittelyn ja säilömisen ohjelman suorituksen myötä. Tämä mahdollistetaan rekursiivisilla muuttujilla – muuttujan alkuarvon määrittämisen jälkeen muuttujan myöhemmät arvot voivat aina riippua niitä edeltävästä arvosta (Courtney ja Hudak 2004). Tästä esimerkki nähdään listauksessa 4.3. Tässä `rpball0`-funktio määrittää varsinaisen ohjelman, ja `pbgame` on peliohjelma, joka käynnistyy uudelleen, jos se saa syötesignaalinaan tyhjän yksikkötyyppisen arvon (engl. *unit type*). `aboveGUI`-operaattori yhdistää napin ja peliohjelman toistensa päälle, ja muodostaa uuden signaalifunktion, jonka syötearvojen vasen puoli (esimerkissä `btext`-kohta) ja oikea puoli (esimerkissä `pressES`) välitetään vastaavasti vasemmalle ja oikealle komponentille, ja samoin tehdään ulostuloarvoille. Nuolet muotoa `pat <- st -< exp` ovat syntaktinen Haskell-kielen laajennos, jossa lauseke `exp` välitetään Fruit:n tapauksessa signaalifunktiolle, ja signaalifunktion palauttama arvo sidotaan muuttujaan `pat` (Courtney ja Elliott 2001). Koko lauseke muodostaa näin uuden signaalifunktion, joka vastaa lauseketta `arr (_ -> exp) >>> st >>> arr (\pat -> pat)`. Syntaktinen laajennos helpottaa useampien tällaisten nuolten koostamista, sillä kukin tällainen signaalifunktio voidaan kirjoittaa omalle rivilleen, ja sillä on aiempien rivien muuttujat käytettävissään. `rec`-avainsanalla voidaan määrittää rekursiivinen lauseke, jossa jokin arvo voi esiintyä nuolen vasemmalla puolella jo ennen sen määritystä. Esimerkissä näin tekee `pressES`-muuttuja, joka luetaan `fbutton`-signaalifunktion ulostuloarvosta. `pressES` esiintyy kuitenkin ennen tä-

```
-- Signaalifunktio, joka ottaa ohjelman kaikki hetkelliset syötteet eri lähteistä
-- ja palauttaa hiiren osoittimen koordinaatit.
mouseST :: ST GUIInput Point

-- Käyttöliittymät määritetään Fruit:ssa signaalifunktioina, jotka vastaanottavat syötteen
-- ja palauttavat käyttöliittymän hetkellisen kuvan. Tyypit a ja b ovat rinnakkaisia
-- arvoja, joita signaalifunktio samalla vastaanottaa ja palauttaa.
type GUI a b = ST (GUIInput, a) (Picture, b)

-- Grafiikkakirjaston tarjoama funktio, joka piirtää annetun kuvan annettuun pisteeseen
-- näytöllä.
move :: Picture -> Point -> Picture

ballPic :: Picture
ballPic = (circle `withColor` red)

-- Näkymämääritelmä, jossa ympyrä seuraa hiiren sijaintia. first-funktio nostaa
-- argumenttina saamansa signaalifunktion GUI-tyypin edellyttämään muotoon.
-- (move ballPic) on funktio tyyppiä (Point -> Picture). arr-kutsu muodostaa
-- siitä signaalifunktion niin, että se voi käsitellä aikapohjaisia signaaliarvoja.
ballGUI :: GUI () ()
ballGUI = first (mouseST >>> arr (move ballPic))

main = runGUI ballGUI
```

Listaus 4.2: Fruit-ohjelma, jossa ympyrä seuraa hiiren osoitinta (Haskell)

```
fbutton :: GUI ButtonConf (Maybe ())

pbgame :: Double -> GUI (Maybe ()) (Maybe ())

rpball0 :: Double -> GUI () ()
rpball0 vel = proc (inpS,_) -> do
  rec (picS, (pressES, _)) <-
    (fbutton `aboveGUI` (pbgame vel)) -<
      (inpS, (btext "Play Again!", pressES))
  returnA -< (picS, ())
```

Listaus 4.3: Fruit-ohjelma, jossa pallopeli käynnistetään uudelleen aina napin painamisen jälkeen (Haskell) (Sage 2000)

tä suoritettavassa kohdassa koodia, kun se välitetään `pbgame`:n palauttamalle signaalifunktiolle. Fruit huolehtii, että muuttujaa voidaan käyttää johdonmukaisesti alustetulla arvolla ennen sen ensimmäistä määrittelykertaa, tässä tapauksessa `Maybe`-tyypin `Nothing`, joka vastaa painamatonta nappia. Näin voidaan määrittää rekursiivinen muuttuja, joka voi riippua itsensä edellisestä arvosta ja siten säilöä tilaa sovelluksen suorituksen edetessä. Lisäksi tämä mekanismi mahdollistaa ulostuloarvojen lukemisen näkymän komponenteilta.

Fruit mahdollistaa lisäksi myös esimerkiksi dynaamisesti määräytyvät komponenttirakenteet (Courtney ja Elliott 2001). Toisaalta koska Fruit:n malli käyttöliittymästä on täysin puhdas, ei sen näkymäkoodiin voida määrittää sivuvaikutuksia aiheuttavia toimintoja ilman tapahtumien eksplisiittistä välittämistä ohjelman juureen IO-monadissa suoritettavaksi (Courtney ja Elliott 2001). Tällä puhtaudella on hyötyjä ohjelmien analysoinnissa, mutta käytännön kannalta se voi tehdä joidenkin ohjelmien kirjoittamisen työläämmäksi. Fruit:lla on kuitenkin toteutettu ei-triviaaleja esimerkkisovelluksia (Courtney ja Hudak 2004), joten sen toimintamallin voidaan uskoa olevan riittävä käyttöliittymien ohjelmointiin.

Edeltävistä esimerkeistä voidaan nähdä, että että Fudgetsin tapaan Fruit käyttää suoraa määrittelyä käyttöliittymien esittämiseen. Näkymä on koostetun signaalifunktion palauttama arvo kunakin ajan hetkenä. Näkymän määritelmässä voidaan viitata rekursiivisiin muuttujiin,

mutta näkymän komponenttien edeltäviin tiloihin sen sijaan ei ikinä viitata. Näkymä päivittyy reaktiivisesti, kun jokin sen määritelmässä käytetty signaali muuttaa arvoaan. Tämän toimintamallin voidaan nähdä vastaavan kaavaa 3.3, missä näkymän määritys reaktiivisten arvojen eli tilan perusteella vastaa toiminnallisesti näkymän määrittämistä tilan funktiona kaavan alemman rivin mukaisesti. Tilan säilöntä ja käsittely Fruit:ssa puolestaan tapahtuu rekursiivisilla muuttujilla, joiden arvot voidaan määrittää perustumaan muuttujan edellisen arvon lisäksi esimerkiksi joihinkin signaalien arvoihin. Tämän voidaan katsoa toiminnallisesti vastaavan kaavan ylempää tilan funktiota, jossa tilaa päivitetään edellisen tilan ja uusimman tapahtuman perusteella iteratiivisesti.

Fruit-kirjastosta ei ole kirjoitushetkellä saatavilla aktiivisesti ylläpidettyjä versioita. Fruit ei siis ole saavuttanut laajaa suosiota. Syitä tähän on jälleen vaikea varmuudella määrittää. Fudgetsiin verrattuna Fruit:n käyttöliittymämääritelmät ovat siinä mielessä yksinkertaisempia, että niissä näkymän käyttämät aikapohjaiset arvot voidaan määritellä ohjelmakoodissa omaan osioonsa ennen komponenttien määrittämistä, jolloin useampi komponentti voi vaivatta riippua samoista arvoista, ja aikapohjaisten arvojen käsittely ja suhteet voidaan määrittää komponenttipuun ulkopuolella näkymän rakenteesta riippumattomasti. Tämä voidaan katsoa eduksi Fruit:lle. Toisaalta Fudgets mahdollistaa näkymän komponenttien rinnakkaisen suorituksen, kun taas Fruit-näkymät on nuolien käytön vuoksi suoritettava vaihe kerrallaan (Courtney ja Hudak 2004). Kenties merkittävin puute Fruit:n mallissa on, että se ei suoraan tarjoa tapaa suorittaa mielivaltaisia sivuvaikutuksia näkymän koodin kautta (Courtney ja Elliott 2001). Tämä tekee monet sovellukset mahdottomiksi toteuttaa Fruit:lla ilman muutoksia kirjaston määrittelyyn. Muita mahdollisia selityksiä Fruit:n matalalle käyttöasteelle voidaan taas etsiä varsinaisen grafiikkakirjaston ulkopuolelta – esimerkiksi huono tunnettuus, ylläpidetyn version puute tai kirjaston vaatimat pohjatiedot esimerkiksi nuolista.

4.2.3 Muut kirjastot

Kolmas tämän tutkimuksen kannalta kiinnostava 90-luvun GUI-komponenttikirjasto on Franiin perustuva FranTk. FranTk perustuu Fran-malliin, josta Fruit:kin on saanut inspiraatiota. Franissa dynaamisesti ajan myötä päivittyvistä arvoista käytetään nimitystä käytös (engl. *behavior*), ja ne vastaavat Fruit:n signaaleja (Courtney ja Hudak 2004). Fran ei kuitenkaan ole

suunnattu ensisijaisesti käyttöliittymien määrittämiseen, minkä vuoksi siitä puuttuu esimerkiksi kattava syötteiden käsittely, kuten syötteiden vastaanottaminen näkymän komponenteilta, sekä työkalut ohjelman sisäisen tilan hallintaan (Elliott ja Hudak 1997). FranTk on Franin päälle rakennettu GUI-komponenttikirjasto, joka esittelee käsitteitä, joita käyttölittymien määrittämisessä tarvitaan – komponentit, sivuvaikutukset monadien avulla kuvattuna, tilan säilömisen muokattavilla muuttujilla, komponenttien tapahtumiin reagoimisen kuuntelijoilla sekä komponenttipuun rakenteen määrittämisen dynaamisesti (Sage 2000). FranTk:lla näkymä määritetään monadisella käskyllä, jossa annetaan piirrettävä näkymän rakennepuu. Näkymä voi riippua käytöksistä, joiden muuttuminen ajan myötä saa myös näkymän päivittymään. Muuttujien muokkaaminen tehdään synkronisesti monadisten käskyjen kautta, ja muuttujat voidaan kytkeä käytöksinä osaksi näkymän määritelmää. Monadisesti määritettyjä käskyjä voidaan kuuntelijoiden avulla kytkeä suoritettavaksi halutuissa tilanteissa, kuten esimerkiksi nappulakomponentin painalluksen yhteydessä. FranTk muistuttaa siis paljon myös Fruit:n toimintaa, mutta yksi keskeinen ero on niiden välisessä komponenttien ulkoisen tilan käsittelyssä – Fruit:ssa tilaa ylläpidetään rekursiivisten muuttujamääritelmien avulla, kun taas FranTk:ssa tilaa kirjoitetaan ja luetaan erillisten käskyjen kautta. Toinen ero Fruit:iin verrattuna on, että FranTk:n kuuntelijat voivat Haskellin IO-monadin kautta suorittaa mielivaltaisia sivuvaikutuksia reaktiona tapahtumiin.

FranTk-ohjelmakoodista esimerkkinä voi toimia listaus 4.4. Siinä määritetään ohjelma `levEditor`, jossa näytölle piirretään tekstinsyöttökenttä ja liukusäädinsyöte, joiden arvot on kytketty toisiinsa. Monadinen `mkBVar`-käsky luo muokattavan muuttujan `bv`, ja `mkWindow`-käsky luo näytölle ikkunakomponentin, jonka sisältönä on toisena parametrina annettu komponenttirakenne. `nabove` ja `beside` ovat asemointikomponentteja, ja `entry` luo näkymän tekstinsyöttökentän, `scale` puolestaan liukusäätimen. Molemmille syötekentille on annettu parametriksi sama muuttuja, jota ne käyttävät syötteenä nykyiselle käyttöliittymässä esitettävälle arvolleen, mutta jota ne myös muokkaavat, kun käyttäjä syöttää uuden arvon komponentin kautta. Näin syötekomponenttien tilat pysyvät synkronoituina.

Vaikka FranTk käyttääkin imperatiivisia monadimuotoisia käskyjä tilanhallintaan, voidaan silti todeta, että sen toimintamalli perustuu näkymän suoraan määrittelyyn – näkymä määritetään kerran, eikä näkymärakenteen edelliseen tilaan viitata suorituksen aikana. Sen sijaan

```
title :: String -> Conf Window
nabove :: [Component] -> Component

levEditor :: String -> Int -> Listener Int -> WComponent
levEditor callsign init send = do
  bv <- mkBVar init
  mkWindow [title callsign]
    (nabove [beside (mkLabel [text "Lev"])
        (entry bv parse),
      scale bv,
      sendB bv send])
```

Listaus 4.4: FranTk-ohjelma, näytölle piirretään tekstinsyöttökenttä ja liukusäädinsyöte, joiden arvot on kytketty toisiinsa (Haskell) (Sage 2000)

näkymä päivittyy automaattisesti vastaamaan määritettyjä muuttujia ja käytöksiä, joiden arvot voivat muuttua ajan myötä. Tämä toimintamalli vastaa hyvin suoraan kaavaa 3.3, missä näkymän määritys reaktiivisten arvojen eli tilan perusteella vastaa toiminnallisesti näkymän määrittämistä tilan funktiona kaavan alemman rivin mukaisesti. Muuttujien päivitys kuuntelijoiden yhteydessä puolestaan vastaa kaavan ylempää tilan funktiota. Tilan päivitys tapahtuu imperatiivisesti, mutta muilta osin näkymän määritelmä on deklaratiivinen.

FranTk:sta julkaistiin käytännön toteutus ja joitain esimerkkisovelluksia, mutta sitä ei ole juuri kehitetty eteenpäin julkaisun jälkeen. Syitä suosion puuttumiselle voivat olla jälleen osin kirjaston ominaisuuksista riippumattomia, kuten esimerkiksi matala tunnettuus tai puuttuva ylläpidetty konkreettinen toteutus. Courtney ja Hudak esittävät lisäksi, että FranTk:n toteutuksessa voi esiintyä Franin muistivuoto-ongelmista periytyviä suorituskykyongelmia (Courtney ja Hudak 2004). Toisaalta FranTk:n komponenttipuusta erotettu imperatiivinen tilanhallinta ja tapahtumienkäsittely voidaan nähdä etuina verrattuna Fudgetsin malliin, ja FranTk kykenee mielivaltaisten sivuvaikutusten suorittamiseen näkymäkoodista toisin kuin Fruit:n perusmalli, joten FranTk:n mallilla on käytännössä omat vahvuutensa näkymien määrittämisessä.

Muita 1990-luvun ja 2000-luvun alun funktionaalisia komponenttikirjastoja, joita ei vielä ole esitelty tarkemmin, olivat Haggis, TkGofer ja Concurrent Clean -kielen standardikirjasto, joka sisälsi myös komponettikirjaston. Yhteistä näille kirjastoille on, että ne funktionaalisesta isäntäkielestä huolimatta mallintavat käyttöliittymien dynaamista päivittymistä inkrementaalisella tyylillä (Finne ja Jones 1995; Noble ja Runciman 1995; Carlsson ja Hallgren 1993; Courtney ja Elliott 2001). Esimerkkinä tästä voi toimia TkGofer, jossa käyttöliittymän alkunäkymä määritetään deklaratiivisesti, mutta tapahtumankäsittely esimerkiksi nappulaa painaessa tapahtuu monadisesti määritetyillä takaisinkutsufunktioilla (Vullinghs, Tuijnman ja Schulte 1995). Takaisinkutsufunktion sisällä tilan säilömiseen tapahtumien välillä käytetään `readState`- ja `writeState`-funktioita, ja näkymän muokkaus tapahtuu viittaamalla alkunäkymässä komponenteille annettuihin nimiin ja suorittamalla monadisilla käskyillä komponenteille haluttuja muutoksia. Käytännössä siis tämä malli noudattaa näkymän inkrementaalista ja imperatiivista päivittämistä, vaikka isäntäkielenä onkin funktionaalinen ohjelmointikieli ja alkunäkymä määritetään deklaratiivisesti. Siksi tämän tutkimuksen kannalta aiemmin tarkemmin esitellyt kirjastot Fudgets, Fruit ja FranTk ovat kiinnostavimpia ratkaisuja vastaesimerkkeinä perinteisemmille inkrementaalisille komponenttikirjastoille.

4.3 2000-luvusta nykyhetkeen – suorat komponenttikirjastot

Kenties merkittävimmät kehitykset suosittujen GUI-komponenttikirjastojen toimintaperiaatteissa ovat tapahtuneet 2000- ja 2010-luvuilla. Kuten aiemmin nähtiin, inkrementaalisesti toimivat komponenttikirjastot olivat aikaisemmin olleet valtavirtaa, mutta tänä aikajaksona suosituissa komponenttikirjastoissa alkoi esiintyä myös muita arkkitehtuureja. Tässä luvussa esittelen tunnettuja komponenttikirjastoja, jotka eivät käytä inkrementaalista mallia käyttöliittymien määrittämiseen.

Varhaisin löytämäni kiinnostava suuntaus komponenttikirjastoissa 2000-luvun alussa olivat IMGUI-paradigma (engl. *immediate-mode GUI*) ja sitä edeltänyt Zero Memory Widget -kirjasto (ks. Excoffier 2003). Molemmat perustuivat samoihin käsitteisiin, mutta paradigma saavutti lopulta enemmän suosiota IMGUI-nimellä. IMGUI viittaa käyttöliittymän määrittelytyyliin, jossa näkymä piirretään halutulla päivitystiheydellä kokonaan uudestaan näkymän aiempaa tilaa käyttämättä (ks. “About the IMGUI paradigm · ocornut/imgui Wi-

ki" 2023a; "Immediate-Mode Graphical User Interfaces (2005)" 2023; "Immediate Mode Model/View/Controller" 2023a). IMGUI viittaa tähän termillä *välitön piirtäminen* (engl. *immediate-mode rendering*), missä näkymä piirretään ohjelmakoodin säilömän tilan pohjalta uudelleen jokaisella päivityssyklillä grafiikkakirjastoa kutsumalla ("Immediate Mode Model/View/Controller" 2023b). Vaihtoehtona sille on *tilapohjainen piirtäminen* (engl. *retained-mode rendering*), jossa grafiikkakirjasto pitää yllä tilaa näkymän osista ja huolehtii niiden piirtämisestä näytölle kunakin hetkenä, ja ohjelmakoodi ainoastaan ohjaa grafiikkakirjastoa. Huomionarvoista on, että välitöntä piirtämistä on käytetty GUI-kehityksessä jo kauan ennen IMGUI:n esittelyä käsitteenä – esimerkiksi videopelien käyttöliittymät on usein piirretty tätä menetelmää käyttäen ("About the IMGUI paradigm · ocornut/imgui Wiki" 2023b). Toisaalta useimpien tähän asti tässä tutkimuksessa esiteltyjen työpöytäsovellusten kehittämiseen tarkoitettujen korkean tason komponenttikirjastojen voidaan nähdä käyttäneen tilapohjaista piirtämistä käyttöliittymän määrittämiseen – esimerkiksi listauksen 4 Motif-ohjelmassa ohjelmakoodi määrittää näytölle aluksi piirrettävät Motif-komponentit ja saa takaisin viitteet näihin komponentteihin, ja tämän jälkeen näkymän päivittäminen tapahtuu antamalla käskyjä komponenteille, jolloin Motif-kirjaston ylläpitämä sisäinen tila muuttuu, ja kirjasto piirtää näytölle päivittynyttä tilaa vastaavan näkymän. Ohjelmakoodi ei siis itse suoraan ylläpidä lopullista tilaa, jonka kautta näkymä piirretään. IMGUI-paradigman oivallus oli, että välitön piirtäminen on varteenotettava vaihtoehto myös varsinaisten GUI-komponenttikirjastojen käytettäväksi, kun aiemmin valtaosa komponenttikirjastoista oli perustunut tilapohjaiseen piirtämiseen.

Esimerkkinä IMGUI-tyylillä toteutetusta käyttöliittymästä toimii Dear ImGui -kirjastolla toteutettu ohjelmakoodi listauksessa 4.5, joka on mukailtu kirjaston koodiesimerkeistä ("ocornut/imgui" 2023). Ohjelmakoodissa määritetään painike ja tekstikomponentti, joka näyttää kuinka monta kertaa painiketta on painettu. Lisäksi siinä määritetään liukusäädin, joka on kytketty muokkaamaan muuttujan `f` arvoa. Ohjelmakoodista voidaan nähdä, että näkymän komponentit piirretään silmukan sisällä toistuvasti uudelleen imperatiivisia käskyjä käyttäen. Näkymän dynaaminen päivittyminen voidaan toteuttaa yksinkertaisesti muuttamalla silmukan kierroksella tehtäviä piirtokäskyjä, esimerkiksi jättämällä jokin komponentti piirtämättä. Käyttöliittymän tilaa voidaan säilöä esimerkiksi silmukan iteraatioiden ulkopuolisiin staattisiin muuttujiin, jolloin seuraavalla kierroksella piirretty näkymä päivittyy automaattises-

```
static int counter = 0;
static float f = 0.0f;

// Main loop
while (true)
{
  // Create a window called "Hello, world!" and append into it.
  ImGui::Begin("Hello, world!");

  // Buttons return true when clicked (most widgets return true when edited/activated)
  if (ImGui::Button("Button"))
    counter++;
  ImGui::SameLine();
  ImGui::Text("counter = %d", counter);

  // Edit 1 float using a slider from 0.0f to 1.0f
  ImGui::SliderFloat("float", &f, 0.0f, 1.0f);

  ImGui::End();
}
```

Listaus 4.5: Laskurinäkymä Dear ImGui -kirjastolla (C++) ("ocornut/imgui" 2023)

ti vastaamaan uutta tilaa. Tästä toimii esimerkkinä `counter`-muuttujan arvo, joka piirretään näkymän tekstikomponenttiin. Reagointi käyttäjäsyötteisiin tapahtuu koodiesimerkissä synkronisesti näkymän piirtokäskyjen lomassa, esimerkiksi kun näkymän painike palauttaa tosi-arvon, ja `counter`-muuttujan arvoa päivitetään.

IMGUI-paradigman esittämien välittömän ja tilapohjaisen piirtämisen voidaan nähdä vastaavan hyvin läheisesti tässä tutkimuksessa esitettyjä suoran ja inkrementaalisen näkymämäärittelyn käsitteitä. IMGUI:n mukaisessa välittömässä piirtämisessä, kuten listauksessa 4.5, piirtosilmukan sisäisen näkymän määritelmän voidaan katsoa olevan funktio tilasta hetkelliseen

näkymään kaavan 3.3 alemman rivin mukaisesti. Syötteenä tälle funktiolle toimii silmukan ulkopuolinen tila, tässä tapauksessa ohjelmakoodin muuttujat saman kaavan alemman rivin mukaisesti. Silmukan toistuvan suorittamisen myötä funktion uusin arvo piirretään kunakin ajan hetkenä näytölle. IMGUI-kirjastoissa näkymän hetkellinen määrittäminen puolestaan tapahtuu imperatiivisilla käskyillä – komponentteja voidaan lisätä näkymään käskyillä ja asemoida toistensa suhteen esimerkiksi samalle riville `SameLine()`-käskyllä. Reagointi käyttäjäsyötteisiin suoritetaan lukemalla komponenttien palauttamia tapahtumia ja suorittamalla haluttu ohjelmakoodi komponenttien määrityksen lomassa, mutta halutessa tapahtuman jälkeen suoritettava ohjelmakoodi voidaan abstrahoida erillisiin aliohjelmiin, jolloin tapahtumienkäsittely muistuttaa syntaksiltaan takaisinkutsufunktioiden käyttöä.

Tämän tutkimuksen kannalta välitön ja tilapohjainen piirtäminen ovat hyvin kiinnostavia käsitteitä, sillä ne kuvaavat vastaavia ilmiöitä kuin suora ja inkrementaalinen näkymämäärittely. Välittömän ja tilapohjaisen piirtämisen käsitteet ovat kuitenkin syntyneet havaintoina käytännön GUI-ohjelmoinnista, kun taas tämän tutkimuksen esittelemät termit pohjautuvat käsiteanalyysiin siitä, mitä osia käyttöliittymän määritelmässä on minimissään esiinnyttävä, ja mitä osia on hyödyllistä käyttää. Välittömästä ja tilapohjaisesta piirtämisestä GUI-kehityksessä ei tekemäni kartoituksen perusteella ole juurikaan tehty akateemista tutkimusta. Täten tämän tutkimuksen esittämät käsitteet voivat olla hyödyllinen lisä jatkotutkimuksen tekemiselle. Lisäksi tutkimuksen käsitteiden hyödyllisyyttä vahvistaa, että vastaavat ilmiöt on jo aiemmin tunnistettu ohjelmistokehitysyhteisön keskuudessa. Tämä tutkimus esittää myös suoran ja inkrementaalisen näkymämäärittelyn käsitteet hieman välittömän ja tilapohjaisesen piirtämisen käsitteistä poiketen – ensimmäiset perustuvat näkymän mallintamiseen funktiona. Jälkimmäiset puolestaan perustuvat siihen, millaisen rajapinnan GUI-komponenttikirjasto tarjoaa näkymän dynaamiseen päivittämiseen – säilöökö se tietoa näytölle piirrettyjen komponenttien tiloista, vai piirtääkö se näkymän näytölle pelkästään ohjelmakoodin antamien ohjeiden perusteella kunakin ajan hetkenä. Molempien määritelmätyyppien voidaan nähdä kuitenkin viittaavan samoihin ilmiöihin – joko näkymämääritelmä hyödyntää edellistä näkymän tilaa uuden hetkellisen näkymän muodostamiseen, tai se määrittää jokaisen hetkellisen näkymän kokonaisuudessaan jonkin muun tilan pohjalta. Tässä tutkimuksessa myös tunnistettiin luvussa 3.2, että ero suoran ja inkrementaalisen näkymämäärittelyn välillä ei riipu pelkästään komponenttikirjaston rajapinnasta, kuten välittömän

ja tilapohjaisen piirtämisen määritelmissä, vaan kyseessä on ennen kaikkea näkymämääritelmän arkkitehtuurillinen valinta. Komponenttikirjasto voi toki silti ohjata käyttämään jompaa kumpaa näkymän määrittelytyyliä, ja sen suorituskyky voi olla parhaimmillaan, kun sitä käytetään suunnitellusti. Lisäksi tässä tutkimuksessa kiinnitettiin huomiota suoran ja inkrementaalisen määrittelyn sekä imperatiivisen ja deklaratiivisen määrittelyn käsitteiden eroihin – esimerkiksi IMGUI-paradigmaa noudattavat kirjastot kuten Dear ImGui usein käyttävät imperatiivista mallia hetkellisen näkymän määrittämiseen, ja Casey Muratori korostaa varhaisessa IMGUI-esityksessään myös IMGUI-paradigman synkronista toimintaa. Toisaalta kuten aiemmin tässä luvussa todettiin, suoraan määrittelyyn pohjautuvat käyttöliittymät voivat käyttää myös deklaratiivista lähestymistä. Esimerkiksi Fudgets-kirjasto toimii näin – siinä käyttöliittymä on määritetty deklaratiivisena rakenteena, mutta jokainen hetkellinen näkymä piirretään silti ilman edellisen hetkellisen näkymän tilaa, eikä komponenttikirjasto säilö ohjelmakoodissa määritetyn tilan lisäksi mitään omaa sisäistä tilaansa siitä, miten komponentit näytöllä esitetään.

IMGUI:n jälkeen seuraavat kiinnostavat kehitykset komponenttikirjastoissa tapahtuivat web-ohjelmoinnin parissa. 2010 julkaistiin AngularJS-ohjelmistokehys, joka tarjosi HTML-syntaksilla toimivan sapluunakielen, jolla voidaan esittää dynaamisia käyttöliittymiä ("angular/angular.js: AngularJS - HTML enhanced for web apps!" 2024; "AngularJS — Superheroic JavaScript MVW Framework" 2024). Tähän sapluunakieleen voidaan upottaa avaimia, jotka vastaavat dynaamisesti päivittyviä arvoja, ja AngularJS-kehys piirtää sapluunan ja sen sisältämien avainten esittämän näkymärakenteen niin, että se päivittyy vastaamaan avainten arvoja sitä mukaa, kun niitä muokataan ("AngularJS — Superheroic JavaScript MVW Framework" 2024). Näkymän avaimet voivat olla AngularJS:n JavaScript-rajapinnoilla määritetyn datamallin eli kontrollerin avaimia, jotka AngularJS yhdistää niin sanotulla datasidonnalla (engl. *data binding*) näkymään ("AngularJS — Superheroic JavaScript MVW Framework" 2024). Tämä datasidonta muistuttaa reaktiivista mallia, jossa esitetyn näkymäpuun rakenne päivittyy automaattisesti, kun sen osina käytetyt arvot muuttuvat. AngularJS-kehys sai syntynsä nimenomaan oivalluksesta, että HTML-kieli soveltuu hyvin staattisen näkymän määrittämiseen, mutta ei näkymän dynaamisten muutosten esittämiseen ("AngularJS — Superheroic JavaScript MVW Framework" 2024; "AngularJS: Developer Guide: Introduction" 2024). AngularJS:n tarjoama datasidonta datamallien ja näkymän välillä mahdol-

listaa näkymien määrittämisen niin, että näkymän koko dynaaminen intensio esitetään komponenttipuuna, joka päivittyy vastaamaan näkymän ulkopuolelta tulevaa tilaa. Tämä vastaa siis suoraan näkymän suoraa määrittelyä kaavassa 3.3. Esimerkki AngularJS-sovelluksesta nähdään listauksissa 5, 6 ja 7, joissa AngularJS:llä määritetään muistilistanäkymä, johon voi lisätä rivejä, ja jonka rivejä voidaan merkitä käsitellyiksi. Muistilistan HTML-syntaksi määrittää näkymää esittävän komponenttipuun, joka mukautuu siinä esitettävän datamallin päivityksiin. JavaScript-koodi puolestaan määrittää datamallin, joka näkymään sidotaan `ng-controller`-direktiivillä, ja määrittää, miten dataa ohjelman suorituksen aikana päivitetään. Kyseisessä datamallissa esiintyvät avaimet `todos`, joka on lista näkymässä esitettävistä muistioriveistä, ja `todoText`, jota ei ole erikseen datamallissa alustettu, mutta joka on näkymässä `ng-model`-direktiivillä sidottu syötekenttään, johon voidaan syöttää uuden luotavan muistiorivin teksti. `todoText` päivittyy datamallissa automaattisesti käyttäjän syöttäessä tekstiä syötekenttään. Näkymän painikkeeseen on kytketty `addTodo`-metodi, jonka kutsuminen lisää uuden tietueen `todos`-listaan ja asettaa `todoText`-kentän tyhjäksi. Koska datamalli on sidottu näkymään, päivittyy näkymä automaattisesti, kun malliin tehdään muutoksia. Esimerkiksi syötekenttä tyhjenee, kun `todoText`-avain asetetaan tyhjäksi. `ng-repeat`-direktiivi mahdollistaa listarakenteen sitomisen näkymään niin, että kun muistilistaan lisätään tietue, päivittyy myös näkymä vastaavasti, ja siihen ilmestyy uusi tietuetta vastaava komponentti. Samoin näkymä päivittyy listan tietueita poistettaessa tai muutettaessa. AngularJS tukee myös muita komponenttikirjastojen tarpeellisia ominaisuuksia, kuten uudelleenkäytettävien omaa sisäistä logiikkaansa sisältävien komponenttien määritystä (“AngularJS — Superheroic JavaScript MVW Framework” 2024). Alkuperäisen AngularJS-kehyksen kehitystuki lopetettiin 2022, mutta sen korvaava ohjelmistokehys on saman tyylisesti nimetty Angular (“AngularJS — Superheroic JavaScript MVW Framework” 2024).

Toinen kiinnostava web-pohjainen komponenttikirjasto oli 2011 julkaistu FaxJs, josta myöhemmin kehitettiin suosittu React-kirjasto (“jordwalke/FaxJs: Fax Javascript Ui Framework” 2023). React toimi alunperin kirjastona, joka käytti taustalla web-selainten DOM-rajapintaa dynaamisten käyttöliittymien määrittämiseen. Sittemmin on kehitetty esimerkiksi React Native, jolla React-ohjelmakoodia voidaan hyödyntää käyttöliittymien toteuttamiseen muun muassa Android- ja iOS-mobiililaitteille (ks. “React Native · Learn once, write anywhere”

2023). Reactin keskeinen toimintaperiaate on, että käyttöliittymä koostuu komponenteista, ja komponentit päivittyvät näytöllä automaattisesti vastaamaan niiden syötteinä toimivia tiloja (ks. “Quick Start – React” 2023). Komponentin syötteinä voivat olla joko sille sen ulkopuolelta annetut ajan myötä mahdollisesti vaihtuvat arvot tai komponentin itsensä määrittämä sisäinen muokattava tila. React-komponentti määritetään tavallisesti funktiona, jonka syötteenä ovat komponentille annetut arvot, joiden perusteella funktio voi palauttaa ulostulonaan HTML-koodia muistuttavan JSX-komponenttipuun (ks. (“Quick Start – React” 2023)). Näin komponenttifunktio määrittää komponentin hetkellisen rakenteen kunakin ajan hetkenä sille annettuja syötearvoja vastaavasti. React-kirjasto huolehtii siitä, että komponenttifunktiota kutsutaan aina, kun komponentin syötearvot muuttuvat, ja että näytölle piirretty hetkellinen näkymä vastaa komponenttifunktion palauttamaa rakennetta. Ulkopuolelta tulevien syötearvojen lisäksi komponentti voi myös säilöä omaa sisäistä tilaansa, jota se voi käyttää hetkellisen rakenteensa muodostamiseen. Tämä voidaan tehdä `useState`-funktiolla, joka palauttaa tilan hetkellisen arvon ja funktion, jolla arvoa voidaan muuttaa. React huolehtii siitä, että komponenttifunktiota kutsutaan uudestaan, kun tilan arvoa muokataan, ja mahdolliset muutokset piirretään näytölle. React-komponentti siis vastaa käytännössä suoraan näkymän suoran määritelmän kaavan 3.3 alempaa riviä, missä jonkin tilan perusteella muodostetaan hetkellinen näkymä.

Esimerkkinä yksinkertaisesta React-ohjelmasta toimii listaus 4.6, jossa React-kirjastolla määritetään komponentti, joka toimii laskurina ja näyttää, miten monta kertaa painiketta on painettu. Komponenttifunktion palauttama JSX-lauseke määrittää komponentin rakenteen tietyllä hetkellä, ja se viittaa dynaamisesti päivittyvään `count`-tilaan. `setCount`-funktio päivittää `count`-tilaa, ja se on kytketty painikkeen painallustapahtumiin takaisinkutsufunktiona `onClick`-parametrin kautta. Komponenttifunktio voidaan piirtää näytölle, jolloin kirjasto myös kutsuu komponenttifunktiota aina, kun sen käyttämä tila muuttuu.

React ja AngularJS ovat saavuttaneet laajaa suosiota web-kehityksessä, vaikka ne aluksi tarjosi vain vaihtoehtoisen tavan määritellä käyttöliittymiä esimerkiksi DOM-rajapinnan inkrementaaliseen määrittelyyn verrattuna. Ainakin joissain tilanteissa tämä määrittelytapa on siis todennäköisesti ollut web-kehittäjien mielestä kokonaisratkaisuna DOM:ia ja muita vastaavia inkrementaalisesti toimivia komponenttikirjastoja parempi. Tämän valinnan merkittävä

```
import React, { useState } from ’react’;

function Example() {
  // Declare a new state variable, which we’ll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Listaus 4.6: Laskurinäkymä React-kirjastolla (JavaScript, React) (“Using the State Hook – React” 2023)

tekijä voi olla ero inkrementaalisen ja suoran määrittelyn välillä – React tai AngularJS voidaan valita käyttöön niiden tarjoaman käyttöliittymien suoran määrittelyn ansiosta. Ennen näitä kirjastoja suurin osa dynaamisten web-käyttöliittymien määrityksestä tapahtui inkrementaalisesti – HTML:n käyttäminen näkymän alkutilan määrittämiseen ja DOM:n ja JavaScriptin käyttäminen näkymän dynaamiseen päivittämiseen olivat suoraviivaisimmat tavat dynaamisten web-sivujen kehittämiseen, sillä ne ovat selainten ensisijaiset rajapinnat graafisten käyttöliittymien esittämiseen, ja monet muut GUI-kehityksen kirjastot, kuten jQuery, toimivat niin ikään inkrementaalisesti. Suoraa näkymämäärittelyä käyttävät komponenttikirjastot ovat web-kehityksessä myös pystyneet suorituskykynsä puolesta tarjoamaan riittävän tason, että niitä voidaan käyttää, vaikka ne toimivat inkrementaalisen DOM-rajapinnan päällä. Esimerkiksi Reactin merkittävä ominaisuus on, että se on optimoitu välttämään tarpeetonta näkymän uudelleen piirtämistä. Näkymän suora määrittely olisi mahdollista piirtämällä React-komponenttifunktioiden palauttamat näkymärakenteet aina näkymän tilan päivittyessä kokonaan uudelleen näytölle, mutta inkrementaalisen DOM-rajapinnan päälle rakennettuna tämä vaatisi näkymän kaikkien aiempien komponenttien poistamista ja uusien komponenttien luomista niiden tilalle, mikä puolestaan vaatisi komponenttien tilaa ylläpitävien DOM-objektien uudelleen alustamista tietokoneen muistiin joka kerta, kun näkymä päivitetään näytölle. React sen sijaan käyttää tarpeettomien DOM-operaatioiden välttämiseksi sisäistä VDOM-rakennettaan, joka mahdollistaa näkymän tilan päivittyessä vertailun siitä, mitkä paikat uuden ja edellisen näkymärakenteen välillä ovat muuttuneet (“Reconciliation – React” 2023). Näin näkymää päivittäessä React voi suorittaa tarvittavat operaatiot DOM-rajapinnan yli ainoastaan niihin komponentteihin, jotka ovat muuttuneet.

Kolmas suoraa näkymämäärittelyä hyödyntänyt web-kehityksen ratkaisu oli Elm-ohjelmointikieli ja komponenttikirjasto, joka esiteltiin samoihin aikoihin 2013 (Czaplicki ja Chong 2013). Se toimii Reactin tapaisesti niin, että näkymä määritetään funktiona, joka vastaanottaa tilaa ja palauttaa sen perusteella hetkellistä näkymää esittävän rakenteen. Esimerkki Elmistä on listauksessa 4.7 esitetty laskuriohjelma. Reactista ja JavaScriptistä Elm eroaa siten, että se on ohjelmointikielenä funktionaalinen ja viitteellisesti läpinäkyvä (*engl. referentially transparent*). Lisäksi Elm määrittää arkkitehtuurissaan tarkasti näkymän sisäisen tilan käsittelyn periaatteet – tilaa päivitetään diskreettien tapahtumien yhteydessä funktiolla, joka vastaanottaa viimeisimmän tapahtuman ja edellisen tilan, palauttaen sitten näkymän

uuden tilan (Czaplicki ja Chong 2013; “The Elm Architecture · An Introduction to Elm” 2023). Tämä malli muistuttaa suoraan tässä tutkimuksessa esitettyä suoran näkymämäärittelyn kaavan 3.3 ensimmäistä riviä, jossa tilan muutokset käsitellään näkymän muodostamisesta erillään. Elm oli yksi varhaisista web-kehitysratkaisuista, joka erotti tilan käsittelyn ja näkymän muodostamisen toisistaan. Myöhemmin sama arkkitehtuuri on ollut saatavilla myös esimerkiksi React-kehityksessä Redux-kirjaston kautta. Tilan ja näkymän muodostamisen erottaminen mahdollistaa esimerkiksi tilan tallentamisen pitkäkestoiseen muistiin ja myöhemmin palauttamisen, jolloin näkymän aiempi tila voidaan helposti palauttaa näytölle. Toinen mahdollinen hyöty on, että samaa tilaa voidaan käyttää syötteenä kahden tai useamman komponentin piirtämiseen.

Eräs tuore huomionarvoinen komponenttikirjasto on vuonna 2017 julkaistu Flutter. Flutter on suunnattu tukemaan työpöytäkäyttöjärjestelmiä (Windows, MacOS, Linux-distribuutiot), mobiililaitteita (Android, iOS), web-selaimia ja sulautettuja järjestelmiä (ks. “Flutter - Build apps for any screen” 2023). Se on kiinnostava kehitys GUI-komponenttikirjastoissa, sillä sen taustalla on suuri monikansallinen yritys Google, ja se hyödyntää nimenomaan suoraa määrittelyä näkymien määrittämiseen. Esimerkki tästä nähdään liitteen A listauksessa 8 – kirjasto päivittää automaattisesti näkymän vastaamaan uutta tilaa, kun `_clickCount`-muuttujan arvo muuttuu.

Flutter on yksi harvoista 2010-luvun jälkeen julkaistuista suurempaa huomiota saavuttaneista alustariippumattomista komponenttikirjastoista, joten sen valinta käyttää suoraa näkymämäärittelyä on huomionarvoinen kehitys komponenttikirjastojen historiassa. Samoin Reactin ja sitä myöhemmin seuranneiden kirjastojen kuten Vuen ja Svelten suosio web-kehityksessä noudattaa samaa kaavaa, jossa aiemmin inkrementaalisten komponenttikirjastojen hallitsema kehitysala siirtyy hyödyntämään enemmissä määrin suoraa näkymämäärittelyä.

```
import Browser
import Html exposing (Html, button, div, text)
import Html.Events exposing (onClick)

main =
  Browser.sandbox { init = 0, update = update, view = view }

type Msg = Increment | Decrement

update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1

view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (String.fromInt model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

Listaus 4.7: Laskurinäkymä Elm-ohjelmointikielellä (“Introduction · An Introduction to Elm” 2023)

5 Määrittelytapojen erot

Tässä tutkimuksessa on tähän mennessä esitetty teoria graafisten käyttöliittymien määrittelyn minimalistisille lähestymistavoille, ja teoriaa on peilattu käytännön ratkaisuihin erilaisista komponenttikirjastoista. Nähtiin, että esitellyt komponenttikirjastot noudattivat kaikki joko suoran tai inkrementaalisen määrittelyn periaatteita, ja tyypillisesti tämä valinta jakoi kirjastot kahteen leiriin. Suora ja inkrementaalinen määrittely käsitteinä riittävät siis kuvailemaan yhden perustavanlaisen käyttöliittymämäärittelyn tekijän, joka vaikuttaa näkymän dynaamisen toiminnan semantiikkaan ja sen ohjelmakoodin rakenteeseen. Ne ovat myös perustavan tasoinen ratkaisu komponenttikirjaston muita toiminnallisuuksia määritettäessä. Tässä luvussa pohditaan vielä tarkemmin käyttöliittymien mallinnusta näiden käsitteiden kautta sekä verrataan näiden lähestymistapojen eroja käsiteanalyysin keinoin.

5.1 Inkrementaalisen määrittelyn edut

Kuten luvussa 4 todettiin, varhaiset komponenttikirjastot toimivat valtaosin inkrementaalista näkymämäärittelyä käyttämällä. Usein niissä myös hetkelliset näkymät ja niissä tapahtuvat muutokset määritettiin imperatiivisilla käskyillä. Myöhemmin komponenttikirjastoissa alkoi esiintyä vaihtoehtoisia määrittelytapoja, ennen kaikkea erilaisia suoran määrittelyn malleja näkymien määrittämiseen. Varhainen esimerkki tästä oli Fudgets, myöhemmin laajempaa suosiota ovat saavuttaneet esimerkiksi React ja Flutter. Siksi määrittelytapojen vertailu on kiinnostavaa, sillä niiden erot on hyvä tiedostaa komponenttikirjastoa valitessa.

Inkrementaalisella näkymämäärittelyllä on omat piirteensä ja etunsa suoraan määrittelyyn verrattuna. Se voidaan nähdä intuitiiviseksi tavaksi käyttöliittymän esittämiseen, sillä siinä muokataan näkymää vaihe vaiheelta tapahtumien yhteydessä, mikä muistuttaa jonkin reaalimaailman objektin käsittelyä. Erityisesti jos käyttöliittymän dynaamiset muutokset tapahtumien yhteydessä ovat pieniä, on ne helppo esittää muutoksina näkymän edeltävään tilaan. Inkrementaalisessa määrittelyssä on mahdollista hyödyntää näkymään itseensä säilöttyä tilaa näkymän päivittämiseen, mikä voi olla suoraviivainen tapa käyttöliittymän dynaamisen toiminnan määrittämiseen. Kaavassa 3.4 esitettiin inkrementaalisen määrittelyn osaksi lisäksi

näkymän rinnalla käsiteltävä erillinen tila, sillä osa näkymän muodostamiseen tarvittavasta tilasta ei välttämättä luontevasti tallennu osaksi varsinaista näkymää. Esimerkiksi liitteen A listauksessa 3 `itemsAdded`-tieto on luontevaa tallentaa näkymän ulkopuolelle, kun taas esimerkiksi tieto muistilistanäkymään lisättyjen muistiinpanojen teksteistä voitaisiin lukea suoraan näkymän komponenteista komponenttikirjaston tarjoamien funktioiden avulla, eikä tätä tietoa tarvitse siksi säilöä erilliseen näkymän ulkoiseen tilaan. Esimerkissä `getParam`-funktiolla voidaan lukea nimetyn komponentin jokin sen hetkinen parametri, ja tietyn komponentin tekstisisällön voisi lukea esimerkiksi `getText`-funktiolla. Lisäksi komponenttikirjasto voi mahdollistaa myös mielivaltaisen ei-visuaalisen datan säilömisen näkymään – esimerkiksi DOM-rajapinnassa tätä varten ovat tarjolla data-attribuutit (ks. "Using data attributes - Learn web development | MDN" 2023). Ne mahdollistavat mielivaltaisen tekstimuotoisen datan tallentamisen mihin vain komponenttiin niin, ettei se vaikuta näkymän visuaaliseen esitykseen. Dataa voidaan lukea komponenteista näitä attribuutteja lukemalla. Tällöin näkymä itsessään voi toimia käyttöliittymämäärityksen ainoana tilaa sisältävänä objektina, eikä varsinaista erillistä tilaa tarvita. Voidaan väittää, että näiden inkrementaalinen määrittely voi olla varsin suoraviivainen ja intuitiivinen tapa käyttöliittymien esittämiseen, mikä voi osin selittää sen suosiota monissa komponenttikirjastoissa.

Inkrementaalisessa näkymämäärittelyssä on mahdollisena etuna myös, että siinä hetkellisen näkymän muodostaminen tapahtuu samalla, kun käyttöliittymän tapahtumia käsitellään, kun taas suorassa määrittelyssä tapahtumien yhteydessä päivitetään vain käyttöliittymän tilaa, ja tarvitaan lisäksi erillinen kuvaus, joka muuttaa hetkellisen tilan näkymäpuuksi. Tämä tilan muuntaminen näkymäksi on siis ylimääräinen työvaihe inkrementaaliseen määrittelyyn verrattuna. Esimerkki tästä erosta nähdään liitteen A listausten 3 ja 2 välillä – jälkimmäisessä listauksessa `view`-funktio kuvaa näkymän ulkoisessa tilassa säilötyn tiedon listan alkioista näkymän komponenteiksi. Ensimmäisessä listauksessa sen sijaan tätä muunnosta ei tarvitse määrittää, sillä tapahtumien yhteydessä päivitetään ulkoisen tilan sijasta suoraan itse näkymää haluttuun muotoon. Tässä tapauksessa toki jälkimmäisessä listauksessa käytetty kuvaus tietuelistasta komponenttilistaksi on varsin suoraviivainen, mutta se on joka tapauksessa inkrementaaliseen määritykseen verrattuna ylimääräinen määritelmävaihe, jolla näkymän ulkoinen tila sidotaan muodostettavaan näkymään. Joissain suoraa määrittelyä käyttävissä komponenttikirjastoissa sen sijaan yksinkertaisemman tiedon esittäminen näkymässä ei lisää

kompleksisuutta inkrementaaliseen määrittelyyn verrattuna – näitä ovat kirjastot, jotka tukevat reaktiivisten arvojen kytkemistä suoraan näkymän komponenttien attribuutteihin tai arvoihin. Ilmiöstä voidaan käyttää myös termiä kaksisuuntainen datasidonta, kuten AngularJS-kirjaston tapauksessa (ks. "angular/angular.js: AngularJS - HTML enhanced for web apps!" 2024). Esimerkiksi syötekomponentille voidaan välittää reaktiivinen muuttuja, jolloin syötekomponentti päivittyy vastaamaan muuttujan arvoa sen vaihtuessa ohjelmallisesti ja toisaalta päivittää muuttujaa käyttäjän muokatessa syötekomponentin sisältöä (ks. listaukset 5, 6 ja 7). Tällöin näkymän koodi ei ole monimutkaisempaa kuin inkrementaalisen komponenttikirjaston koodi, jossa syötekomponenttiin ei jouduta kytkemään jotain ulkoisen tilan avainta, mutta komponentille joudutaan määrittämään jokin viite, että sen syötearvoon voidaan myöhemmin viitata. Näin voidaan yksinkertaistaa tilan käsittelyä, kun kyseessä on yksinkertainen arvo, joka sopii sellaisenaan kytkettäväksi suoraan jonkin näkymäkomponentin attribuuttiin. Joissain suoraa määrittelyä käyttävissä komponenttikirjastoissa tällaista toiminnallisuutta ei kuitenkaan ole, ja siten myös yksinkertaisemman tilan sitominen osaksi näkymää on monimutkaisempaa kuin inkrementaalisissa kirjastoissa – esimerkiksi React-kirjastossa jonkin syötekentän arvo pitää syöttää `value`-attribuuttina näkymäkomponentille sovelluksen tilasta, ja syötekentän arvo pitää sitten kenttään kytketyssä takaisinkutsufunktiossa päivittää tilaan käyttäjän muokatessa syötekentän tekstiä, eli tilan esittäminen ja päivittäminen näkymän kautta vaatii kahden eri kytkennän tekemistä näkymäkomponenttiin (ks. "Thinking in React – React" 2023). Lisäksi joissain tilanteissa kumpikaan määrittelytapa ei ole suoraviivaisesti nähtävissä toista paremmaksi – esimerkiksi voidaan kuvitella tilanne, jossa näkymässä on valitsin, jonka alle piirretään erilainen komponenttipuu sen mukaan, onko valitsin valittuna vai ei. Tässä on siis kyseessä tilanne, jossa jokin toiseen näkymäkomponenttiin liitetty tila hallitsee näkymän muun osan piirtämistä. Inkrementaalisessa määrittelyssä suoraviivainen ratkaisu on, että valitsimeen kytketyssä takaisinkutsufunktiossa valitaan, kummanlainen komponenttipuu piirretään, ja korvataan aiempi komponenttipuu uudella. Sen sijaan suorassa määrittelyssä käyttöliittymän tila päivittyy automatisoidusti valitsimen arvon muuttuessa, ja hetkellistä näkymää tilasta muodostettaessa päätetään yksinkertaisella ehtolausekkeella tai ehtolauseella, millainen komponenttipuu valitsimen alle piirretään. Esimerkit näistä tilanteista nähdään liitteen A listauksissa 9 ja 10. On vaikea määrittää objektiivisesti, onko edullisempaa määrittää piirrettävä komponenttipuu tapahtuman yhteydessä näkymää päivit-

tämällä vai näkymän päivittämisen yhteydessä ehtolauseella. Joissain tapauksissa siis inkrementaalinen määrittely voi tehdä tilan ja näkymän kytkennän suoraviivaisemmaksi, mutta usein määrittelytavat ovat tältä osin melko yhdenvertaisia.

5.2 Määrittelytapojen neutraalit erot

Yksi varsin neutraali ero inkrementaalisen ja suoran määrittelyn välillä on, että inkrementaalisessa määrittelyssä näkymän komponentteihin tarvitaan jonkinlaisia viitteitä, jotta niiden sisältämää tilaa voidaan hakea ja päivittää. Tämä nähdään esimerkiksi liitteen A listauksessa 3. Usein komponenttien viitteet tallennetaan ohjelmointikielen muuttujiin. Tämä on vaihe, jota suorassa näkymämäärittelyssä ei tarvitse tehdä, mutta toisaalta myös suorassa näkymämäärittelyssä tarvitsee nimetä ne tilan tilan osat, joihin komponentit ovat kytkettyinä, ja tilan osien ja komponenttien kytkennät pitää määrittää. Kyse on varsin pienistä eroista ohjelmakoodin kompleksisuuden suhteen. Lisäksi viitteiden määrittämisestä voi olla hyötyä myös suorassa määrittelyssä – viitteitä komponenttiin voidaan tarvita, jos halutaan esimerkiksi saada komponenttikirjastolta tietoa siitä, minkä kokoisena komponentti on tällä hetkellä piirrettynä näytölle. Joissain suoraa määrittelyä käyttävissä komponenttikirjastoissa viitteitä saatetaan tarvita myös, jos komponentit pystyvät sisältämään jotain omaa sisäistä tilaansa – esimerkiksi React-komponenttikirjastossa vieritettävien (*engl. scrollable*) komponenttien vierityskohta on komponentin sisäistä tilaa, jota ei voida hallita suoraan ulkoiseen tilaan kytkemällä, vaan komponenttiin on muodostettava viite esimerkiksi `useRef`-funktiolla, ja sitten imperatiivisesti asetettava komponentin vierityskohta esimerkiksi `scrollTo`-funktiolla (ks. "useRef – React" 2023; "Element: scrollTo() method - Web APIs | MDN" 2023).

Suorituskyvyn suhteen inkrementaalisen käyttöliittymämäärittelyn eduksi voidaan laskea se, että inkrementaalinen komponenttikirjasto voi helposti optimoida näytölle piirtämistä niin, että näyttöä päivitetään vain niiden komponenttien osalta, joita muokataan komponenttikirjaston käskyillä (ks. "About the IMGUI paradigm · ocornut/imgui Wiki" 2023b). Esimerkiksi jos näkymän jonkin tekstikentän tekstiä muutetaan käskyllä komponenttikirjastolle, riittää, että komponenttikirjasto piirtää vain kyseisen tekstikentän alueen uudestaan, ja muut alueet voidaan pitää muuttumattomina. Näin voidaan säästää näytön piirtämiseen tarvittavia laskentaresursseja. Tämä on varsinkin varhaisissa komponenttikirjastoissa ollut tärkeä etu,

sillä tietokoneiden laskentaresurssit ovat olleet rajallisia (ks. “About the IMGUI paradigm · ocornut/imgui Wiki” 2023b). Nykyisin kuitenkin tietokoneiden suorituskyky on kohonnut merkittävästi, eikä näkymän piirtäminen täysin uudelleen jokaisen päivityksen yhteydessä aiheuta useimmille laitteille ongelmia – esimerkiksi videopeleissä tämä on yleisin tapa käyttöliittymien määrittämiseen (ks. “About the IMGUI paradigm · ocornut/imgui Wiki” 2023b). Lisäksi monet uudemmat komponenttikirjastot pystyvät tarjoamaan käyttäjälleen suoran näkymämäärittelyn mutta kuitenkin optimoimaan näkymän piirtämisen niin, ettei näytölle tehdä tarpeettomia päivityksiä. Esimerkki tästä on React – kuten luvussa 4.3 todettiin, se tarjoaa suoraan toimivan rajapinnan näkymien määrittämiseen, mutta se voi sisäisesti käyttää inkrementaalista DOM-rajapintaa näkymän piirtämiseen, jolloin DOM-rajapinnan tarjoava web-selain voi optimoida näytölle piirtämisen rajoittumaan vain päivittyviin alueisiin. Suorituskyky ei siis nykyisin ole ratkaiseva tekijä komponenttikirjaston toimintatapaa valitessa.

5.3 Suoran määrittelyn edut

Suorassa näkymämäärittelyssä on kuitenkin myös etunsa inkrementaaliseen määrittelyyn verrattuna. Kenties ilmeisin etu on, että suorassa määrittelyssä näkymän koko intensio voidaan määrittää yhtenä puumaisena kuvauksena, joka määrittää kuvauksen käyttöliittymän tilasta hetkelliseksi näkymäksi, kun taas inkrementaalisessa esityksessä näkymän määritys väistämättä jakaantuu alkunäkymän ja tapahtumien yhteydessä ajettavien takaisinkutsurutiinien välille. Koska inkrementaalinen määritys perustuu näkymän manipulointiin sen aiemman tilan perusteella, ei sen esittäminen yhtenä puumaisena kuvauksena ole mahdollista, vaan mahdolliset muutokset on pakko esittää jossain muodossa näkymän alkutilasta erillään. Suorassa määrityksessä sen sijaan nämä tilan muutokset voidaan eristää näkymän käyttämän tilan käsittelyyn, ja itse näkymän määritelmä voi olla suoraviivainen funktio tilasta hetkelliseen näkymään. Tämän ominaisuuden edukkuus voi riippua käyttökohteesta, mutta sen ansiosta on esimerkiksi mahdollista nähdä näkymän dynaamiset ja staattiset kohdat ilman, että tarvitsee analysoida, mitä tapahtumia ja missä järjestyksessä sovellusta suorittaessa voi tapahtua. Voidaan myös helposti nähdä, mitkä näkymän osat riippuvat mistäkin tilasta. Inkrementaalisessa näkymämääritelmässä sen sijaan on tiedettävä, mitä tapahtumia ja sitä kautta muutoksia käyttöliittymässä voi tapahtua, ennen kuin saa-

daan tarkempaa tietoa näkymän mahdollisista tiloista suorituksen aikana. Suora määrittely voi tehdä myös näkymän ohjelmallisen staattisen analyysin helpommaksi. Esimerkki tästä on ESLint-koodintarkastustyökalun React-lisäosa, joka pystyy tarkistamaan, että websovelluksen linkkielementtien attribuutit on määritelty hyviä tietoturvakäytänteitä noudattaen (ks. “eslint-plugin-react/docs/rules/jsx-no-target-blank.md at master · jsx-eslint/eslint-plugin-react” 2023). Tämä on suoraa määrittelyä käyttävässä React-kirjastossa helpompaa kuin inkrementaalisesti määritellyssä näkymässä, sillä näkymän puurakenteesta voidaan suoraan nähdä, mitä attribuutteja kullekin linkkielementille suorituksen aikana annetaan. Inkrementaalisessa näkymän määrittelyssä sen sijaan pitäisi analysoida kaikkia ohjelmakoodissa näkymään tehtäviä muutoksia, jotta voitaisiin nähdä, mitä attribuutteja linkkielementeille suorituksen aikana mahdollisesti asetetaan. Näiden etujen ulkopuolella kuitenkin määrittelytavan valintaan vaikuttaa myös, halutaanko näkymä ennemmin esittää alkunäkymän ja siihen tehtävien muutosten kautta, vai yhtenä puuna, joka esittää kaikki näkymän mahdolliset tilat. Joissain tilanteissa tämä voi olla ratkaiseva tekijä teknisiä ratkaisuja tehdessä – esimerkiksi alkunäkymän esitys voi inkrementaalisessa komponenttikirjastossa olla suoraviivaisempi kuin suoran komponenttikirjaston määritelmä, joka esittää kaikki näkymän mahdolliset tilat. Toisaalta myös tässä suoralla määrittelyllä on etuja – näkymän määritelmä pysyy yksinkertaisempana, kun se on yksinkertainen funktio tilasta hetkelliseen näkymään, eikä siinä tehdä varsinaista tilan käsittelyä kuten inkrementaalisessa määrittelyssä väistämättä tapahtumien ja näkymämuutosten yhteydessä tehdään. Näkymän määritelmässä ei myöskään suorassa määrittelyssä voi niin helposti esiintyä tilaa, joka jää piiloon komponenttien sisäisenä implisiittisenä tilana – esimerkki tästä voisi olla alasvetovalikko, joka inkrementaalista komponenttikirjastoa käyttäessä voisi olla avattuna käyttäjän toimesta ilman, että ohjelmoija on erikseen määrittänyt sitä avattavaksi.

Suoran määrittelyn eduksi inkrementaalisen yli on usein esitetty myös, että inkrementaalisessa määrittelyssä tilaa voidaan joutua kahdentamaan näkymän sisäisen tilan ja näkymän ulkoisten datarakenteiden välillä (ks. “About the IMGUI paradigm · ocornut/imgui Wiki” 2023b; “Immediate-Mode Graphical User Interfaces (2005)” 2023; “Immediate Mode Model/View/Controller” 2023a). Esimerkki tällaisesta tilanteesta voi olla sovellus, jonka inkrementaalisesti määritetyssä käyttöliittymässä on valintakenttä, ja jossa suoritetaan ajastetusti toistuvasti jotain rutiinia, jonka toiminta riippuu valintakentän arvosta. Tällöin valintaken-

tän arvo on käytännöllistä tallentaa sovelluksessa esimerkiksi globaaliin muuttujaan, jolloin se on suoraan luettavissa ajastetun rutiinin sisältä. Esimerkki tästä nähdään liitteen A listauksessa 11. Tämä kuitenkin johtaa tilanteeseen, jossa valintakentän arvoa säilytetään sekä sovelluskoodin omassa tilassa näkymän ulkopuolella että komponenttikirjaston tilassa näkymän sisällä. Inkrementaalisessa näkymämäärittelyssä sovelluksen komponenteilla on kullakin oma sisäinen tilansa, jonka mukaan ne muokkautuvat, kun niille sovelluksen suorituksen aikana annetaan käskyjä. Niinpä valintakentällä on myös oltava oma sisäinen tilansa. Näiden tilojen pitäminen samanlaisina voi olla haastavaa, sillä se vaatii, että aina jos sovelluksen säilömää tilaa muokataan, myös komponentin tila pitää päivittää vastaavaksi, ja komponentin tilan päivityksien yhteyteen pitää kytkeä jonkinlainen takaisinkutsufunktio tai muu käsittelymekanismi, jolla sovelluksen tila päivitetään vastaamaan näkymän kautta tulevaa uusinta tietoa. Voidaan nähdä, että saman tilan ylläpitäminen tällä tavalla kahdessa paikkaa kiistatta aiheuttaa ohjelmakoodiin lisää monimutkaisuutta ja mahdollisia virheenlähteitä. Mikäli inkrementaalisesti toimiva komponenttikirjasto johtaa tällaisiin ratkaisuihin, voidaan sen siis sanoa olevan tältä osin heikompi ratkaisu kuin esimerkiksi suoraa määrittelyä hyödyntävä komponenttikirjasto.

Tilan kahdentaminen komponenttikirjaston ja sovelluskoodin välillä ei kuitenkaan ole väistämätön ratkaisu inkrementaalista näkymämäärittelyä käytettäessä. Aiemmassa valintakenttää ja ajastettua toistuvaa rutiinia kuvanneessa esimerkissä valintakentän arvo oltaisiin voitu globaaliin muuttujaan kahdentamisen sijasta myös lukea suoraan valintakentän näkymäkomponentista. Inkrementaalisissa komponenttikirjastoissa on yleisesti hyvä tuki siihen, että yksittäiseen komponenttiin voidaan viitata näkymän ulkopuolelta ja sen attribuutteja voidaan lukea tai muokata. Ajastetussa rutiinissa voitaisiin siis valintakentän arvo hakea aina esimerkiksi kutsumalla muuttujaan säilötyltä komponentilta funktiota, joka palauttaa sen hetkellisen arvon. Näin voidaan välttää tilan kahdentaminen. Näkymä väistämättä sisältää oman tilansa, koska sitä hallitaan inkrementaalisesti päivittäen, ja nämä inkrementaaliset päivitykset esitetään aina suhteessa näkymän edelliseen tilaan, jota komponenttikirjasto säilöö. Jos siis tilan kahdentamista halutaan välttää, näkymän sisäisestä tilasta on tehtävä koko käyttöliittymän tilan ensisijainen totuudenlähde. Tällöin näkymän määrittely tehdään niin, että jos jokin tieto voidaan lukea suoraan näkymän komponenttien kautta, ei sitä kahdenneta muuhun tilaan, ja kun käyttäjäsyötteiden tai muiden tapahtumien yhteydessä näkymää päivitetään,

tehdään päivitykset suoraan näkymän visuaalisiin komponentteihin aina kun mahdollista sen sijasta, että ensin muokattaisiin jotain näkymän ulkoista tilaa ja sitten päivitettäisiin näkymä vastaamaan tätä tilaa.

Usein kuitenkin on perusteltuja syitä pitää käyttöliittymän tila osin tai kokonaan erillään komponenttikirjaston näkymäkomponenteista. Ensimmäinen ilmeinen tekijä on joustavuus – kun tilaa ei säilötä komponenttikirjaston sisällä, voidaan käyttää mitä vain haluttuja tietorakenteita ja menetelmiä tilan käsittelyyn. Tilaa voidaan säilöä esimerkiksi johonkin pysyvään muistiin myöhemmin käytettäväksi tai välittää verkkoyhteyden yli jollekin toiselle laitteelle käsiteltäväksi tai käytettäväksi. Näkymä on myös helppo asettaa vastaamaan jotain ulkopuolelta saatua tilaa. Inkrementaalisten komponenttikirjastojen tarjoamat tietorakenteet harvoin tarjoavat näin kattavia ominaisuuksia. Yleinen toiminnallisuus inkrementaalisissa komponenttikirjastoissa on niin sanottujen primitiivisten eli ei-muita-sisältävien arvojen asettaminen komponenttien arvoiksi tai muiksi attribuuteiksi, jotka esitetään jollain tavalla näkymässä visuaalisesti. Toinen yleinen toiminnallisuus on näkymärakenteen hallinta puumaisena tietorakenteena – komponentteja voidaan lisätä puun solmujen alle, jolloin ne näkyvät vastaavan komponentin sisällä näkymässä, tai niitä voidaan poistaa solmujen alta. Lisäksi komponenttipuista voidaan yleensä hakea tietoa, esimerkiksi iteroida tietyn solmun lapsisolmut läpi. Muiden tietorakenteiden esiintyminen on kuitenkin harvinaisempaa, sillä komponenttikirjastot eivät yleensä tarjoa mekanismeja, joilla esimerkiksi jonkinlaisia assosiaatiotaulukoita voitaisiin suoraan yhdistää näkymään niin, että taulukon päivittäminen päivittäisi samalla näkymää. Tämä on luonnollista, sillä assosiaatiotaulukolle ei puurakenteiden tavoin ole suoraviivaista tapaa muuntaa ne isomorfisesti näkymän visuaaliseksi esitykseksi. Jos assosiaatiotaulukkoa kuitenkin halutaan käyttää käyttöliittymän tilan käsittelyyn, joudutaan tietoa siis säilömään myös näkymäpuun ulkopuolelle. Tällöin käyttöliittymän tilaa kahdennetaan sekä assosiaatiotaulukoon että komponenttikirjaston sisäiseen tilaan, jos assosiaatiotaulukon arvot halutaan näyttää esimerkiksi listana näkymässä. Tämän kahdentamisen myötä syntyy tarve päivittää assosiaatiotaulukkoa aina, kun käyttöliittymässä tapahtuu tapahtuma, joka vaatii näytettävän listan päivittämistä, ja toisaalta tarve päivittää näyttöä aina, kun assosiaatiotaulukon sisältö muuttuu. Nämä päivitykset on ohjelmoijan huomattava määrittää jokaiseen paikkaan, jossa assosiaatiotaulukon arvoja tai siihen liittyvää näkymää muokataan. Suorassa määrittelyssä sen sijaan käyttöliittymän tilaa voidaan säilyttää vapaasti

vaikkapa assosiaatiotaulukossa, ja vasta näkymän määrityksessä voidaan lukea tätä tietorakennetta ja sitä kautta muodostaa hetkellinen näkymä senhetkisen ulkoisen tilan perusteella. Tällöin tilaa ei säilytetä kahdessa eri paikassa, vaan assosiaatiotaulukko toimii tässä suhteessa käyttöliittymän tilan ainoana totuudenlähteenä. Esimerkki tästä nähdään liitteen A listauksissa 12 ja 13. Niistä inkrementaalisessa versiossa tapahtumien yhteydessä joudutaan päivittämään paitsi assosiaatiotaulukkoa myös näkymää, kun taas suorassa versiossa assosiaatiotaulukon päivitys saa näkymän päivittymään annetun määrityksen mukaisesti. Lisäksi inkrementaalista versiota monimutkistaisi lisää, jos assosiaatiotaulukkoa voitaisiin päivittää myös jostain näkymän ulkopuolelta, jolloin vastaavat muutokset pitäisi synkronoida samalla näkymään.

Toinen haaste, joka käyttöliittymän kaiken tilan säilömisestä näkymän komponentteihin syntyy, on, että tila on sidottu näkymän visuaaliseen esitykseen. Esimerkiksi jos näkymässä esitetään jonkinlainen listaus eri tietueista, on tietueiden tiedot sijoitettava komponentteihin, jotka sijaitsevat näkymäpuussa toistensa rinnalla. Tällöin ei voida esimerkiksi helposti säilöä listaa esitysjärjestyksestä poikkeavassa järjestyksessä, kun sen tietueita halutaan lukea jossain kohtaa ohjelmakoodia. Samoin olisi vaikeaa jakaa lista useampaan erilliseen listaan, jotka sisältävät jollain tapaa toisistaan poikkeavia tietueita. Lopuksi olisi myös vaikea hyödyntää samaa tietuelistaa useassa eri paikassa, mikäli vastaavat tiedot haluttaisiin näyttää useassa eri paikassa näkymässä – samat tiedot saatetaan joutua manuaalisesti kahdentamaan useita kertoja näkymän tietorakenteisiin. Inkrementaaliset komponenttikirjastot voisivat tarjota toimintoja esimerkiksi siihen, että kaksi komponenttipuuta voitaisiin määrittää mukautumaan toisiinsa automaattisesti, tai että näkymän listauksille voitaisiin määritellä erilliset järjestykset esitystä ja tietojen lukemista varten. Tällaiset ratkaisut eivät kuitenkaan kata kovin joustavasti erilaisia tarpeita – esimerkiksi jos kahden komponenttipuun haluttaisiin näyttävän lähes samalta mutta pienin eroavaisuuksin, ei ensimmäinen ratkaisu enää riittäisikään. Samoin kahden erilaisen järjestyksen määrittäminen listaukselle ei onnistu suoraviivaisesti, jos tietoja lukiessa halutaan jakaa listauksen tietueet useampaan eri listaan. Yleensä inkrementaaliset komponenttikirjastot eivät tällaisia ratkaisuja tarjoakaan. Siispä jos käyttöliittymässä halutaan esittää dataa niin, että käsiteltävän datan rakenne poikkeaa näkymän visuaalisesta esityksestä, joudutaan dataa inkrementaalisessa komponenttikirjastossa säilömään komponenttipuun ulkoisiin rakenteisiin tai kahdentamaan useaan paikkaan komponenttipuussa.

Suoraa määrittelyä käyttäessä sen sijaan käytettyjen tietorakenteiden ei tarvitse olla mitenkään sidottuja näkymän visuaaliseen esitykseen – esimerkiksi jos jotain tiettyä tietoa halutaan esittää useammassa paikassa näkymässä, ei tuota juurikaan lisää kompleksisuutta käyttää samaa muuttujaa vastaavasti useammassa paikassa hetkellistä komponenttipuuta määritettäessä. Samoin hetkellistä näkymää muodostettaessa voidaan lukea minkä vaan muotoisia tietorakenteita ja muuntaa niiden sisältämää dataa haluttuun esitykseen komponenttipuussa. Usein tämä komponenttipuun muodostus tietorakenteiden perusteella voidaan tehdä suorituskykyisesti, esimerkiksi jos iteroidaan assosiaatiotaulukon arvot järjestyksestä välittämättä ja lisätään kukin vuorollaan näytölle piirrettävään komponenttipuuhun. Sen sijaan, jos halutaan esimerkiksi lajitella jonkin esitettävän listauksen alkiot eri järjestykseen kuin käytettävässä tietorakenteessa, voi lajittelu joissain komponenttikirjastoissa tapahtua jokaisella näytön päivityksellä ja siten huonontaa sovelluksen suorituskykyä. Tällöin myös suorassa määrittelyssä voidaan vaatia tiedon kahdentamista useaan tietorakenteeseen ja näiden rakenteiden synkronoituna pitämistä, kun niitä tarvitsee päivittää. Näkymän määrittely on tällöin kuitenkin edelleen erillään tilan käsittelystä, ja näkymän ja tietorakenteiden ei edelleenkään tarvitse olla suoraan sidoksissa toisiinsa.

Merkittävä inkrementaalisten komponenttikirjastojen haaste on siis, että joko kaikki käyttöliittymään vaikuttava tila joudutaan säilömään komponenttikirjaston komponentteihin, tai samaa tilaa joudutaan kahdentamaan komponenttikirjaston ja muiden tietorakenteiden välillä. Ensimmäisessä lähestymisessä on käyttöä rajoittavana tekijänä, että komponenttikirjaston ulkoisia tietorakenteita voidaan tarvita monissa eri tilanteissa, kuten suorituskyvyn optimoimiseksi tehokkaammilla tietorakenteilla tai tiedon säilömiseksi pysyvästi esimerkiksi tietokantaan. Lisäksi kaiken käyttöliittymän tilan säilöminen komponenttikirjaston rakenteisiin sitoo tietorakenteiden muodon mukailemaan näkymän visuaalista esitystä, mikä voi tehdä tilan käsittelystä monimutkaisempaa. Yksinkertaisemmissa sovelluksissa komponenttikirjaston tarjoamat tietorakenteet kuitenkin voivat tarjota riittävät toiminnallisuudet käyttöliittymän tilan käsittelyyn, jolloin inkrementaalinen komponenttikirjasto ei tältä osin lisää kompleksisuutta suoraan näkymämäärittelyyn verrattuna. Jos kuitenkin komponenttikirjaston ulkopuolisia tietorakenteita tarvitaan käsittelemään kaikkea tai osaa käyttöliittymän tilasta, joudutaan tietoa synkronoimaan näiden tietorakenteiden ja komponenttikirjaston komponenttien välillä, mikä vaatii manuaalista synkronoinnin määrittelyä ohjelmakoodissa. Tä-

mä lisää inkrementaalisen määrityksen kompleksisuutta verrattuna suoraan näkymämäärittelyyn. Tämä voi olla painava syy valita käyttöön suoraa näkymämäärittelyä käyttävä komponenttikirjasto, kun lähdetään rakentamaan käyttöliittymää, joka luultavasti inkrementaalista määrittelyä käyttäessä vaatisi tilan kahdentamista komponenttikirjaston ja ulkoisten tietorakenteiden välille. Usein ei-triviaaleja käyttöliittymiä kehitettäessä tämä tilanne voi tulla vastaan. Muitakin aiemmin mainittuja eroavaisuuksia määrittelytapojen välillä on, kuten se, esitetäänkö käyttöliittymän määritys yhdessä osassa kuten suorassa määrittelyssä, vai alkunäkymän ja siihen tehtävien mutaatioiden kautta kuten inkrementaalisessa määrittelyssä. Näitä eroja ei ole yhtä helppoa kuin käyttöliittymän tilan kahdentamisen tapauksessa katsoa suoraksi eduksi jomman kumman määrittelytavan puolesta, ja voi riippua sovelluksen ja sen kehittäjien tarpeista, mitkä ominaisuudet ovat suotuisia komponenttikirjaston kannalta. Suoralla määrittelyllä on kuitenkin tässäkin joitain melko objektiivisia etuja, kuten se, että suorassa määrittelyssä näkymän rakennetta ohjelman suorituksen aikana on helpompi analysoida staattisia menetelmiä käyttäen. Kaikkiaan suoran käyttöliittymämäärittelyn voidaan siis katsoa olevan joustavampi vaihtoehto kuin inkrementaalisen määrittelyn, sillä se voi tehdä ei-triviaalien käyttöliittymien tilanhallinnasta suoraviivaisempaa, ja yksinkertaisemmissa käyttöliittymissä määrittelytapojen välillä ei ole yhtä suuria eroja.

6 Pohdinta ja yhteenveto

6.1 Tutkimuksen taustat

Tämän tutkimuksen tavoite oli selvittää, mitä käsitteitä käyttöliittymän määrittelyyn liittyy. Erityisesti keskityttiin suoran ja inkrementaalisen määrittelyn käsitteisiin sekä siihen, mitä eroja näiden määrittelytapojen välillä on. Tutkimusmenetelmänä käytettiin käsiteanalyysiä, joka mielestäni oli onnistunut valinta vastaamaan tutkimuskysymyksiin. Käsiteanalyysin avulla voitiin aluksi kartoittaa käsitteitä, joita käyttökäsitteitä, joita graafisen käyttöliittymän määrittämiseen minimissään liittyy. Tämä oli välttämätöntä, jotta voitiin määrittää suoran ja inkrementaalisen määrittelyn käsitteet suhteessa muihin käyttöliittymämäärittelyn käsitteisiin ja erottaa ne käsitteistä, jotka ovat irrallisia suoran ja inkrementaalisen määrittelyn käytöstä. Esimerkiksi deklaratiivinen ja imperatiivinen määrittely ovat toinen keskeinen osa-alue, joka vaikuttaa käyttöliittymämääritelmän semantiikkaan, mutta niitä voidaan hyödyntää yhtälailla suoran tai inkrementaalisen määrittelyn rinnalla, kuten luvussa 3.2 todettiin. Deklaratiivisesti tai imperatiivisesti voidaan määrittää esimerkiksi inkrementaalisen komponenttikirjaston hetkellinen näkymä tai muutokset, joita näkymään tapahtumien yhteydessä tehdään. On kuitenkin perusteltua väittää, että valinnalla inkrementaalisen ja suoran määrittelyn välillä komponenttikirjastossa on merkittävämpi vaikutus kuin deklaratiivisen ja imperatiivisen lähestymisen hyödyntämisellä, sillä valinta muuttaa määrittelyn semantiikkaa siltä osin, säilötäänkö näkymän tilaan liittyvää tietoa enemmän komponenttikirjaston sisäisessä tilassa vai sovelluskoodin datarakenteissa. Tämän vuoksi tässä tutkimuksessa keskityttiin tähän näkymämäärittelyn osa-alueeseen. Ennen tutkimuksen aloittamista olin työskennellyt erilaisten inkrementaalista tai suoraa näkymämäärittelyä hyödyntävien komponenttikirjastojen parissa, esimerkkeinä näistä Java Swing ja React. Tämän kokemuksen perusteella osasin odottaa, että valinta suoran tai inkrementaalisen määrittelyn käytöllä on merkittävä käyttöliittymäohjelmointiin vaikuttava tekijä. Lisäksi ennakoin, että suoran määrittelyn eduksi löytyisi merkittävämpi määrä etuja kuin inkrementaalisen määrittelyn käyttöön.

Tätä tutkimusta varten tehdyn kirjallisuuskatsauksen aikana etsin aiempia tutkimuksia, jotka keskittyvät graafisten käyttöliittymien ohjelmallisen määrittelyn käsitteiden kartoittamiseen.

Kuten luvussa 4 todettiin, varhaiset komponenttikirjastot toimivat valtaosin inkrementaalista näkymämäärittelyä käyttämällä, ja usein niissä hetkelliset näkymät ja käyttöliittymä muutokset esitettiin imperatiivisilla käskyillä. Tältä ajanjaksolta en kuitenkaan löytänyt juurikaan akateemista työtä, jota olisi tehty selvittämään käyttöliittymien määrittelyn käsitteitä. Myöhemmin esimerkiksi Carlsson ja Hallgren esittivät luvussa 4.2 uudenlaisen mallin näkymän määrittelyyn Fudgets-kirjastollaan, joskaan eivät erityisesti keskittyneet kuvaamaan osia, joita näkymän abstraktiin määritelmään heidän mallissaan kuuluu. Sittemmin Courtney ja Hudak ovat esittäneet Fruit-kirjaston yhteydessä Yampa-mallin, jolla nuolia ja reaktiivista ohjelmointia käyttäen voidaan määrittää käyttöliittymiä hieman Fudgets-kirjaston tapaan suoraan määrittelytyyliin (Courtney ja Hudak 2004). Myös Czaplicki ja Chong esittelivät Elmin yhteydessä oman reaktiivisen mallinsa näkymien määrittämiseen (Czaplicki ja Chong 2013), ja Elm itsessään tarjosi hyvin johdonmukaisen mallin näkymän tilan hallintaan. IMGUI-paradigman yhteydessä käyttöliittymämäärittelyn tarvittavia käsitteitä on pohdittu, joskaan formaalia määrittelyä ei vaikuta juuri olevan tehty.

Suoran ja inkrementaalisen käyttöliittymämäärittelyn suhteen en löytänyt aiempaa akateemista kirjallisuutta. Käsitteet oli sen sijaan tunnistettu toisilla nimillä ohjelmointiyhteisön parissa, esimerkiksi IMGUI-paradigman kehittäjien keskuudessa termeillä välitön piirtäminen ja tilapohjainen piirtäminen. Myös muissa yhteyksissä oli kiinnitetty huomiota näkymämäärittelyn merkitykseen – esimerkiksi React-kirjasto rakennettiin web-selainten inkrementaalisen DOM-rajapinnan päälle, jotta web-kehityksessä voitaisiin käyttää suoraa näkymämäärittelyä. Kuitenkaan suuri osa näistä tahoista ei ole pyrkinyt määrittämään suoran ja inkrementaalisen määrittelyn käsitteitä tai formalismeja, jotka näkymämäärittelyyn tältä osin vaikuttavat. Merkittävin taho, jolla suoran ja inkrementaalisen määrittelyn käsitteiden määrittämiseen on tehty työtä, on siis uskoakseni IMGUI-yhteisö. Tällä suunnalla on myös tehty vertailua määrittelytapojen eduista (ks. “About the IMGUI paradigm · ocornut/imgui Wiki” 2023b; “Immediate Mode Model/View/Controller” 2023a). Välitön ja tilapohjainen piirtäminen termeinä eroavat tässä tutkimuksessa käytetyistä suoran ja inkrementaalisen määrittelyn käsitteistä, mutta ne kuvaavat samaa ilmiötä ja ovat myös päteviä termejä tähän tarkoitukseen, kuten luvussa 4.3 jo todettiin. Välitön ja tilapohjainen piirtäminen määritellään sen mukaan, missä käyttöliittymän tilaa säiltöään, mikä liittyy suoraan suoran ja inkrementaalisen määrittelyn valintaan – suorassa määrittelyssä komponenttikirjasto ei säilö mitään ti-

laa, joten se vain piirtää sovelluskoodin antaman määrityksen mukaisen näkymän, kun taas inkrementaalinen kirjasto käyttää piirtämiseen aiempaa tietoaan komponenttien tilasta yhdistettynä sovelluskoodin antamiin muutosohjeisiin. Tämän tutkimuksen käsitteet sen sijaan perustuvat kahteen eri tapaan mallintaa näkymä funktiona.

6.2 Tutkimuksen tulokset

Tämän tutkimuksen myötä esitettiin formaalit määritelmät suoralle ja inkrementaaliselle käyttöliittymämäärittelylle sekä analysoitiin niitä käsiteanalyysin keinoin ja erotettiin ne muista käyttöliittymämäärittelyn käsitteistä. Samalla määritettiin myös näiden kahden määrittelytavan näkökulmasta yksinkertainen abstrakti määritelmä graafiselle käyttöliittymälle kahden erilaisen funktiopohjaisen määritelmän kautta. Käyttöliittymille on esitetty muitakin formalismeja, kuten Fruit:n signaaleihin ja signaalifunktioihin perustuva malli, mutta tietääkseni mikään aiempi akateeminen tutkimus ei ole keskittynyt analysoimaan käyttöliittymien mallintamista yksinkertaisina funktioina tilasta hetkelliseen näkymään joko aiempaa näkymän tilaa hyödyntäen tai ilman sitä. Niinpä tämä tutkimus siis määrittää käsitteet, joiden avulla on mahdollista tutkia helpommin lisää suoran ja inkrementaalisen käyttöliittymämäärittelyn käyttöä ohjelmistokehityksessä. Kiinnostavia jatkokohteita olisivat esimerkiksi kvantitatiivinen vertailu eri määrittelytapoja hyödyntävän ohjelmakoodin ymmärrettävyydestä tai kvalitatiivien vertailu määrittelytapojen vahvuuksista eri käyttötilanteissa.

Tässä tutkimuksessa myös verrattiin suoraa ja inkrementaalista määrittelyä niiden ominaisuuksien suhteen. Määrittelytapoja on verrattu aiemmin akateemisen kirjallisuuden ulkopuolella erityisesti ImGui-yhteisössä, mutta tässä tutkimuksessa oli joitain uusia näkökulmia, joita aiemmassa keskustelussa ei ole huomioitu. Ensinnäkin suoran ja inkrementaalisen määrittelyn käsitteitä tutkittiin hieman laajemmin sen osalta, mitkä käsitteet suoraan ja inkrementaaliseen määrittelyyn linkittyvät tai eivät ole sidoksissa. Toinen kenties merkittävämpi lisänäkökulma on sen asian tunnistaminen, että käyttöliittymän tilan kahdentaminen ei ole inkrementaalisen määrittelyn väistämätön ominaisuus, vaan se voidaan välttää säilömällä näkymän piirtämiseen tarvittava tila komponenttikirjaston tietorakenteisiin. Tällöin ei kohdata tilan kahdentamisen ja synkronoimisen tarpeesta syntyviä ilmeisiä ongelmia, vaan inkrementaalinen näkymämäärittely voi olla jopa yksinkertaisempaa kuin suoran näkymämäärittelyn

käyttö. Sen sijaan tunnistettiin, että näkymän komponenttipuuta vastaavat tietorakenteet eivät kata kaikkia mahdollisia tietorakenteita, ja että tilan sitominen näkymän osiin voi hankaloittaa näkymän muodostamista esimerkiksi, jos samaa tilaa tarvitaan näkymän usean eri osan määrittelyyn. Lisäksi tilan käsittely näkymästä erillään ei yleensä onnistu inkrementaalisten komponenttikirjastojen kanssa yhtä helposti kuin suoran määrittelyn kautta toimivien komponenttikirjastojen kanssa. Tämän tutkimuksen perusteella voidaan siis todeta, että suora määrittely on inkrementaalista näkymämäärittelyä joustavampi ja monipuolisempi vaihtoehto, mutta voi riippua määritettävän käyttöliittymän tarpeista, kumpi määrittelytapa tuottaa suoraviivaisemman määritelmän pienemmällä määrällä ohjelmakoodia. Tätä olisikin mielenkiintoista verrata jatkotutkimuksella niin, että mahdollisesti jo käyttöliittymän ohjelmointia aloitettaessa voitaisiin tehdä valistunut päätös siitä, millaista komponenttikirjastoa tulisi suoran ja inkrementaalisen määrittelyn osalta suosia.

6.3 Tutkimuksen luotettavuus

Ensimmäinen haaste, jonka tämän tutkimuksen aikana havaitsin, on kirjoittajan ennakkoasenne aiheeseen. Tutkimusaiheen valintaan motivoiduin, kun opiskelussa ja työelämässä huomasin eroja erilaisten komponenttikirjastojen käytettävyydessä ohjelmoinnin kannalta, ja havaitsin, että suurimmat erot liittyivät siihen, miten komponenttikirjastot suhtautuivat näkymien määrittämiseen suoran tai inkrementaalisen mallin kannalta. Tämän jälkeen totesin, että suoraa määrittelyä käyttävät komponenttikirjastot tekivät yleensä vähänkään monimutkaisempia käyttöliittymiä ohjelmoitaessa työstä helpompaa ja selkeämpää. Tämän mielipiteen kautta lähdin tutkimaan aihetta tarkemmin, mutta asenteellisuuden tai virhepäätelmien välttämiseksi olen kuitenkin pyrkinyt tekemään käsitenalyysin mahdollisimman puolueettomasti ja objektiivisesti käsitteitä tarkastellen. Samoin myös suoran ja inkrementaalisen määrittelyn eroja verratessani pyrin löytämään kaikki mahdolliset merkittävät edut ja ongelmat kummastakin määrittelytavasta. Tutkimusta luettaessa kirjoittajan ennakkoasenteet on kuitenkin hyvä tiedostaa, jotta voidaan varautua mahdollisiin tahattomiin epäobjektiivisiin johtopäätöksiin.

Toinen ja tutkimuksen merkittävin haaste on uskoakseni siinä tehdyn käsiteanalyysin kattavuus – tuliko kirjoittaja varmasti huomioineeksi kaikki käsitteet, jotka linkittyvät suo-

raan ja inkrementaaliseen määrittelyyn? Onko lisäksi olemassa joitain muita samankaltaisia tapoja lähestyä käyttöliittymän määrittämistä funktiona syötearvoista hetkelliseen näkymään? Näihin kysymyksiin olen pyrkinyt vastaamaan tutkimuksessa, mutta onko lisäksi joitain muita kysymyksiä, joita ei ole huomioitu? Tästä tilanteesta on hyvä esimerkki ImGui-yhteisössä käyty keskustelu suoran ja inkrementaalisen määrittelyn eroista, joissa inkrementaalisen määrittelyn heikkoudeksi on oletettu, että tilan kahdentaminen ohjelmakoodin ja komponenttikirjaston välillä on välttämättömyys, mitä se ei kuitenkaan aina ole, kuten tässä tutkimuksessa on todettu. Tämä tutkimus siis onnistui löytämään uuden tekijän, jota aiemmassa keskustelussa ei oltu huomioitu. Samoin myös tämän tutkimuksen analyyseissä saattaa jäädä huomiotta merkittäviä tekijöitä, jotka vaikuttavat sen lopputuloksiin. Tätä on pyritty ehkäisemään pyrkimällä suorittamaan käsiteanalyysi, joka kattaa tutkimuskysymyksiin liittyvät käsitteet ja niiden suhteet toisiin käyttöliittymämäärittelyn käsitteisiin. Lisäksi on tehty kirjallisuuskatsaus, jonka kautta voitiin verrata esiteltyjä käsitteitä käytännön todellisuuteen ja siten vahvistaa niiden pätevyyttä. Siten tämän tutkimuksen johtopäätökset ovat uskoakseni päteviä, mutta niitä on mahdollista tarkentaa tai korjata myöhemmässä tutkimuksessa.

Lähteet

Adelsberger, Stephan, Anton Setzer ja Eric Walkingshaw. 2018. "Declarative GUIs: Simple, Consistent, and Verified". Teoksessa *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming.* PPDP '18. Frankfurt am Main, Germany: Association for Computing Machinery. ISBN: 9781450364416. https://doi.org/10.1145/3236950.3236962.

Carew, Herbert ja Meledath Damodaran. 2008. "X-Windows, GUI programming, and Microsoft Windows", https://doi.org/10.48009/2_iis_2008_551-559.

Carlsson, Magnus ja Thomas Hallgren. 1993. "Fudgets: A graphical user interface in a lazy functional language". Teoksessa *Proceedings of the conference on Functional programming languages and computer architecture,* 321–330. https://doi.org/10.1145/165180.165228.

Chalmers, David. 2015. "Constructing the World". *The Philosophical Review* 124, numero 3 (heinäkuu): 430–437. ISSN: 0031-8108. https://doi.org/10.1215/00318108-2895429. eprint: https://read.dukeupress.edu/the-philosophical-review/article-pdf/124/3/430/462753/430deRosset.pdf.

Chalmers, David J. 1996. *The conscious mind: In search of a fundamental theory.* Oxford university press. ISBN: 0195117891.

Cook, William R. 2009. "On Understanding Data Abstraction, Revisited". Teoksessa *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications,* 557–572. OOPSLA '09. Orlando, Florida, USA: Association for Computing Machinery. ISBN: 9781605587660. https://doi.org/10.1145/1640089.1640133.

Cook, William R., Walter Hill ja Peter S. Canning. 1989. "Inheritance is Not Subtyping". Teoksessa *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* 125–135. POPL '90. San Francisco, California, USA: Association for Computing Machinery. ISBN: 0897913434. https://doi.org/10.1145/96709.96721.

Courtney, Antony ja Conal Elliott. 2001. “Genuinely functional user interfaces”. Teoksessa *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop,* 41–69. https://www.haskell.org/haskell-symposium/2001/2001-62.pdf#page=47.

Courtney, Antony Alexander ja Paul Hudak. 2004. “Modeling User Interfaces in a Functional Language”. AAI3125177. Väitöskirja. https://doi.org/10.5555/1023359.

Czaplicki, Evan ja Stephen Chong. 2013. “Asynchronous Functional Reactive Programming for GUIs”. Teoksessa *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation,* 411–422. New York, NY, USA: ACM Press, kesäkuu. https://doi.org/10.1145/2499370.2462161.

Elliott, Conal ja Paul Hudak. 1997. “Functional Reactive Animation”. Teoksessa *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming,* 263–273. ICFP ’97. Amsterdam, The Netherlands: Association for Computing Machinery. ISBN: 0897919181. https://doi.org/10.1145/258948.258973.

Excoffier, Thierry. 2003. “Zero Memory Widgets, LIRIS Research Report 20030311”, https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=37be14c79afadb8ef5623de58e513911c3f8934d.

Fahland, Dirk, Daniel Lübke, Jan Mendling, Hajo Reijers, Barbara Weber, Matthias Weidlich ja Stefan Zugal. 2009. “Declarative versus Imperative Process Modeling Languages: The Issue of Understandability”. Teoksessa *Enterprise, Business-Process and Information Systems Modeling,* toimittanut Terry Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, Pnina Soffer ja Roland Ukor, 353–366. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-01862-6. https://doi.org/10.1007/978-3-642-01862-6_29.

Finne, Sigbjørn ja Simon Peyton Jones. 1995. “Composing haggis”. Teoksessa *Programming Paradigms in Graphics: Proceedings of the Eurographics Workshop in Maastricht,* 85–101. Springer. ISBN: 978-3-7091-9457-7. https://www.microsoft.com/en-us/research/wp-content/uploads/1995/09/composing-haggis.pdf.

Gettier, Edmund L. 1963. “Is justified true belief knowledge?” *Analysis* 23 (6): 121–123. https://doi.org/10.1093/analys/23.6.121.

Hanus, Michael ja Christof Kluß. 2009. “Declarative Programming of User Interfaces”. Teoksessa *Practical Aspects of Declarative Languages,* toimittanut Andy Gill ja Terrance Swift, 16–30. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-92995-6. https://doi.org/10.1007/978-3-540-92995-6_2.

Heller, Dan, Paula Ferguson ja David Brennan. 1994. “Motif Programming Manual, Vol 6A”, https://doi.org/0937175706.

Hughes, J. 1989. “Why Functional Programming Matters”. *The Computer Journal* 32, numero 2 (tammikuu): 98–107. ISSN: 0010-4620. https://doi.org/10.1093/comjnl/32.2.98. eprint: https://academic.oup.com/comjnl/article-pdf/32/2/98/1445644/320098.pdf.

Hughes, John. 2000. “Generalising monads to arrows”. *Science of Computer Programming* 37 (1): 67–111. ISSN: 0167-6423. https://doi.org/10.1016/S0167-6423(99)00023-4.

Jackson, Frank. 1998. *From metaphysics to ethics: A defence of conceptual analysis.* Oxford University Press. ISBN: 0191519030.

Jansen, Bernard J. 1998. “The graphical user interface”. *ACM SIGCHI Bulletin* 30, numero 2 (huhtikuu): 22–26. https://doi.org/10.1145/279044.279051.

Järvinen, Pertti. 2004. *Tutkimustyön metodeista.* [Uud. p.] Toimittanut Pertti Järvinen ja Annikki Järvinen. Tampere: Opinpajan kirja.

Kaijanaho, Antti-Juhani. 2017. “Concept Analysis in Programming Language Research: Done Well It is All Right”. Teoksessa *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software,* 246–259. Onward! 2017. Vancouver, BC, Canada: Association for Computing Machinery. ISBN: 9781450355308. https://doi.org/10.1145/3133850.3133868.

Koskela, Timo, Otso Kassinen, Erkki Harjula ja Mika Ylianttila. 2013. “P2P Group Management Systems: A Conceptual Analysis”. *ACM Comput. Surv.* (New York, NY, USA) 45, numero 2 (maaliskuu). ISSN: 0360-0300. https://doi.org/10.1145/2431211.2431219.

Lieberman, Henry. 1986. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems". Teoksessa *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications,* 214–223. OOPSLA '86. Portland, Oregon, USA: Association for Computing Machinery. ISBN: 0897912047. https://doi.org/10.1145/28697.28718.

Lumsden, Joanna. 2002. "Selecting the 'Invisible' User Interface Development Tool". Teoksessa *People and Computers XVI - Memorable Yet Invisible,* toimittanut Xristine Faulkner, Janet Finlay ja Françoise Détienne, 365–380. London: Springer London. ISBN: 978-1-4471-0105-5. https://doi.org/10.1007/978-1-4471-0105-5_22.

Margolis, Eric ja Stephen Laurence. 2020. "Concepts". Teoksessa *The Stanford Encyclopedia of Philosophy,* Summer 2019, toimittanut Edward N. Zalta. Metaphysics Research Lab, Stanford University, 11. kesäkuuta 2020. https://plato.stanford.edu/entries/concepts/.

Martinez, Wendy L. 2011. "Graphical user interfaces". *WIREs Computational Statistics* 3 (2): 119–133. https://doi.org/10.1002/wics.150. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.150.

Myers, Brad A. 1995. "User Interface Software Tools". *ACM Trans. Comput.-Hum. Interact.* (New York, NY, USA) 2, numero 1 (maaliskuu): 64–103. ISSN: 1073-0516. https://doi.org/10.1145/200968.200971.

Noble, Rob ja Colin Runciman. 1995. "Lazy functional components for graphical user interfaces". Väitöskirja, Citeseer. https://doi.org/10.1007/BFb0026828.

Overton, David. 1997. "Graphical User Interfaces for Declarative Languages", https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=56681a068f30e0e165979b596716c6ab201882d2.

Sage, Meurigd. 2000. "FranTk - a Declarative GUI Language for Haskell". *SIGPLAN Not.* (New York, NY, USA) 35, numero 9 (syyskuu): 106–117. ISSN: 0362-1340. https://doi.org/10.1145/357766.351250.

Salvaneschi, Guido, Sven Amann, Sebastian Proksch ja Mira Mezini. 2014. "An Empirical Study on Program Comprehension with Reactive Programming". Teoksessa *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering,* 564–575. FSE 2014. Hong Kong, China: Association for Computing Machinery. ISBN: 9781450330565. https://doi.org/10.1145/2635868.2635895.

Scheifler, Robert W. ja Jim Gettys. 1986. "The X Window System". *ACM Trans. Graph.* (New York, NY, USA) 5, numero 2 (huhtikuu): 79–109. ISSN: 0730-0301. https://doi.org/10.1145/22949.24053.

SCHROETER, LAURA. 2004. "The Limits of Conceptual Analysis". *Pacific Philosophical Quarterly* 85 (4): 425–453. https://doi.org/10.1111/j.1468-0114.2004.00209.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1468-0114.2004.00209.x.

Silva, Carlos Eduardo ja José Creissac Campos. 2012. "Can GUI Implementation Markup Languages Be Used for Modelling?" Teoksessa *Human-Centered Software Engineering,* toimittanut Marco Winckler, Peter Forbrig ja Regina Bernhaupt, 112–129. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-34347-6. https://doi.org/10.1007/978-3-642-34347-6_7.

Stefik, Andreas ja Stefan Hanenberg. 2014. "The Programming Language Wars: Questions and Responsibilities for the Programming Language Community". Teoksessa *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software,* 283–299. Onward! 2014. Portland, Oregon, USA: Association for Computing Machinery. ISBN: 9781450332101. https://doi.org/10.1145/2661136.2661156.

Turing, A. M. 1937. "On Computable Numbers, with an Application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society* s2-42 (1): 230–265. https://doi.org/10.1112/plms/s2-42.1.230. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230.

Valaer, L. A. ja R. G. Babb. 1997. "Choosing a user interface development tool". *IEEE Software* 14 (4): 29–39. https://doi.org/10.1109/52.595896.

Wand, Yair ja Richard Y Wang. 1996. “Anchoring data quality dimensions in ontological foundations”. *Communications of the ACM* 39 (11): 86–95. https://doi.org/10.1145/240455.240479.

Vullinghs, Ton, Daniel Tuijnman ja Wolfram Schulte. 1995. “Lightweight GUIs for functional programming”. Teoksessa *Programming Languages: Implementations, Logics and Programs,* toimittanut Manuel Hermenegildo ja S. Doaitse Swierstra, 341–356. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-45048-1. https://doi.org/10.1007/BFb0026829.

Vertaisarvioimattomat lähteet

"A Brief History of Mac OS X". 2023, 27. kesäkuuta 2023. https://web.archive.org/web/20120514135706/http://osxbook.com/book/bonus/ancient/whatismacosx/history.html.

"A First Motif Program". 2023, 20. toukokuuta 2023. https://users.cs.cf.ac.uk/Dave.Marshall/X_lecture/node5.html#SECTION00500000000000000000.

"A History of the GUI | Ars Technica". 2023, 9. kesäkuuta 2023. https://arstechnica.com/features/2005/05/gui/.

"About the IMGUI paradigm · ocornut/imgui Wiki". 2023a, 17. syyskuuta 2023. https://github.com/ocornut/imgui/wiki/About-the-IMGUI-paradigm.

"About the IMGUI paradigm · ocornut/imgui Wiki". 2023b, 17. syyskuuta 2023. https://archive.org/details/GDM_September_2005.

"Add interactivity to your Flutter app | Flutter". 2023, 17. syyskuuta 2023. https://docs.flutter.dev/ui/interactivity.

"angular/angular.js: AngularJS - HTML enhanced for web apps!" 2024, 4. helmikuuta 2024. https://github.com/angular/angular.js.

"AngularJS — Superheroic JavaScript MVW Framework". 2024, 4. helmikuuta 2024. https://angularjs.org/.

"AngularJS: Developer Guide: Introduction". 2024, 4. helmikuuta 2024. https://docs.angularjs.org/guide/introduction.

"Calculator Form | Qt Designer Manual". 2023, 9. heinäkuuta 2023. https://doc.qt.io/qt-6/qtdesigner-calculatorform-example.html.

"CalculatorApp". 2023, 27. kesäkuuta 2023. http://www.nextcomputers.org/NeXTfiles/Docs/NeXTStep/3.3/nd/DevTools/16_CalculatorApp/CalculatorApp.htmld/index.html.

"Common Desktop Environment - Wikipedia". 2023, 9. kesäkuuta 2023. https://en.wikipedia.org/wiki/Common_Desktop_Environment.

"Element: scrollTo() method - Web APIs | MDN". 2023, 28. joulukuuta 2023. https://developer.mozilla.org/en-US/docs/Web/API/Element/scrollTo.

"eslint-plugin-react/docs/rules/jsx-no-target-blank.md at master · jsx-eslint/eslint-plugin-react". 2023, 28. joulukuuta 2023. https://github.com/jsx-eslint/eslint-plugin-react/blob/ca162fdc5dc37f9f3447640a5a14a91daf73ea47/docs/rules/jsx-no-target-blank.md.

"Flutter - Build apps for any screen". 2023, 20. toukokuuta 2023. https://flutter.dev/.

"Graphical user interface (GUI) | Britannica". 2023, 9. kesäkuuta 2023. https://www.britannica.com/technology/graphical-user-interface.

"Gtk – 4.0: Getting Started with GTK". 2023, 1. heinäkuuta 2023. https://docs.gtk.org/gtk4/getting_started.html.

Harris, Rich. 2020. "Rich Harris - Rethinking reactivity", 14. heinäkuuta 2020. https://youtu.be/AdNJ3fydeao.

"HTML Standard". 2020, 14. heinäkuuta 2020. https://html.spec.whatwg.org/.

"Immediate Mode Model/View/Controller". 2023a, 17. syyskuuta 2023. http://www.johno.se/book/imgui.html.

"Immediate Mode Model/View/Controller". 2023b, 17. syyskuuta 2023. https://learn.microsoft.com/en-us/windows/win32/learnwin32/retained-mode-versus-immediate-mode.

"Immediate-Mode Graphical User Interfaces (2005)". 2023, 17. syyskuuta 2023. https://caseymuratori.com/blog_0001.

"Introducing the Smalltalk Zoo - CHM". 2023, 9. kesäkuuta 2023. https://computerhistory.org/blog/introducing-the-smalltalk-zoo-48-years-of-smalltalk-history-at-chm/.

"Introduction · An Introduction to Elm". 2023, 17. syyskuuta 2023. https://guide.elm-lang.org/.

"Introduction to the DOM". 2020, 13. heinäkuuta 2020. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.

"Introduction to the DOM - Web APIs | MDN". 2023, 1. heinäkuuta 2023. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.

"jordwalke/FaxJs: Fax Javascript Ui Framework". 2023, 17. syyskuuta 2023. https://github.com/jordwalke/FaxJs.

"Layouts in Flutter | Flutter". 2023, 20. toukokuuta 2023. https://docs.flutter.dev/ui/layout.

Levien, Raph. 2020. "Towards a unified theory of reactive UI", 18. heinäkuuta 2020. https://raphlinus.github.io/ui/druid/2019/11/22/reactive-ui.html.

Linzmayer, Owen W. 1999. *Apple confidential : the real story of Apple Computer, Inc.* San Francisco, Calif. : No Starch Press.

"Macintosh System 1: What Was Apple's Mac OS 1.0 Like?" 2023, 9. heinäkuuta 2023. https://www.howtogeek.com/732546/macintosh-system-1-what-was-apples-mac-os-1.0-like/.

"Motif (software) - Wikipedia". 2023, 9. kesäkuuta 2023. https://en.wikipedia.org/wiki/Motif_(software).

"Moxie: Incremental Declarative UI in Rust | Hacker News". 2020, 18. heinäkuuta 2020. https://news.ycombinator.com/item?id=21683054.

"Netscape and Sun announce JavaScript, the Open, Cross-platform Object Scripting Language for Enterprise Networks and the Internet". 2023, 1. heinäkuuta 2023. https://web.archive.org/web/20070916144913/https://wp.netscape.com/newsref/pr/newsrelease67.html.

"ocornut/imgui". 2023, 17. syyskuuta 2023. https://github.com/ocornut/imgui/blob/357f752bed2716651f5ae69cebd9cbf0bcccccbe/examples/example_glfw_vulkan/main.cpp.

"QLineEdit Class | Qt Widgets 5.15.13". 2023, 20. toukokuuta 2023. https://doc.qt.io/qt-5.15/qlineedit.html.

"Qt | Tools for Each Stage of Software Development Lifecycle". 2023, 20. toukokuuta 2023. https://www.qt.io/.

"Qt Quick States | Qt Quick 6.6.0". 2023, 17. syyskuuta 2023. https://doc.qt.io/qt-6/qtquick-statesanimations-states.html.

"Quick Start – React". 2023, 20. toukokuuta 2023. https://react.dev/learn.

"React". 2023, 20. toukokuuta 2023. https://react.dev/.

"React Native · Learn once, write anywhere". 2023, 17. syyskuuta 2023. https://reactnative.dev/.

"Reconciliation – React". 2023, 17. syyskuuta 2023. https://legacy.reactjs.org/docs/reconciliation.html.

"TextField class - material library - Dart API". 2023, 20. toukokuuta 2023. https://api.flutter.dev/flutter/material/TextField-class.html.

"The Elm Architecture · An Introduction to Elm". 2023, 17. syyskuuta 2023. https://guide.elm-lang.org/architecture/.

"The GTK Project - A free and open-source cross-platform widget toolkit". 2023, 20. toukokuuta 2023. https://www.gtk.org/.

"The Lisa: Apple's Most Influential Failure - CHM". 2023, 9. heinäkuuta 2023. https://computerhistory.org/blog/the-lisa-apples-most-influential-failure/.

"The Xerox Alto, Smalltalk, and rewriting a running GUI". 2023, 9. kesäkuuta 2023. http://www.righto.com/2017/10/the-xerox-alto-smalltalk-and-rewriting.html.

"Thinking in React – React". 2023, 28. joulukuuta 2023. https://react.dev/learn/thinking-in-react.

"Tk examples". 2023, 1. heinäkuuta 2023. https://wiki.tcl-lang.org/page/Tk+examples.

"TkDocs - ttk.Entry". 2023, 1. heinäkuuta 2023. https://tkdocs.com/pyref/ttk_entry.html.

"TkDocs Tutorial - A First (Real) Example". 2023, 1. heinäkuuta 2023. https://tkdocs.com/tutorial/firstexample.html.

"Trail: Creating a GUI With Swing (The Java™ Tutorials)". 2023, 20. toukokuuta 2023. https://docs.oracle.com/javase/tutorial/uiswing/.

"useRef – React". 2023, 28. joulukuuta 2023. https://react.dev/reference/react/useRef.

"Using data attributes - Learn web development | MDN". 2023, 28. joulukuuta 2023. https://developer.mozilla.org/en-US/docs/Learn/HTML/Howto/Use_data_attributes.

"Using the State Hook – React". 2023, 17. syyskuuta 2023. https://legacy.reactjs.org/docs/hooks-state.html.

Wadlow, Thomas A. 1981. "The Xerox Alto Computer". *Byte Magazine* 6 (9): 58–68.

"Virtual DOM is pure overhead (2018) | Hacker News". 2020, 18. heinäkuuta 2020. https://news.ycombinator.com/item?id=19950253.

"wxWidgets: Samples Overview". 2023, 1. heinäkuuta 2023. https://docs.wxwidgets.org/3.0/page_samples.html.

"XView - Wikipedia". 2023, 16. kesäkuuta 2023. https://en.wikipedia.org/wiki/XView.

"XView & OpenLook". 2023, 16. kesäkuuta 2023. http://www.martin-graefe.homepage.t-online.de/xview_en.html.

"XView, an OPEN LOOK Toolkit for X Window download | SourceForge.net". 2023, 16. kesäkuuta 2023. https://sourceforge.net/projects/xview/.

Liitteet

A Koodinäytteitä

```
const view = events => {
 let items = [];
 let inputText = "";

 for (let event of events) {
  if (event.type === "inputChange") {
   inputText = event.value;
  }
  else if (event.type === "addItem") {
   // Muistilistan alkioihin lisätään nappia painaessa tietue,
   // joka kuvaa näytölle piirrettävää komponenttia.
   items = [
    ...items,
    {component: "div", params: {}, children: inputText}
   ];
   inputText = "";
  }
 }

 return {component: "div", params: {}, children: [
  // Tapahtumien myötä lisätyt muistilistan alkiot
  // sijoitetaan komponenttipuuhun.
  ...items,
  {component: "input",
   params: {type: "text", value: inputText},
   eventBindings: {change: "inputChange"},
   children: []
  },
  {component: "button",
```

```
    params: {},
    eventBindings: {click: "addItem"},
    children: "Lisää"
   }
  ]};
};
```

Listaus 1. Muistilistanäkymän määrittely aiempien tapahtumien funktiona (JavaScript, kuvitteellinen komponenttikirjasto)

```
const initialState = {inputText: "", items: [], itemsAdded: 0};

const updateState = (state, event) => {
 if (event.type === "inputChange") {
  return {...state, inputText: event.value};
 }
 else if (event.type === "addItem") {
  return {
   items: [...state.items, state.inputText],
   inputText: "",
   itemsAdded: state.itemsAdded + 1
  };
 }
 return state;
};

const view = state => {
 const items = state.items.map(item =>
  ({component: "div", params: {}, children: item})
 );

 return {component: "div", params: {}, children: [
  ...items,
```

```
    {component: "input",
     params: {type: "text", value: state.inputText},
     eventBindings: {change: "inputChange"},
     children: []
    },
    {component: "button",
     params: {},
     eventBindings: {click: "addItem"},
     children: "Lisää"
    },
    {component: "div",
     params: {},
     children: `Lisätty yhteensä: ${state.itemsAdded}`
    }
  ]};
};
```

Listaus 2. Muistilistanäkymän suora määrittely (JavaScript, kuvitteellinen komponenttikirjasto)

```
const initialView = {component: "div", params: {}, children: [
  {component: "input",
   id: "itemInput",
   params: {type: "text", value: ""},
   eventBindings: {change: "inputChange"},
   children: []
  },
  {component: "button",
   id: "addButton",
   params: {},
   eventBindings: {click: "addItem"},
   children: "Lisää"
  },
```

```
 {component: "div",
  id: "addedCount",
  params: {},
  children: "Lisätty yhteensä: 0"
 }
]};

const initialState = {itemsAdded: 0};

const update = (view, state, event) => {
 if (event.type === "inputChange") {
  // "setParam"-funktio ottaa vastaan näkymän ja palauttaa siitä
  // muokatun version niin, että pyydetyllä tunnisteella eli id:llä
  // varustetun komponentin jokin parametri on muutettu uuteen arvoon.
  // Parametrit järjestyksessä: Komponentin id, komponentin muutettava
  // parametri, parametrin uusi arvo, muokattava näkymä.
  const newView = setParam("itemInput", "value", event.value, view);
  return [newView, state];
 }
 else if (event.type === "addItem") {
  const newState = {itemsAdded: state.itemsAdded + 1};

  let newView = view;
  // "addBefore" palauttaa näkymän, jossa pyydetyn komponentin eteen
  // on lisätty toinen komponentti.
  newView = addBefore(
   "addButton",
   {
    component: "div",
    params: {},
    children: getParam("itemInput", "value", newView)
   },
   newView
```

```
  );
  newView = setParam("itemInput", "value", "", newView);
  newView = replaceChildren(
   "addedCount",
   `Lisätty yhteensä: ${newState.itemsAdded}`,
   newView
  );

  return [newView, newState];
 }
 return [view, state];
};
```

Listaus 3. Muistilistanäkymän inkrementaalinen määrittely (JavaScript, kuvitteellinen komponenttikirjasto)

```
/* alpha_list.c -- insert items into a list.  */

#include <Xm/List.h>
#include <Xm/RowColumn.h>
#include <Xm/TextF.h>

main(argc, argv)
int argc;
char *argv[];
{
    Widget          toplevel, rowcol, list_w, text_w;
    XtAppContext    app;
    Arg             args[5];
    int             n = 0;
    void            add_item();

    XtSetLanguageProc (NULL, NULL, NULL);
```

```
    toplevel = XtVaAppInitialize (&app, "Demos", NULL, 0,
        &argc, argv, NULL, NULL);

    rowcol = XtVaCreateWidget ("rowcol",
        xmRowColumnWidgetClass, toplevel, NULL);

    XtSetArg (args[n], XmNvisibleItemCount, 5); n++;
    list_w = XmCreateScrolledList (rowcol, "scrolled_list", args, n);
    XtManageChild (list_w);

    text_w = XtVaCreateManagedWidget ("text",
        xmTextFieldWidgetClass, rowcol,
        XmNcolumns,     25,
        NULL);
    XtAddCallback (text_w, XmNactivateCallback, add_item, list_w);

    XtManageChild (rowcol);
    XtRealizeWidget (toplevel);
    XtAppMainLoop (app);
}

/* Add item to the list.
 * This is the callback routine for the TextField widget.
 */
void
add_item(text_w, client_data, call_data)
Widget text_w;
XtPointer client_data;
XtPointer call_data;
{
    Widget list_w = (Widget) client_data;
    char *text, *newtext = XmTextFieldGetString (text_w);
```

```
    XmString str;
    int u_bound;

    /* newtext is the text typed in the TextField widget */
    if (!newtext || !*newtext) {
        /* non-null strings must be entered */
        XtFree (newtext); /* XtFree() checks for NULL */
        return;
    }
    /* get the current entries (and number of entries) from the List */
    XtVaGetValues (list_w,
        XmNitemCount, &u_bound,
        NULL);
    str = XmStringCreateLocalized (newtext);
    XtFree (newtext);
    /* positions indexes start at 1 */
    XmListAddItemUnselected (list_w, str, u_bound);
    XmStringFree (str);
    XmTextFieldSetString (text_w, "");
}
```

Listaus 4. Motif-komponenttikirjastolla toteutetun listanäkymän lähdekoodi

```
<!doctype html>
<html ng-app="todoApp">
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/
1.8.2/angular.min.js"></script>
    <script src="todo.js"></script>
    <link rel="stylesheet" href="todo.css">
  </head>
  <body>
    <h2>Todo</h2>
```

```
    <div ng-controller="TodoListController as todoList">
      <span>
        {{todoList.remaining()}} of {{todoList.todos.length}} remaining
      </span>
      [ <a href="" ng-click="todoList.archive()">archive</a> ]
      <ul class="unstyled">
        <li ng-repeat="todo in todoList.todos">
          <label class="checkbox">
            <input type="checkbox" ng-model="todo.done">
            <span class="done-{{todo.done}}">{{todo.text}}</span>
          </label>
        </li>
      </ul>
      <form ng-submit="todoList.addTodo()">
        <input type="text" ng-model="todoList.todoText"  size="30"
               placeholder="add new todo here">
        <input class="btn-primary" type="submit" value="add">
      </form>
    </div>
  </body>
</html>
```

Listaus 5. AngularJS-ohjelmistokehyksellä toteutetun muistilistasovelluksen lähdekoodin HTML-osa (“AngularJS — Superheroic JavaScript MVW Framework” 2024)

```
angular.module('todoApp', [])
  .controller('TodoListController', function() {
    var todoList = this;
    todoList.todos = [
      {text:'learn AngularJS', done:true},
      {text:'build an AngularJS app', done:false}];

    todoList.addTodo = function() {
```

```
    todoList.todos.push({text:todoList.todoText, done:false});
    todoList.todoText = '';
  };

  todoList.remaining = function() {
    var count = 0;
    angular.forEach(todoList.todos, function(todo) {
      count += todo.done ? 0 : 1;
    });
    return count;
  };

  todoList.archive = function() {
    var oldTodos = todoList.todos;
    todoList.todos = [];
    angular.forEach(oldTodos, function(todo) {
      if (!todo.done) todoList.todos.push(todo);
    });
  };
});
```

Listaus 6. AngularJS-ohjelmistokehyksellä toteutetun muistilistasovelluksen lähdekoodin JavaScript-osa (“AngularJS — Superheroic JavaScript MVW Framework” 2024)

```
.done-true {
  text-decoration: line-through;
  color: grey;
}
```

Listaus 7. AngularJS-ohjelmistokehyksellä toteutetun muistilistasovelluksen lähdekoodin CSS-osa (“AngularJS — Superheroic JavaScript MVW Framework” 2024)

```
class CounterWidget extends StatefulWidget {
```

```
  const CounterWidget({super.key});

  @override
  State<CounterWidget> createState() => _CounterWidgetState();
}

class _CounterWidgetState extends State<CounterWidget> {
  bool _clickCount = 0;

  void _handleClick() {
    setState(() {
      _clickCount += 1;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Row(
      mainAxisSize: MainAxisSize.min,
      children: [
        Container(
          padding: const EdgeInsets.all(0),
          child: IconButton(
            padding: const EdgeInsets.all(0),
            alignment: Alignment.centerRight,
            icon: const Icon(Icons.star),
            color: Colors.red[500],
            onPressed: _handleClick,
          ),
        ),
        SizedBox(
          width: 18,
          child: SizedBox(
```

```
            child: Text('$_clickCount'),
          ),
        ),
      ],
    );
  }
```

Listaus 8. Laskurinäkymä Flutter-kirjastolla (Dart) (mukaillen "Add interactivity to your Flutter app | Flutter" 2023)

```
const renderTree = () => {
 let tree
 if (checkbox.checked) {
  tree = document.createElement("div")
  tree.textContent = "Puu, jos valittu"
 }
 else {
  tree = document.createElement("div")
  tree.textContent = "Puu, jos ei valittu"
 }
 treeContainer.replaceChildren(tree)
}

const root = document.createElement("div")
const checkbox = document.createElement("input")
root.appendChild(checkbox)
checkbox.setAttribute("type", "checkbox")
// Puun sisältöä päivitetään valintanapin painallusten yhteydessä.
checkbox.addEventListener("click", renderTree)
const treeContainer = document.createElement("div")
root.appendChild(treeContainer)
renderTree()
```

Listaus 9. Inkrementaalinen näkymämääritys, jossa valintasyötekentän arvo vaikuttaa muun näkymän piirtämiseen (JavaScript, DOM)

```
const Component = () => {
 const [checked, setChecked] = useState(false)

 // Puun sisältö päätetään tilan perusteella piirrettäessä.
 let tree
 if (checked) {
  tree = <div>Puu, jos valittu</div>
 }
 else {
  tree <div>Puu, jos ei valittu</div>
 }

 return <div>
  <input
   type="checkbox"
   checked={checked}
   onClick={() => setChecked(!checked)}
  />
  <div>
   {tree}
  </div>
 </div>
}
```

Listaus 10. Suora näkymämääritys, jossa valintasyötekentän arvo vaikuttaa muun näkymän piirtämiseen (JavaScript, React)

```
let checkboxChecked = false
```

```
const root = document.createElement("div")
const checkbox = document.createElement("input")
root.appendChild(checkbox)
checkbox.setAttribute("type", "checkbox")
checkbox.addEventListener("click", () => {
 checkboxChecked = !checkboxChecked
})

const routine = () => {
 if (checkboxChecked) {
  alert("Valittu!")
 }
}

setInterval(routine, 60_000)
```

Listaus 11. Inkrementaalinen näkymämääritys, jossa ajastettu rutiini lukee näkymän tilasta kahdennettua muuttujaa (JavaScript, DOM)

```
const associationTable = {
 "kissa": 5,
 "koira": 4,
 "norsu": 10
}

const root = document.createElement("div")
const title = document.createElement("div")
root.appendChild(title)
title.textContent = "Lista eläimistä ja niiden määristä"
const animalsContainer = document.createElement("div")
root.appendChild(animalsContainer)

// Alkunäkymän muodostaminen assosiaatiotaulukon datan perusteella
```

```
// ja näkymää päivittävien takaisinkutsufunktioiden kytkentä
for (const key of Object.keys(associationTable)) {
 const animalContainer = document.createElement("div")
 animalsContainer.appendChild(animalContainer)
 const animalText = document.createElement("span")
 animalContainer.appendChild(animalText)
 animalText.textContent = `${key}:`
 const amountText = document.createElement("span")
 animalContainer.appendChild(amountText)
 amountText.textContent = associationTable[key]
 const button = document.createElement("button")
 animalContainer.appendChild(button)
 button.textContent = "Lisää"
 button.addEventListener("click", () => {
  // Assosiaatiotaulukon arvon päivittäminen vastaamaan uutta arvoa
  // sekä näkymän tilan päivittäminen samalla vastaamaan
  // assosiaatiotaulukkoa.
  associationTable[key] = associationTable[key] + 1
  amountText.textContent = associationTable[key]
 })
}
```

Listaus 12. Inkrementaalinen näkymämääritys, jossa dynaaminen assosiaatiotaulukko piirretään näkymään (JavaScript, DOM)

```
const Component = () => {
 // Assosiaatiotaulukon määrittäminen näkymän ulkoisena
 // muokattavana tilana
 const [associationTable, setAssociationtTable] = useState({
  "kissa": 5,
  "koira": 4,
  "norsu": 10
 })
```

```
const tableKeys = Object.keys(associationTable)

const animals = tableKeys.map(key => (
  <div>
    <span>{key}:</span>
    <span>{associationTable[key]}</span>
    <button
      onClick={() => {
        // Assosiaatiotaulukon päivittäminen,
        // joka laukaisee myös näkymän päivittymisen
        setAssociationtTable({
          ...associationTable,
          [key]: associationTable[key] + 1
        })
      }
    >
      Lisää
    </button>
  </span>
))

return <div>
  <div>Lista eläimistä ja niiden määristä</div>
  <div>
    {animals}
  </div>
</div>
}
```

Listaus 13. Suora näkymämääritys, jossa dynaaminen assosiaatiotaulukko piirretään näkymään (JavaScript, React)