Henna-Riikka Kari

Graafisen käyttöliittymän tyyliopas ja sen noudattamisen ohjelmallinen valvonta

Tietotekniikan pro gradu -tutkielma

21. marraskuuta 2024

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Henna-Riikka Kari

Yhteystiedot: henna.r.kari@student.jyu.fi

Ohjaaja: Tommi Mikkonen

Työn nimi: Graafisen käyttöliittymän tyyliopas ja sen noudattamisen ohjelmallinen valvonta

Title in English: Graphical UI style guide and programmatic monitoring of its compliance

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 101+0

Tiivistelmä: Toimiva ja tyylikäs graafinen käyttöliittymä on tärkeä osa ohjelmistokehitystä ja vaikuttaa merkittävästi käyttäjäkokemukseen ja sen myötä tuotteen laatuun ja kaupalliseen menestykseen. Tehokas käyttöliittymäsuunnittelu puolestaan nojaa toimivaan komponenttikirjastoon ja sen uudelleen käytettäviin, keskenään vuorovaikuttaviin komponentteihin. Kun organisaatiossa useat kehittäjät luovat uusia komponentteja, lopputulos on usein epäyhtenäinen ja kehittäjänsä näköinen. Selkeällä ja ohjelmallisesti testattavalla tyyliohjeistuksella voidaan käyttöliittymien ilmeeseen saada kaivattua yhtenäisyyttä, toimivuutta ja laatua.

Tämän suunnittelututkimuksen tuloksena toteutettiin kohdeorganisaatiolle käyttöliittymän tyyliopas ja malli sen visuaalisista regressiotesteistä. Kohdeorganisaatio on keskisuomalainen ohjelmistoalan yritys, joka tuottaa SaaS-palveluja lähinnä energiatoimialan tarpeisiin. Tyyliopas toteutettiin olemassa olevan komponenttikirjaston yhteyteen osaksi suurempaa kokonaisuutta. Se sisälsi joukon organisaatiospesifejä uudelleenkäytettäviä graafisia komponentteja, sekä ohjeet typografiasta, ikoneista, väreistä ja sivumalleista. Tyylinvalvojaksi valittiin Vitest image snapshots, joka soveltuu niin yksittäisten komponenttien kuin kokonaisten sivunäkymien snapshot-testaukseen ja integroituu helposti käytössä olevaan Vite-ympäristöön, jossa toiminnallinen testaus suoritetaan jo Vitest:illä. Tutkimus sisälsi myös ohjeet komponenttikirjaston rekonstruktiolle niin, että tyylioppaasta saadaan elävä ja ylläpidettävä. Tutkimus antaa arvokasta informaatiota ja toimivia esimerkkejä kohdeorganisaatiolle ja toimii mallina myös muille.

Avainsanat: pro gradu -tutkielmat, käyttöliittymän tyyliopas, käyttöliittymän visuaalinen testaus, komponenttikirjasto, suunnittelujärjestelmä

Abstract: A functional and elegant graphical user interface is an important part of software development and has a significant impact on user experience and impacts the quality and commercial success of the product. Effective user interface design, in turn, relies on a functional component library and its reusable, mutually interacting components. When several developers in an organization create new components, the result is often inconsistent and looks like its developer. With clear and programmatically testable style guidelines, the desired consistency, functionality and quality can be achieved in the appearance of user interfaces.

As a result of this design science research, a user interface style guide and a model of its visual regression tests were implemented for the target organization. The target organization is a central Finnish software company that produces SaaS services mainly for the needs of the energy sector. The style guide was implemented in connection with an existing component library as part of a larger whole. It included a set of organization-specific reusable graphic components, as well as instructions on typography, icons, colors and page templates. Vitest image snapshots was chosen as the tool to control the usage of the style guide, which is suitable for snapshot testing of both individual components and entire page views and easily integrates into the existing Vite environment, where functional testing is already performed with Vitest. The study also included instructions for reconstructing the component library so that the style guide is alive and maintainable. This research provides valuable information and working examples for the target organization and also serves as a model for others.

Keywords: Master's Theses, GUI Style Guide, GUI Visual Testing, Component Library, GUI Design System

Koodilohkot

Kuviot

Taulukot

Sisällys

1 Johdanto

Graafinen käyttöliittymä (GUI) on digitaalinen käyttöliittymä, jossa käyttäjä vuorovaikuttaa erilaisten graafisten komponenttien kanssa. Tällaisia ovat esimerkiksi kuvakkeet, painikkeet, valikot ja erilaiset lomakkeet. Graafisessa käyttöliittymässä olevat visuaaliset elementit tarjoavat käyttäjälle informaatiota ja mahdollistavat toimintoja, joita hän voi tehdä esimerkiksi lähettääkseen tietoa eteenpäin.(HubSpot 2024) Toimiva ja tyylikäs käyttöliittymä on tärkeä osa ohjelmistokehitystä, vaikuttaa merkittävästi käyttäjäkokemukseen ja tukee ratkaisujen kaupallista menestystä.

Graafisten käyttöliittymien suunnittelussa on tasapainoteltava mm. helppokäyttöisyyden, toiminnallisuuden, turvallisuuden ja houkuttelevan ulkoasun välillä. Käyttäjät vaativat yhä dynaamisempia käyttöliittymiä ja odottavat niiden toimivan monenlaisilla laitteilla (Lynch ja Horton 2016). Vaaditaan opittavuutta, johdonmukaisuutta ja saavutettavuutta (HEAVY.AI 2024). Näihin seikkoihin voidaan vaikuttaa käyttöliittymän graafisilla komponenteilla. Suunnittelun tehokkuutta voidaan parantaa tekemällä komponenteista uudelleenkäytettäviä ja järjestämällä ne komponenttikirjastoon.

Kohdeorganisaatio Akamon Innovations Oy kehittää ohjelmistotuotteita, joita käyttävät useat asiakkaat. Tuotteet räätälöidään asiakkaan tarpeiden mukaan komponenteista koostetuista tuoterungoista. Asiakastoteutusten toisteisuutta vähentämään on perustettu käyttöliittymäkomponenttien kirjasto, jonka uudelleenkäytettäviä komponentteja toteuttavat useat kehittäjät. Käyttöliittymien yhtenäinen ulkoasu on kuitenkin tämän myötä kärsinyt. On luotava ohjeistus, jonka pohjalta jokainen kehittäjä voi saada aikaan yhtiön ilmeen mukaisia tuotteita. Pelkkä ohjeistus ei kuitenkaan riitä, vaan komponenttien ja niistä koostettujen käyttöliittymien vastaavuus tyylioppaaseen on tarkastettava ohjelmallisesti. Tutkimusongelma on näin ollen olemassa.

Tässä tutkimuksessa suunniteltiin ja toteutettiin kohdeorganisaatiolle heidän tarkoitustaan vastaava tyyliohjeistus, joka liitettiin olennaiseksi osaksi olemassa olevaa komponenttikirjastoa. Laadittiin ohjeet kirjaston rakenteen rekonstruoimiseksi niin, että kaikki komponenttien tyylimääritykset löytyvät yhdestä paikasta, jolloin niiden ylläpito helpottuu merkittävästi. Tyylien valvontaa varten rakennettiin malli käyttöliittymän visuaalisista regressiotesteistä.

Luvussa 2 esitellään tutkimuksen viitekehys. Puhutaan komponenttilähtöisestä kehityksestä, komponenttien uudelleenkäytöstä ja niiden järjestämisestä kirjastoon. Esitellään käsitteet käyt-

töliittymän tyyliopas, suunnittelujärjestelmä ja tyylinvalvonta, mitä ne pitävät sisällään, kuinka ne liittyvät toisiinsa ja miten niitä voi käyttää. Luvussa 3 johdetaan tutkimusongelman perusteella varsinaiset tutkimuskysymykset, esitellään tutkimusmenetelmä ja sen käyttö tässä tutkimuksessa. Luvussa 4 esitellään tutkimuksen keskiössä olevan työkalun suunnittelu ja toteutus. Työ toteutettiin tutkimusmenetelmän mukaisesti sykleittäin, tutkittiin kohdeorganisaation nykytilaa ja toimintatapoja, muodostettiin tavoitetila vaatimusmäärittelyn avulla ja iteroitiin suunnittelusyklejä, kunnes työkalun todettiin olevan hyväksyttävissä. Luvussa 5 raportoidaan menetelmän mukaisten evaluointien tulokset ja arvioidaan tutkimuksen onnistumista. Luvussa 6 vedetään yhteen tutkimuksen keskeinen sisältö, pohditaan sen vaikutuksia kohdeorganisaatiolle ja esitetään jatkotutkimustarpeita.

2 Tutkimuksen viitekehys ja määritelmät

Viitekehys muodostuu kohdeorganisaation ja käytettävän järjestelmän kontekstista. Toimitaan komponenttilähtöisen ohjelmistokehityksen parissa, jossa komponenttikirjastoa hyödyntämällä toteutetaan asiakkaan toiveiden mukaisia käyttöliittymiä. Asiakastoteutukset eivät kuitenkaan ole täysin mielivaltaisia, vaan muodostuvat modulaarisista tuoterungoista. Graafisen käyttöliittymän yhtenäinen ilme tulisi perustua tyyliohjeistukseen, jota jokainen kehittäjä noudattaa.

Tässä luvussa esitellään keskeiset tähän ympäristöön liittyvät määritelmät. Tutkimusongelman perusteella on syytä puhua myös komponenttien ja niistä koottavien graafisten käyttöliittymien visuaalisesta testaamisesta sekä kirjaston ja tyylioppaan ympärille muodostuvasta suunnittelujärjestelmästä. Tässä tutkimuksessa, puhuttaessa käyttöliittymästä, tarkoitetaan aina graafista käyttöliittymää, ellei toisin mainita.

2.1 Komponenttilähtöinen kehitys ja koodin uudelleenkäyttö

Tänä päivänä käyttöliittymien odotetaan olevan houkuttelevia, dynaamisia ja toimivan moitteetta kaikilla laitteilla. Niiltä odotetaan yhä enemmän logiikkaa ja korkeampaa laatua. Sovelluksen koon kasvaessa käyttöliittymät käyvät raskaaksi ja niiden ylläpito vaikeutuu. Jakamalla käyttöliittymä pienempiin modulaarisiin osiin saadaan aikaan hallittavissa olevia joustavia ratkaisuja (ComponentDriven, n.d.).

Tähän perustuu idea komponenttilähtöisestä kehityksestä, joka on laajalti käytetty ja edelleen yleistyvä tapa ohjelmistokehityksessä (Qureshi ja Hussain 2008). Käyttöliittymät rakennetaan alhaalta ylös, alkaen pienistä osista, peruskomponenteista, yhdistellen niitä lopulta kokonaisiksi näkymiksi (Coleman 2017).

Idea komponenttien uudelleenkäytöstä ei ole uusi, sen kehitti jo yli neljäkymmentä vuotta sitten Douglas McIlroy (1968), joka sai ajatuksensa kaupallisten komponenttien tuotannosta muilta tekniikan aloilta. Menetelmän taustalla on liiketoiminnallisia ja teknisiä tavoitteita: tehokkuuden ja tuottavuuden kasvu, kustannussäästöt, nopea toimitus ja korkeampi laatu (Vale ym. 2016).

Menetelmän peruspilarit, komponentit, ovat itsenäisiä ohjelmayksiköitä, joita yhdistelemällä saadaan aikaan suurempia kokonaisuuksia. Komponentit voivat olla käyttöliittymä- tai lo-

giikkakomponentteja ja toimivat kuten rakennuspalikat. Komponentit vuorovaikuttavat toisten komponenttien kanssa rajapintojensa kautta. (ComponentDriven, n.d.).

Komponenttilähtöinen kehitys tuo mukanaan kiistattomia hyötyjä (Saring 2019; ComponentDriven, n.d.; Qureshi ja Hussain 2008; Osmani, n.d.). Ehkä tärkeimpänä hyötynä nähdään koodin uudelleenkäytettävyys, joka säästää aikaa ja parantaa näin tehokkuutta. Uusia komponentteja voidaan rakentaa laajentamalla olemassa olevia. Komponentteja uudelleen käyttämällä tuotteisiin saadaan johdonmukaisuutta ja niiden ilmeeseen yhtenäisyyttä.

Komponenttien ansiosta kehitystyö voidaan jakaa osiin ja tiimit voivat itsenäisesti kehittää, omistaa ja toimittaa uusia komponentteja. Tämä edelleen parantaa tehokkuutta (ComponentDriven, n.d.).

Jokaisella komponentilla on oma vastuualueensa ja se käsittelee tiettyä ongelmaa, eikä sen tarvitse tietää muiden komponenttien yksityiskohtia. Tämän myötä myös suurempien järjestelmien ymmärtäminen helpottuu (Osmani, n.d.). Uusia kehittäjiä on helpompi perehdyttää järjestelmään, joka koostuu pienemmistä osista (Saring 2019).

Komponenteista rakennettu käyttöliittymä on modulaarinen, ylläpidettävä ja skaalautuva. Virheiden löytäminen helpottuu, koska yksittäisten komponenttien testaaminen on helpompaa kuin kokonaisten sovellusten (Saring 2019).

Ohjelmistojen koon kasvaessa komponenttien määrä voi kasvaa hallitsemattoman suureksi. Tarkoitukseen sopivan komponentin löytäminen voi olla hankalaa ja dokumentaatio puutteellista, versionhallinta vaikeaa ja koodin jäljitettävyys heikkoa (Kalia ja Sood 2017). Tämän vuoksi komponentit kannattaa järjestää komponenttikirjastoon.

2.2 Graafiset komponenttikirjastot

Graafiset komponenttikirjastot ovat uudelleenkäytettävien graafisten komponenttien kokoelmia, joissa komponentit on järjestetty niin, että ne on helppo löytää ja ottaa käyttöön. Komponentin visuaalisen esityksen lisäksi jokaisesta komponentista tulisi olla seuraavat tiedot: komponentin uniikki nimi, kuvaus komponentin toiminnasta ja siitä mihin sitä yleensä käytetään, ominaisuudet, muuttujat ja tieto siitä, kuinka komponenttia voidaan erikoistaa, komponentin tila oletusasetuksineen, lähdekoodi sekä käyttöönotto-ohjeet (Fessenden 2021). Tällä tavoin oikean komponentin etsintä ja valinta sujuu tehokkaasti.

Graafinen komponenttikirjasto voi yksittäisten komponenttien lisäksi sisältää myös suunnittelumalleja (engl. *design pattern*), joilla voidaan komponenteista koostaa ryhmittelyjä tai vaikka kokonaisia näkymiä. Nämä mallit ovat komponenttien tapaan uudelleenkäytettäviä (Fessenden 2021). Komponenttikirjastoa, joka sisältää laajemman joukon yhteen koottuja komponentteja kutsutaankin usein mallikirjastoksi (engl. *pattern library*) (Friedman 2016).

Voidaan puhua kahdenlaisista graafisista komponenttikirjastoista: yrityksen omaan tarkoitukseen perustetuista kirjastoista tai avoimen lähdekoodin kirjastoista. Jälkimmäisellä tarkoitetaan joukkoa valmiita, testattuja, dokumentoituja ja muokattavissa olevia komponentteja, joilla käyttöliittymään saadaan pienellä vaivalla rakennetta ja ilmeikkyyttä (UPXin 2023). Tämän tutkimuksen viitekehykseen kuuluvat molemmat, sillä kohdeorganisaation omat komponentit pohjautuvat avoimen lähdekoodin React UI -kirjastoon, Chakra UI. Muita markkinoilla paljon käytettyjä avoimen lähdekoodin graafisia React UI -komponenttikirjastoja ovat mm. MUI (ent. Material UI), React Bootstrap ja AntDesign. Nämä ovat kokoelmia Reactin päälle rakennetuista, käyttövalmiista UI-komponenteista.

2.3 Käyttöliittymän tyyliopas

Internetin käyttäjät ovat yhä vaativampia nykyään ja haluttomampia kompromisseihin kuin ennen. Käyttöliittymän täytyy olla paitsi saavutettava ja käytettävä myös houkutteleva. Sen pitää olla helppokäyttöinen, selkeä, opittava, tehokas ja suorittaa käyttäjän haluama toiminta monenlaisilla laitteilla. Myös mahdolliset käyttäjän fysiologiset rajoitteet on huomioitava (Lynch ja Horton 2016). Näihin seikkoihin voidaan vaikuttaa mm. käyttöliittymäkomponenttien yhtenäisellä ilmeellä, värien ja typografioiden huolellisella suunnittelulla sekä elementtien asettelulla.

Kun käyttöliittymiä rakennetaan itsenäisistä komponenteista ja kehitystyö on organisaation sisällä hajautettua, millä voidaan varmistaa, että yhtenäinen ilme säilyy ja käyttökokemus pysyy miellyttävänä? Erityisesti tällaisessa ympäristössä yhtenäisellä ohjeistuksella on ratkaiseva merkitys.

Käyttöliittymän tyyliopas on kokoelma ohjeita yrityksen visuaalisesta ilmeestä. Siihen kuuluu yrityksen brändistrategiaan liittyvien ohjeiden lisäksi logoon, välistyksiin, väreihin, typografiaan, kuviin ja asetteluihin liittyvät ohjeistukset (Neville 2010). Tässä tutkimuksessa keskitytään käyttöliittymän tyylioppaaseen (engl. *UI style guide*), johon yllä mainittujen lisäksi kuuluvat kaikki verkkosovellusten käyttöön vaadittavat komponentit, kuten painikkeet, lomakkeet

ja ikonit sekä sivujen suunnittelumallit.

Tärkein tyylioppaan tuoma hyöty on käyttöliittymän johdonmukainen ja yhtenäinen ilme, joka luo asiakkaalle vaikutelman ammattimaisuudesta. Tyylioppaan laatiminen pakottaa hiomaan jokaisen visuaalisen yksityiskohdan tarkalleen, mikä ei usein ole kehittäjän näkökulmasta tärkeää. Aikaa säästyy oleelliseen, kun kehittäjän ei tarvitse miettiä värejä tai fontteja saadakseen uuden komponentin tuotantoon (Delong 2020). Tyylioppaan myötä syntyy jaettu sanasto organisaation kesken, esimerkiksi siitä, millä nimillä komponentteja tai malleja kutsutaan. Kommunikointi helpottuu ja syntyy vähemmän väärinymmärryksiä. Uusien jäsenten liittäminen projektiin helpottuu myös, kun voidaan viitata yhteiseen ohjeistukseen (Frost 2016).

Staattinen ja elävä tyyliopas. Pelkkä tyylioppaan laatiminen ei tietysti takaa onnistunutta lopputulosta. Sen integroiminen käytännön työhön voi olla haastavaa. Tyyliopas voi olla staattinen tai elävä (Feather 2014). Staattinen tyyliopas on erillinen dokumentaatio, ohjeistus komponenttien ulkoasulle ja niiden käytölle. Tällainen tyyliopas toimii hyvin yleisenä ohjeena ja keskustelun välineenä, mutta sen pysymisestä ajan tasalla ei ole takeita. Kun käyttöliittymät elävät ja komponentteja on tarvetta muuttaa tai lisätä, pitäisi staattiseen tyyliohjeeseen tehdä samat muutokset, muutoin se menettää merkityksensä. Tämä luonnollisesti aiheuttaa ylimääräisen työvaiheen eikä tavoiteltua tehokkuutta saavuteta.

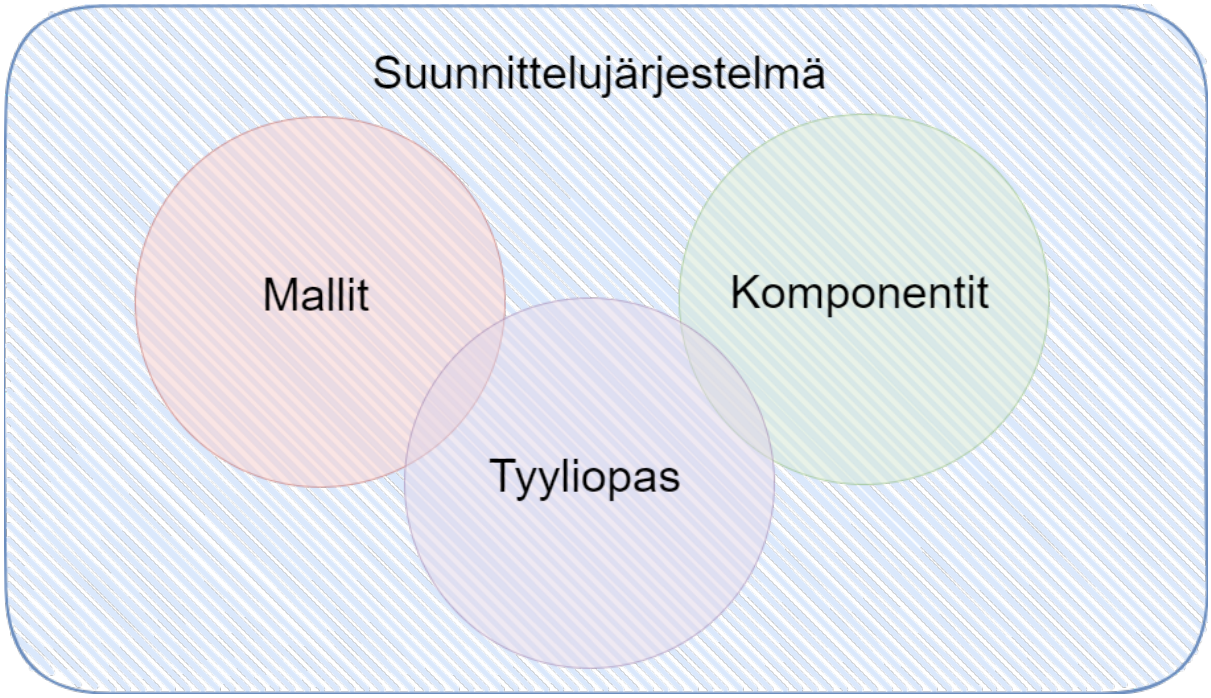
Elävä tyyliopas toimii hieman toisella tavalla. Siinä käyttöliittymäkoodi jaetaan tyylikirjaston ja varsinaisen palvelun toteutuksen kesken. Verkkosovelluksen toteutus käyttää siis suoraan tyylikirjaston komponentteja tyylimäärittelyineen. Näin ollen aina, kun muutoksia tarvitaan, ne tehdään tyylioppaan komponenttikirjastoon, josta ne päätyvät toteutuksiin. Vaihtoehtoisesti voidaan käyttää samaa tyylitiedostoa sekä tyyliohjeen että sen perusteella toteutetun verkkosivun komponenteille. Elävä tyyliopas pysyy tällä lailla paremmin ajan tasalla kuin staattinen tyyliopas. Ajan saatossa tyyliopasta kuuluukin muokata, laajentaa ja suunnitella jatkuvasti, jolloin sen hyödyt ylettyvät pitkälle tulevaisuuteen (Frost 2016).

Tämän tutkimuksen puitteissa toteutetaan nimenomaan elävä tyyliopas, jotta mahdollisuudet sen tehokkaalle, aktiiviselle ja jatkuvalle käytölle ovat olemassa.

2.4 Suunnittelujärjestelmä

Tyyliopas, mallikirjasto ja komponenttikirjasto dokumentointeineen yhdessä muodostavat suunnittelujärjestelmän (engl. *design system*). Suunnittelujärjestelmä on siis kokoelma organisaa-

tion kesken jaettuja, johdonmukaisesti järjestettyjä malleja ja käytänteitä, yhtenäinen tapa toimia (Kholmatova 2017). Alla olevassa kuviossa 1 havainnollistetaan suunnittelujärjestelmän osia.



Kuvio 1. Suunnittelujärjestemä.

Suunnittelujärjestelmän rakentamisen apuna voi käyttää esimerkiksi Brad Frostin kehittämää atomisuunnittelun menetelmää (engl. *Atomic Design*) (Frost 2016). Menetelmä perustuu ajatukseen, että suurempia ja monimutkaisempia komponentteja voidaan rakentaa pienemmistä ja yksinkertaisemmista komponenteista. Ajatus on samanlainen kuin komponenttilähtöisessä kehityksessä, mutta Frost ajattelee menetelmän pikemminkin hierarkioiden kautta. Hän nimeää suunnitteluhierarkian viisi tasoa:

- **Atomit** ovat käyttöliittymän pienimpiä osia, joita ei voida enää jakaa pienempiin osiin. Atomeita ovat esimerkiksi painike, syötekenttä ja ikoni.
- **Molekyylit** ovat atomeista koottuja suhteellisen yksinkertaisia komponentteja, jotka muodostavat yhdessä jonkinlaisen kokonaisuuden. Tällainen olisi esimerkiksi syötekentästä, sen otsikosta ja painikkeesta muodostuva lomake.
- **Organismit** ovat monimutkaisempia molekyyleistä ja/tai atomeista koottuja kokonaisuuksia. Organismin osa voi olla myös toinen organismi. Esimerkkinä organismista voisi olla verkkosivun yläpalkki, joka koostuu logosta, otsikosta, hakulomakkeesta ja kirjautumispainikkeesta.

- **Mallit** asettelevat atomit, molekyylit ja organismit järjestykseen muodostaen sivusuunnitelmia. Esimerkkinä voisi olla vaikkapa etusivun asettelu, jossa hallittu kokonaisuus syntyy useasta osasta. Mallien tarkoitus on pitää yllä komponenttien järjestystä sivulla, ei keskittyä sen sisältöön.
- **Näkymät** muodostuvat, kun malleihin lisätään todellista sisältöä. Lopullisten näkymien perusteella voidaan havaita tarvetta muunnella malleja, jotta kokonaisuudesta saadaan toimiva monenlaisella sisällöllä.

Tyylioppaan ja suunnittelujärjestelmän rakentamisessa atomisuunnittelun menetelmästä voi olla hyötyä. Frost muistuttaa, että vaiheiden suoritusjärjestystä ei ole tarkoin säädelty. Tärkeämpää on ajatella menetelmää henkisenä mallina, jonka avulla ymmärrämme paitsi kokonaisuutta, myös pieniä yksityiskohtia. (Frost 2016)

2.5 Tyylinvalvonta

Kun suunnittelujärjestelmä on luotu, sen käyttöä tulee myös valvoa. Tyylioppaan ja digitaalisen tuotteen välinen yhteys on näin muodostettu mutta sen hyödyntämisestä ei ole varmuutta. Staattisen tyylioppaan kohdalla sen noudattaminen jää kehittäjän vastuulle. Ohjeiden sisäistäminen ja niiden tarkka soveltaminen paineen alla on yksilöllistä ja epävarmaa. Elävän tyylioppaan tapauksessa tyyliohjeen noudattaminen on pääosin automatisoitua. Mikään ei kuitenkaan teknisesti estä kehittäjää lisäämästä esimerkiksi tyylimäärityksiä käyttämiinsä komponentteihin. On oltava jokin keino, jolla tyyliohjeen noudattamista voidaan valvoa.

Graafiseen käyttöliittymätestaukseen kuuluu sekä visuaalista että toiminnallista testausta. Käyttöliittymän toiminnallinen testaus koetaan tärkeänä osana kehitystyötä, koska sillä varmistetaan, että sovellus toimii odotusten mukaisesti. Visuaalinen testaus sen sijaan koetaan usein toissijaiseksi. Sillä on kuitenkin käyttäjäkokemuksen kannalta tärkeä merkitys ja se vaikuttaa suoraan tuotteen laatuun ja uskottavuuteen (Software Testing Help 2023). Tässä tutkimuksessa keskitytään vain käyttöliittymän visuaaliseen testaamiseen ja sen myötä tyylinvalvontaan.

Käyttöliittymän visuaalinen testaus tarkoittaa eroavaisuuksien havainnointia sovelluksen sisällössä ja ulkoasussa, eli siinä, mistä elementeistä sivu koostuu, miten ne on aseteltu ja miltä ne näyttävät ja tuntuvat. Sivun sisällön oikeellisuutta arvioidaan elementtien sijainnin, muodon ja koon perusteella. Varmistetaan, etteivät elementit sijoitu päällekkäin ja että ne ovat näkyvissä. Ulkoasun oikeellisuus todennetaan tarkastelemalla mm. värejä, fontteja, kirkkautta ja kont-

rastia. Tänä päivänä oman haasteensa tuovat erilaiset käyttöjärjestelmät, verkkoselaimet, resoluutiot ja lisäksi valtava määrä erilaisia laitteita. Visuaalisen testauksen tavoitteena on myös varmistaa, että sovellus näyttää oikealta ja ulkoasu on yhtenäinen huolimatta käyttäjän valitsemasta yhdistelmästä.(Software Testing Help 2023)

Visuaalinen testaus voidaan toteuttaa manuaalisesti, tarkastelemalla kahta tilannekuvaa vierekkäin ja etsimällä poikkeavuuksia (Software Testing Help 2023). Ihmissilmä havaitsee riittävät eroavaisuudet, eikä pikselin tarkkuudelle tarvitse mennä, jos koko tyylioppaan tavoitteena on yhtenäinen ilme. Tällainen testaus on kuitenkin käsityötä ja työn laatu on yksilöllistä. Yksi testaaja havaitsee erot toista tarkemmin. Lisäksi tehtävä vaikeutuu merkittävästi, kun tarkastettavaa sisältöä on paljon, tai kun jokainen mahdollinen näyttöyhdistelmä on tutkittava erikseen.

Visuaalinen testaus voidaan toteuttaa myös automaattisesti niin, että ohjelmisto suorittaa elementtien visuaalisen vertailun erilaisissa näyttöyhdistelmissä. Tämä voi tapahtua kahdella tavalla: snapshot-testauksena tai tekoälyalgoritmeihin ja tietokonenäköön perustuvana testauksena (Software Testing Help 2023).

Valtaosa markkinoilla olevista visuaalisen testauksen työkaluista käyttää snapshot-tekniikkaa. Siinä ohjelma ottaa testauksen eri vaiheissa käyttöliittymästä kuvakaappauksia, joita verrataan verrokkikuviin. Vertailu tapahtuu tutkimalla kuvien bittikarttoja, käymällä läpi jokainen pikselipari vuorollaan, jolloin virheilmoitus syntyy, kun pikseleiden välillä on eroja. Testauksen jälkeen ohjelma luo raportin, jota testaaja voi analysoida. Tällä tavoin virheitä voidaan löytää tarkemmin, nopeammin ja johdonmukaisemmin kuin ihmissilmällä. Menetelmään liittyy kuitenkin haasteita. Erilaiset selaimet, näytönohjaimet ja resoluutiot, jopa renderöintialgoritmit aiheuttavat merkityksettömiä pikselieroja, eikä kyse ole aidosta visuaalisesta virheestä. Testaajan analyysin varaan jää löytää todelliset virheet, mikä jälleen vaatii manuaalista työtä (Xu 2022).

Modernein tapa on tekoälyalgoritmien ja tietokonenäön käyttö visuaalisessa testaamisessa. Kaikesta ohjelmistotestauksessa käytetystä tekoälystä, on visuaalisella AI-testaamisella ollut viime vuosina suurin vaikutus (Volk 2021). Periaate on samanlainen kuin snapshot-testauksessa, kuvakaappauksia otetaan testauksen edetessä, mutta kuvien vertailu tapahtuu tekoälyavusteisesti. Pikselien vertailun sijaan tietokonenäön avulla kone koulutetaan tunnistamaan elementtejä käyttöliittymästä ja käsittelemään visuaalista tietoa kuten ihmisnäkö. Ihmisen näkö perustuu verkkokalvon, näköhermojen ja visuaalisen aivokuoren yhteistoimintaan (Pandey 2022). Tietokonenäkö hyödyntää kameroita, dataa ja algoritmeja suorittaakseen saman toiminnon (Pandey 2022). Tällainen menetelmä luo huomattavasti vähemmän virheellisiä hälytyksiä (engl.*false*

positives) kuin pikselikohtainen vertailu (Battat 2024) ja sillä voidaan analysoida tuhansia kuvia sekunnissa (Pandey 2022).

Tämän tutkimuksen puitteissa etsitään parhaiten soveltuva menetelmä ja työkalu kehitettävän tyylioppaan tyylinvalvojaksi. Jos päädytään markkinoilla valmiina olevaan ratkaisuun, on tärkeää, että se taipuu myös käyttöliittymän toiminnalliseen testaukseen huolimatta siitä, että tämä tutkimus keskittyy visuaaliseen testaukseen. Näin valitun työkalun täysi potentiaali on mahdollista hyödyntää jatkossa.

3 Tutkimuskysymykset, tutkimusmenetelmä ja sen soveltaminen

Tutkimusongelmana on, kuinka säilyttää käyttöliittymän yhtenäinen visuaalinen ilme, kun kehittäjiä on monta ja jokaisella oma tapansa toimia. Keskitytään liikaa toiminnallisuuteen ja visuaaliset seikat jäävät toissijaisiksi, vaikka niillä on tärkeä merkitys tuotteen laatuun ja kaupalliseen menestykseen.

3.1 Tutkimuskysymykset

Tutkimusongelman perusteella voidaan asettaa seuraavat tutkimuskysymykset:

- **TK1: Mitä asioita tyylioppaan täytyy sisältää, jotta varmistetaan käyttöliittymän yhtenäinen ilme ja parannetaan käyttäjäkokemusta?**
- ATK1: Kuinka tyyliopas tulee rakentaa, jotta se toimii yhdessä olemassa olevan komponenttikirjaston kanssa?
- ATK2: Mitä muutoksia kirjaston komponentteihin on tehtävä, jotta tyylimääritysten ylläpito voidaan keskittää yhteen paikkaan?
- ATK3: Miten varmistetaan, että eri kehittäjien luomat komponentit vastaavat annettuja tyyliohjeita?
- **TK2: Mitä vaihtoehtoja on olemassa valvoa komponenteista koostetun käyttöliittymän visuaalista ilmettä ja vastaavuutta tyylioppaaseen?**

Näistä kysymyksistä ensimmäinen ja viimeinen ovat varsinaiset tutkimuskysymykset ja kiteyttävät tutkimuksen tavoitteen. Alitutkimuskysymyksiä käytetään apuna tutkimuksen osatavoitteiden saavuttamisessa.

3.2 Tutkimusmenetelmän periaatteet

Tietojärjestelmätieteen tutkimusta luonnehtivat kaksi toisiaan täydentävää mutta erillistä paradigmaa: käyttäytymistiede (engl. *behavioral science*) ja suunnittelutiede (engl. *design science*). Käyttäytymistiede pyrkii kehittämään ja todentamaan teorioita, jotka selittävät tai ennustavat ihmisten tai organisaation käyttäytymistä. Suunnittelutiede sen sijaan pyrkii laajentamaan ihmisten ja organisaatioiden valmiuksia luomalla uusia innovatiivisia työkaluja, artefakteja.

(Hevner ym. 2004)

Tämän tutkimuksen menetelmänä on suunnittelututkimus (engl. *design science research*), jonka tuloksena suunnitellaan ja toteutetaan uusi artefakti, työkalu, joka ratkaisee kohdeorganisaation ongelman. Suunnittelututkimus on siis ratkaisukeskeistä tutkimusta, jossa käytännön ongelmaan haetaan toimivaa ratkaisua. Tutkimuksen tuloksena suunnitellaan jokin uusi, innovatiivinen IT-artefakti, joka ratkaisee ratkaisemattoman ongelman, tai tarjoaa tehokkaamman ratkaisun jo aiemmin ratkaistuun ongelmaan. Menetelmän tavoitteena on parantaa sekä teoriaa että käytäntöä kehittämällä artefakteja, jotka vastaavat todellisiin haasteisiin. Suunnittelututkimukseen kuuluu näiden artefaktien rakentaminen, niiden hyödyllisyyden ja tehokkuuden testaaminen sekä edelleen jalostaminen testauksesta saadun palautteen ja arvioinnin perusteella (Hevner ym. 2004; Peffers ym. 2007).

Suunnittelun käsite tekniikassa on tuttu jo varhaisilta ajoilta, mutta varsinainen suunnittelututkimus, sen muodollinen tunnustaminen strukturoituna tutkimusmenetelmänä nousi esiin vasta 1990-luvulla, kun haluttiin erottaa puhtaasti teoreettinen työ käytännön ratkaisujen kehittämisestä. Alan Hevnerin ja hänen kollegoidensa vuonna 2004 julkaisemaa artikkelia ” Design Science in Information Systems Research” pidetään akateemisessa kirjallisuudessa laajalti perustavanlaatuisena.

Hevner ym. (2004) mukaan suunnittelututkimukselle luonteenomaista on se, että ratkaistavana on ns. viheliäinen ongelma, jolle he tietojärjestelmätieteen osalta määrittelevät seuraavat tunnusmerkit:

- epävakaat vaatimukset ja rajoitukset, jotka perustuvat huonosti määriteltyyn kontekstiin
- monimutkainen vuorovaikutussuhde ongelman ja sen ratkaisun osakomponenttien välillä
- luontainen joustavuus muuttaa suunnitteluprosesseja sekä artefakteja
- kriittinen riippuvuus ihmisen kognitiivisista kyvyistä tehokkaiden ratkaisujen tuottamiseksi
- kriittinen riippuvuus ihmisen sosiaalisista kyvyistä tehokkaiden ratkaisujen tuottamiseksi.

Hevner ym. (2004) mukaan suunnittelutieteen paradigmassa tieto ja ymmärrys vallitsevasta ongelmasta ja sen ratkaisu saavutetaan rakentamalla ja hyödyntämällä suunniteltua artefaktia. Suunnitteluprosessi sisältää sarjan asiantuntijatoimia, joka johtaa innovatiivisen artefaktin luomiseen. Artefaktin evaluointi puolestaan antaa palautetta ja lisää ymmärrystä ongelmasta

parantaen näin sekä tuotteen laatua ja että itse suunnitteluprosessia. Tällaista rakenna-evaluoi-silmukkaa iteroidaan useita kertoja, kunnes artefakti saa lopullisen muotonsa (March ja Smith 1995). Tällä tavoin suunnittelutieteen tutkija kehittää samaan aikaan sekä suunnitteluprosessia että suunniteltua artefaktia osana tutkimusta (Hevner ym. 2004). Suunnittelututkimusta koskevassa artikkelissaan Hevner ym. (2004) määrittelevät seitsemän ohjenuoraa tutkimusmenetelmän ymmärtämiseen, toteuttamiseen ja arviointiin:

1. **Luodaan innovatiivinen ja tarkoituksenmukainen artefakti.** Suunnittelututkimuksen tulos on määritelmänsä mukaan tarkoituksenmukainen IT-artefakti, joka on luotu ratkaisemaan tärkeä organisatorinen ongelma. Se on kuvattava tehokkaasti, jotta se voidaan toteuttaa ja soveltaa asianmukaisella tavalla ympäristössä, johon se on suunniteltu.

2. **Ongelma on tarkoin määritetty.** Koska artefaktin tulee olla tarkoituksenmukainen, sen tulee luoda hyödyllinen ratkaisu määriteltyyn ongelmaan. Tavoitteena on hankkia tietoa ja ymmärrystä, joka mahdollistaa tähän asti ratkaisemattomien, tärkeiden liiketoimintaongelmien teknologiapohjaisten ratkaisujen kehittämisen ja toteuttamisen. Suunnittelutiede lähestyy tätä tavoitetta rakentamalla innovatiivisia artefakteja, joista jokaisen on tarjottava uutta tietoa ja haastettava entiset ratkaisut. Ongelma voidaan määritellä eroiksi tavoitetilan ja järjestelmän nykyisen tilan välillä. Ongelmanratkaisu on etsintäprosessi, jossa etsitään tehokkainta ratkaisua näiden erojen vähentämiseksi tai poistamiseksi.

3. **Artefaktin evaluointi on ratkaisevaa.** Suunnitellun artefaktin hyödyllisyys, laatu ja tehokkuus on osoitettava tarkasti hyvin toteutettujen arviointimenetelmien avulla. IT-artefakteja voidaan arvioida toimivuuden, täydellisyyden, johdonmukaisuuden, tarkkuuden, suorituskyvyn, luotettavuuden, käytettävyyden ja yhteensopivuuden organisaatioon ja muiden asiaankuuluvien laatuattribuuttien perusteella. Arviointivaihe antaa olennaista palautetta suunnitteluprosessin ja kehitettävän tuotteen laadusta. Artefakti on täydellinen ja tehokas, kun se täyttää sen ongelman vaatimukset ja rajoitukset, siinä liiketoimintaympäristössä, jossa sen oli tarkoitus toimia.

4. **Tutkimuksen tulee tarjota jokin selkeä panos.** Tehokkaan suunnittelututkimuksen on tarjottava selkeä panos joko suunnitteluartefaktiin, suunnittelun perustuksiin ja/tai suunnittelun menetelmiin liittyen. Useimmiten tutkimuksen tarjoama panos on suunniteltu IT-artefakti.

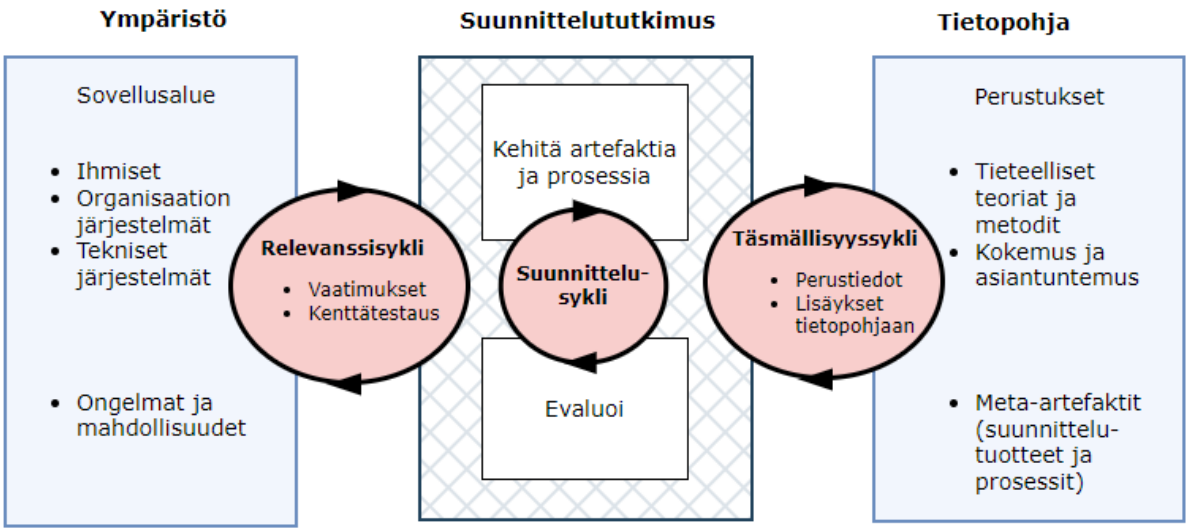
5. **Artefaktin tulee olla täsmällisesti määritelty, virallisesti esitetty ja johdonmukainen.** Täsmällisyys liittyy tapaan, jolla tutkimus suoritetaan. Suunnittelututkimus edellyttää täsmällisten menetelmien soveltamista sekä suunnitellun artefaktin rakentamisessa että sen arvioinnissa.

6. **Suunnittelutiede toimii parhaan ratkaisun etsintäprosessina.** Suunnittelututkimus on pohjimmiltaan etsintäprosessi tehokkaan ratkaisun löytämiseksi. Ratkaisua haetaan iteroiden, kunnes paras ja optimaalisin tulos saavutetaan.

7. **Suunnittelututkimuksen tulokset on kommunikoitava tehokkaasti.** Suunnittelututkimuksen tulokset tulee esittää sekä teknologiaorientoituneelle että johtamislähtöiselle yleisölle. Tämä antaa organisaatiolle mahdollisuuden hyödyntää artefaktin tarjoamia etuja ja antaa samalla myös tutkijoille mahdollisuuden rakentaa kumulatiivista tietopohjaa. Näin luodaan ymmärrystä prosesseista, joilla artefakti on rakennettu ja kuinka sitä on arvioitu.

3.3 Tutkimusmenetelmän keskeiset vaiheet

Siinä missä Hevner ym. (2004) määrittelevät tutkimusmenetelmän yleiset ohjenuorat, keskittyy Hevner (2007) artikkelissaan analysoimaan tutkimusprosessia kolmen läheisesti toisiinsa liittyvän toimintasyklin kautta, jotka on esitetty kuviossa 2.



Kuvio 2. Suunnittelututkimuksen vaiheet Hevner (2007) mallin mukaan, vapaa käännös.

Relevanssisyklissä (engl. *relevance cycle*) määritellään vaatimukset, jotka syntyvät tutkimuksen kontekstuaalisesta ympäristöstä, kuten ihmiset, organisaatio ja tekniset järjestelmät. Samalla pyritään tunnistamaan uusia mahdollisuuksia ja ongelmia kyseisessä ympäristössä. Vaatimusten lisäksi relevanssisyklissä määritellään myös hyväksymiskriteerit tutkimustulosten lopullista arviointia varten. Suunnittelututkimuksen tuotos, artefakti palautetaan ympäristöönsä kenttätestausta varten. Relevanssisyklin toinen iteraatio alkaa palautteella kenttätestauksesta ja etenee tutkimusvaatimusten uudelleenmääritykseen todellisen kokemuksen perusteella. Relevanssisykli yhdistää näin tutkimusprojektin kontekstuaalisen ympäristön suunnittelutieteen toimintoihin.

Täsmällisyyssyklissä (engl. *rigor cycle*) tarkastellaan teorioita ja menetelmiä, alan kokemusta ja asiantuntemusta sekä sovellusalueelta löytyneitä artefakteja ja prosesseja, eli tutkimuksen tietopohjaa. Tutkimuksen aikana syntyvä uusi tieto lisätään olemassa olevaan tietopohjaan. Tällaisia lisäyksiä voivat olla laajennukset teorioihin, uudet artefaktit sekä tutkimuksen perusteella syntyvä kokemus. Näin voidaan arvioida tutkimusprojektin innovatiivisuutta. Tällä tavoin täsmällisyyssykli yhdistää suunnittelutieteen toiminnot tietopohjaan.

Suunnittelusykli (engl. *design cycle*) on jokaisen suunnittelututkimusprojektin sydän. Tässä syklissä iteroidaan varsinaisen artefaktin ja prosessien rakennus- ja evaluointivaiheita, kunnes haluttu lopputulos saavutetaan. Hevner (2007) pitää tärkeänä ymmärtää suunnittelusyklin riippuvan kahdesta muusta syklistä. Artefaktia arvioidaan sille relevanssisyklissä asetettuja vaatimuksia vastaan. Suunnittelu- ja evaluointiteoriat ja -metodit puolestaan pohjautuvat täsmällisyyssykliin.

Hevner (2007) mukaan näiden kolmen syklin on oltava läsnä ja selkeästi tunnistettavissa hyvässä suunnittelutieteen tutkimusprojektissa. Tämä erottaa suunnittelututkimuksen selvästi muista tutkimusparadigmoista.

3.4 Menetelmän käyttö tässä tutkimuksessa

Suunnittelututkimuksen valinta tämän tutkimuksen menetelmäksi on perusteltua, koska tutkimus on selkeästi ratkaisulähtöinen ja on olemassa aito ongelma. Tutkimuksen tuloksena suunnitellaan ja toteutetaan uusi artefakti, työkalu, joka vastaa kohdeorganisaation konkreettiseen haasteeseen. Toteutettavaa artefaktia tutkitaan ja evaluoidaan siinä kontekstissa, jossa sillä on tarkoitus. Samalla tuotetaan uutta ymmärrystä tilanteesta, jossa artefakti vaikuttaa.

Kohdeorganisaation ongelma lukeutuu viheliäiseksi. Vuorovaikutussuhde nykyisen komponenttikirjaston komponenttien välillä on monimutkainen, eikä tue tyyliltään yhtenäisen ilmeen muodostusta. Kyseessä on lisäksi ongelma, jonka ratkaisu perustuu ihmisen sekä kognitiivisiin että sosiaalisiin kykyihin. Kun useat kehittäjät luovat komponentteja ilman yhteisiä ohjeita, lopputulos on vahvasti riippuvainen tekijänsä teknisistä taidoista, ajatusmallista ja siitä kuinka hän itse on tehtävän ymmärtänyt. Toteutus riippuu myös ryhmädynamiikasta ja siitä, miten tehokkaasti tiimityötä hyödynnetään parhaan ratkaisun saavuttamiseksi.

3.5 Tutkimuksen vaiheet

Tutkimus toteutetaan sykleittäin Hevner (2007) mallia mukaillen. Relevanssisyklissä selvitetään toteutettavan työkalun vaatimukset ja hyväksymiskriteerit. Täsmällisyyssyklissä selvitetään tutkimuksen tietopohja, olemassa olevat ratkaisut, työskentelytavat, tietämys, kokemus ja asiantuntemus. Tutkimuksen tärkein sykli on kuitenkin suunnittelusykli, jota iteroiden suunnitellaan ja toteutetaan työkalu kohdeorganisaation tarpeisiin. Jokaisen iteraation päätteeksi suoritetaan evaluointi. Lopuksi palataan relevanssisykliin arvioimaan, kuinka hyvin asetetut vaatimukset täyttyivät ja täsmällisyyssykliin selvittämään, mitä uutta tietämystä artefakti lisäsi tietopohjaan.

3.5.1 Nykytilan kartoitus: täsmällisyyssykli

Ensin on muodostettava kuva kohdeorganisaation tämänhetkisestä toimintatavasta, teknisistä ratkaisuista sekä näihin liittyvistä haasteista. Tutkitaan komponenttikirjaston sisältöä ja poimitaan kaikki se, mikä voisi jollain lailla liittyä käyttöliittymän ilmeeseen. Kuinka laajalti Chakra UI -komponentteja on käytössä ja mitä tyylejä niihin liittyy? Kuvataan esimerkinomaisesti, kuinka tuoterungot rakentuvat kirjaston komponenteista.

3.5.2 Tavoitetilan kartoitus: relevanssisykli

Määritellään, mitkä komponentit tulisi sisällyttää tyylioppaaseen ja millaisia variaatioita niistä tulisi olla. Tämän perusteella muodostetaan tyylioppaan tavoitetila. Määritellään myös, miltä osin komponenttien tulisi olla muokattavissa asiakastoteutuksissa ja kuinka tämä tulee toteuttaa. Millaisia asioita tyylinvalvojan halutaan tarkastavan yksittäisistä komponenteista ja edelleen tuoterungoista? Kartoitetaan saatavilla olevien testityökalujen kirjo ja valitaan tutkimuksessa käytettävät työkalut. Päätetään, miten työkalun onnistumista arvioidaan tutkimuksen

päätteeksi ja mitkä ovat hyväksymiskriteerit.

3.5.3 Tyyliopas: suunnittelusykli

Suunnitellaan ja toteutetaan tyylioppaan sisältö, rakenne, tarvittavat komponentit ja niiden eri variaatiot. Kirjoitetaan niihin liittyvä dokumentaatio sekä yleiset tyyliohjeet. Toteutus evaluoidaan ja tarvittaessa suoritetaan uusi iteraatio.

3.5.4 Ohjeet komponenttikirjaston rekonstruktiolle: suunnittelusykli

Nykyisen komponenttikirjaston koko ja rakenne huomioiden täydellisen rekonstruktion toteuttaminen tutkimuksen puitteissa on mahdotonta. Sen sijaan laaditaan ohjeet komponenttikirjaston rekonstruktiolle. Täsmällisyyssyklissä kerättyjen tietojen perusteella ohjeistetaan kohdeorganisaatiota komponenttien tyylimääritysten poistamisessa ja siinä, kuinka viittaaminen tyylioppaan komponentteihin tehokkaimmin tapahtuisi. Ohjataan myös sivumallien käyttöönotossa ja muutoksissa organismikohtaisten näyte-ikkunoiden (Playground) esityksissä. Määritellään toimenpiteet vaiheittain, jotta aikaansaadaan komponenttikirjasto, joka perustuu elävään ja helposti ylläpidettävään tyylioppaaseen.

3.5.5 Tyylinvalvoja: suunnittelusykli

Suoritetaan tutkimukseen valittujen testityökalujen vertailu toteuttamalla kullakin työkalulla sama muutaman komponentin testijoukko. Evaluoidaan vertailun tulokset ja valitaan varsinaisessa tyylinvalvojassa käytettävä työkalu. Suunnitellaan ja toteutetaan visuaaliset käyttöliittymätestit. Yksittäisten komponenttien sekä mahdollisuuksien mukaan tuoterunkojen näyte-ikkunoiden graafinen ulkoasu testataan ohjelmallisesti. Komponenttikirjastoon liitetään raportti visuaalisen testauksen tilasta ja koodikattavuudesta. Toteutus evaluoidaan ja tarvittaessa suoritetaan uusi iteraatio.

3.6 Artefaktin evaluointi

Suunnittelusyklien iteraatioiden päätteeksi suoritetaan artefaktin evaluointi. Arvioidaan, täyttääkö suunniteltu ja toteutettu työkalu iteraatiolle asetetut vaatimukset. Arvioidaan myös, kuinka herkkä työkalu on ympäristön muutoksille tai muutoksille itse työkalussa eli kuinka työkalun ja ympäristön vuorovaikutus muuttuu, jos ympäristö tai työkalu itse muuttuu.

Työkalulle asetetaan laatuvaatimuksia, joita vastaan suunniteltua ja toteutettua artefaktia arvioidaan. Valitut laatuvaatimukset ovat toiminnallisuus, käytettävyys, tehokkuus ja ylläpidettävyys. Näiden lisäksi artefaktin osia arvioidaan niille vaatimusmäärittelyn kautta asetettuja tarkempia vaatimuksia vastaan. Lähdekoodi arvioidaan myös kohdeorganisaation ohjelmointikäytäntöjä silmällä pitäen.

Käytännössä arviointi tapahtuu keräämällä asiantuntijapalautetta kohdeorganisaatiosta aiheeseen perehtyneiltä kehittäjiltä hyvää tieteellistä käytäntöä noudattaen. Kohdeorganisaation puolelta ohjaaja toimii ensisijaisena arvioijana. Vaiheita iteroidaan, kunnes haluttu lopputulos kunkin vaiheen osalta saavutetaan.

4 Työkalun suunnittelu ja toteutus sykleittäin

Työkalun suunnittelussa pyritään noudattamaan menetelmän mukaista järjestystä, mutta vaiheiden välillä joudutaan iteroimaan. Täsmällisyyssyklin toteuttaminen ensin on väistämätöntä, jotta saadaan riittävä tietämys aloitustilanteesta. Sen sijaan relevanssisyklin osalta on edettävä vaiheittain: tyylioppaan vaatimukset on määriteltävä ensin ja siirryttävä suunnittelusyklin kautta toteutukseen, jotta nähdään, mitä tyylinvalvojalta voidaan edes vaatia. Hevner (2007) puhuu artikkelissaan artefaktin palauttamisesta kenttätestaukseen ja sen arvioimisesta näiden tulosten perusteella. Hän kehottaa iteroimaan myös relevanssisykliä, kunnes artefakti täyttää sille asetetut vaatimukset. Tämän tutkimuksen kohdalla työkalun vaatimukset ovat tietoisesti epätäydellisiä ensimmäisen iteraation aikaan, koska tyylinvalvojan vaatimusmäärittely voidaan suorittaa varsinaisesti vasta tyylioppaan toteutuksen jälkeen.

4.1 Täsmällisyyssykli - komponenttikirjaston nykytila

Aliluvuissa kuvataan kohdeorganisaation tämänhetkistä toimintatapaa, teknisiä ratkaisuja ja näihin liittyviä haasteita. Esitellään komponenttikirjaston sisältöä, komponenttien tyyleihin liittyviä määrityksiä ja niiden toteutustapaa. Luvussa kerrotaan esimerkinomaisesti myös, kuinka tuoterungot rakentuvat kirjaston komponenteista ja kuvataan esimerkkimoduulia UML-diagrammilla.

4.1.1 Modulaarinen tuoterakenne

Kohdeorganisaatio kehittää tuotteita, joita käyttävät useat asiakkaat. Tuotteet räätälöidään asiakkaan tarpeiden mukaan komponenteista koostetuista tuoterungoista. Tutkimuksen alkaessa kohdeorganisaatiolla on käytössään komponenttikirjasto, joka sisältää erään tuoterungon rakentamiseen tarkoitettuja, uudelleenkäytettäviä komponentteja. Kirjaston avulla rakennettu tuote on tällä hetkellä käytössä 12 asiakastoteutuksessa ympäri Suomen.

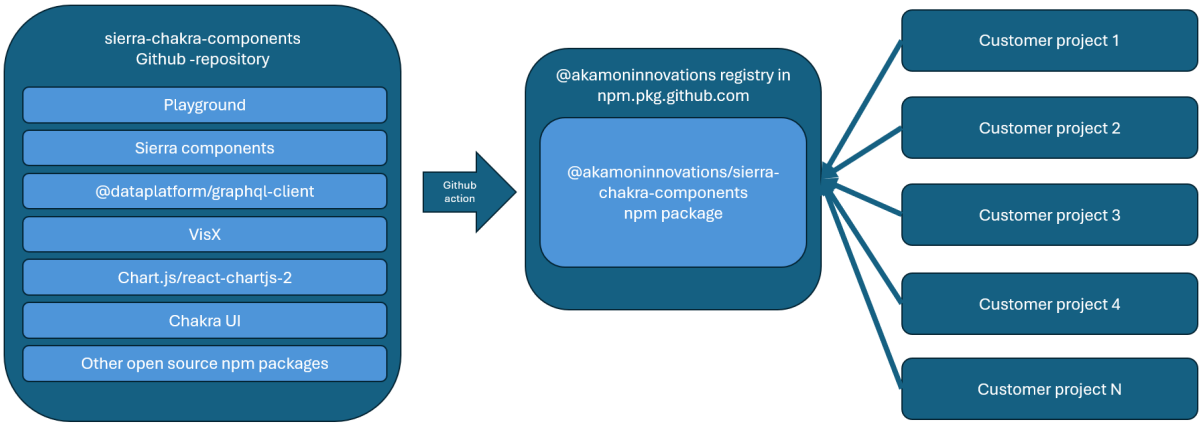
Asiakastoteutuksissa tuotteeseen liitetään ne kirjaston komponenteista kootut ominaisuudet, jotka asiakas haluaa. Myöhemmin toteutukseen on mahdollisuus lisätä ominaisuuksia. Komponenttikirjasto sisältää kaikki tämän tuotteen saatavilla olevat ominaisuudet. Asiakastoteutuksissa on lisäksi mahdollista räätälöidä tyyliin ja teemaan liittyviä asioita, kuten värit ja fontit. Tämä tapahtuu käytännössä sovelluksessa olevan asiakaskohtaisen teematiedoston avulla.

Tuotteella on siis perusrunko, jota muuntelemalla voidaan räätälöidä asiakkaan toiveiden mukainen kokonaisuus. Puhutaan modulaarisesta tuoterakenteesta, jossa kaikilla moduuleilla on selkeät rajapinnat ja ne toimivat toisistaan riippumatta.

Kohdeorganisaatiossa työskentelee tällä hetkellä 18 ohjelmistokehittäjää, joista 10-18 työskentelee komponenttikirjaston parissa.

4.1.2 Tekninen toteutus

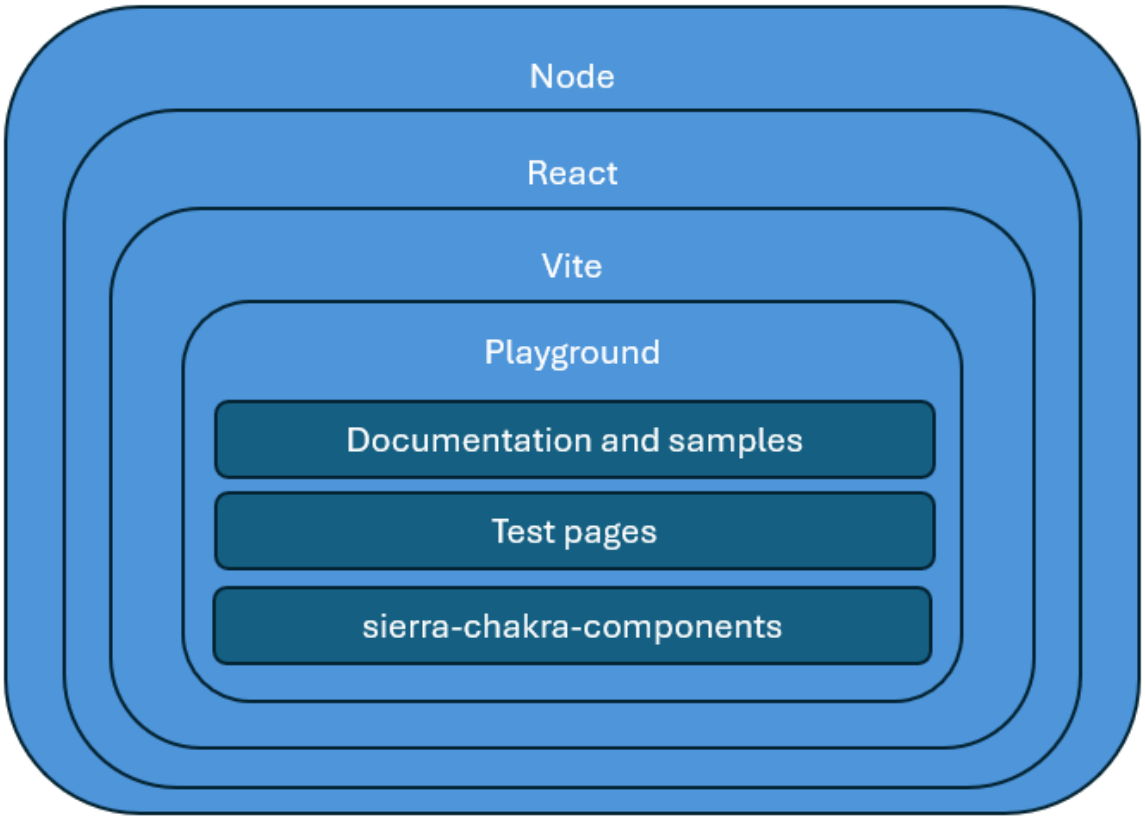
Kohdeorganisaation sierra-chakra-components-komponenttikirjasto on toteutettu TypeScript-kielellä (Microsoft 2024b) React (Meta 2024) + Vite (You 2024a)-projektina. Komponenttikirjastosta julkaistaan npm-paketti Github:in npm-rekisteriin, josta se jaellaan haluttuihin asiakasprojekteihin. Projektin versionhallinta on Github:ssa, josta sierra-chakra-components npm-paketin julkaisu on automatisoitu Github Action:ien avulla. Alla olevassa kuviossa 3 on esitetty komponenttikirjaston tekninen toimintaympäristö.



Kuvio 3. Komponenttikirjaston tekninen toimintaympäristö.

Toteutetuissa komponenteissa hyödynnetään avoimen lähdekoodin komponenttikirjastoa Chakra UI (Adebayo 2024). Datan vaativampia esityksiä varten on käytössä Chart.js ja yhteensopiva React-komponenttikirjasto react-chartjs-2 sekä datan visualisointiin erikoistunut VisX-kirjasto. Komponenttien tilanhallintaan on käytetty Zustand:ia ja lokalisointi on toteutettu react-i18next-kirjastolla. Komponenttien tiedonhaku ja päivitys palvelinpuolelle tapahtuu GraphQL:lla (Foundation 2024) käyttäen Akamonin @dataplatform/graphql-client-komponenttikirjastoa.

Komponenttikirjastoon on toteutettu Playground-ympäristö, jonka avulla kehittäjät voivat ajaa ja testata komponentteja lokaalissa kehitysympäristössä. Playground sisältää myös esimerkkejä, dokumentaation sekä testisivut. Playground:ia voidaan ajaa lokaalisti Node.js-ympäristössä (OpenJSFoundation 2024). Alla olevassa kuviossa 4 on esitetty Playground-ympäristö.



Kuvio 4. Komponenttikirjaston Playground-ympäristö.

4.1.3 Komponenttikirjaston komponentit

Komponenttikirjastossa on kartoitushetkellä 174 komponenttia, joista suurin osa on monimutkaisempia, useista pienemmistä komponenteista koostettuja komponentteja. Chakra UI -kirjastosta käytössä on 118 erilaista komponenttia, 18 Chakra-hookia, 14 välitettyä ominaisuusjoukkoa ja 25 ikonia. Lisäksi React:in ikonikirjastosta on käytössä 9 muuta ikonia. Kirjaston komponenteista vain 15:lla ei ole Chakra UI -riippuvuutta.

Kirjaston komponentit siis nojaavat vahvasti avoimen lähdekoodin Chakra UI -kirjastoon. Tämän tutkimuksen puitteissa keskustellaan myös siitä, mitä tämä riippuvuus merkitsee ja millaiset mahdollisuudet kirjaston nykyinen ja tuleva rakenne tarjoaa sen muuttamiseksi, jos esimerkiksi halutaan siirtyä käyttämään jotain toista avoimen lähdekoodin komponenttikirjastoa.

Tarvetta ei tällä hetkellä ole, mutta kirjaston laajentamisen ja jatkuvuuden kannalta on järkevää ennakoida mahdolliset muutostarpeet.

Valtaosa kirjaston komponenteista vastaa organismitason komponentteja, jotka koostuvat useista osista, jotkut jopa toisista organismeista. Atomitason komponentteja ei juurikaan ole, molekyylirakenteitakin vähemmän. Tyylikirjastoa rakennettaessa tämä on huomioitava niin, että olemassa olevia komponentteja on kenties pilkottava pienempiin osiin, jotta uudelleenkäytettävyys paranee ja tyylikomponentit saadaan keskitetysti jaettua kirjaston monimutkaisille komponenteille.

Useat kirjaston komponentit sisältävät dataa, jota haetaan GraphQL-rajapinnan kautta. Osin dataa haetaan saman sivun sisällä useaan eri kertaan, koska datan hakeminen tapahtuu komponentin sisällä. Olisi järkevää, jos datan hakeminen olisi sivukehyksen (engl. *wrapper*) vastuulla ja se välittäisi tiedon sivun muille komponenteille. Näin valtaosa komponenteista olisi ns. ”tyhmiä komponentteja”, jotka vaan vastaanottavat ja käsittelevät datan näytettäväksi käyttöliittymässä halutussa muodossa.

Komponenttikirjaston yhteydessä olevassa Playground:issa on tarkoituksena näyttää komponentit siinä asussaan, kuin ne valmiissa tuotteessa tulisivat olemaan. Playgound:issa ei ole tarkoitus esitellä yksittäisten komponenttien ulkoasua, vaan niistä muodostuvaa, testidataa sisältävää kokonaisuutta.

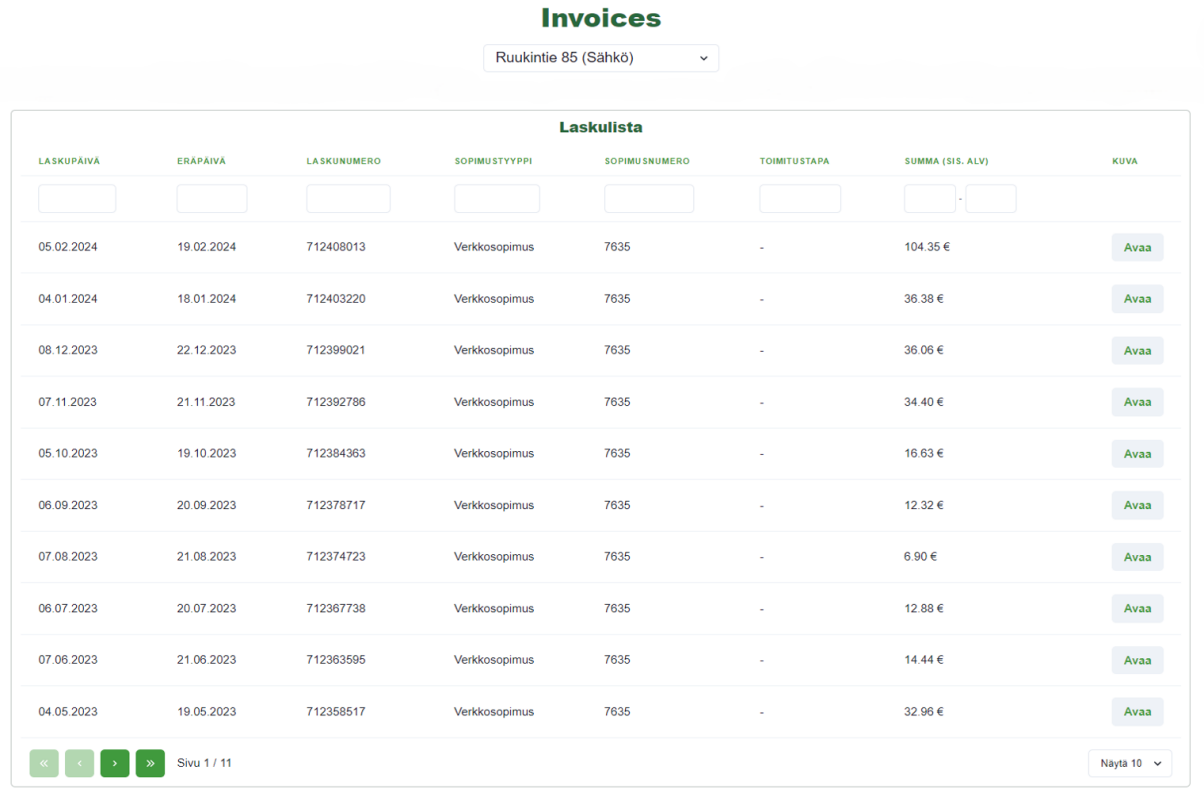
4.1.4 Tuoterungot ja komponenttien rakenne

Tällä hetkellä kirjasto on tarkoitettu vain tuoteperheen yhden tuoterungon kokoamiseen mutta olisi tietysti tehokasta luoda yksi kirjasto, jolla voisi rakentaa useamman tuotteen. Tämän tutkimuksen yhteydessä toteutettavan tyylikirjaston komponenttien tulee olla niin pieniä (atomeita, molekyylejä), että niitä voidaan hyödyntää jatkossa myös muiden tuoteperheen tuoterunkojen rakentamiseen.

Tuote on tarkoitettu energiayhtiöiden asiakkaille. Se sisältää asiakassuhteeseen liittyvien tietojen, kuten asiakastietojen ja sopimusten lisäksi runsaasti erilaista tietoa asiakkaan omasta energiankulutuksesta ja sen kustannuksista. Asiakkaan ja energiayhtiön välinen kommunikointi tapahtuu järjestelmän kautta. Asiakas voi selata laskuhistoriaansa ja tehdä muutoksia sopimuksiinsa. Sovellus antaa myös reaaliaikaista tietoa sähkön hinnasta ja esittää suosituksia edulliseen käyttöajankohtaan. Näytetään asiakkaan käyttövaikutus/kulutusvaikutus pörssisäh-

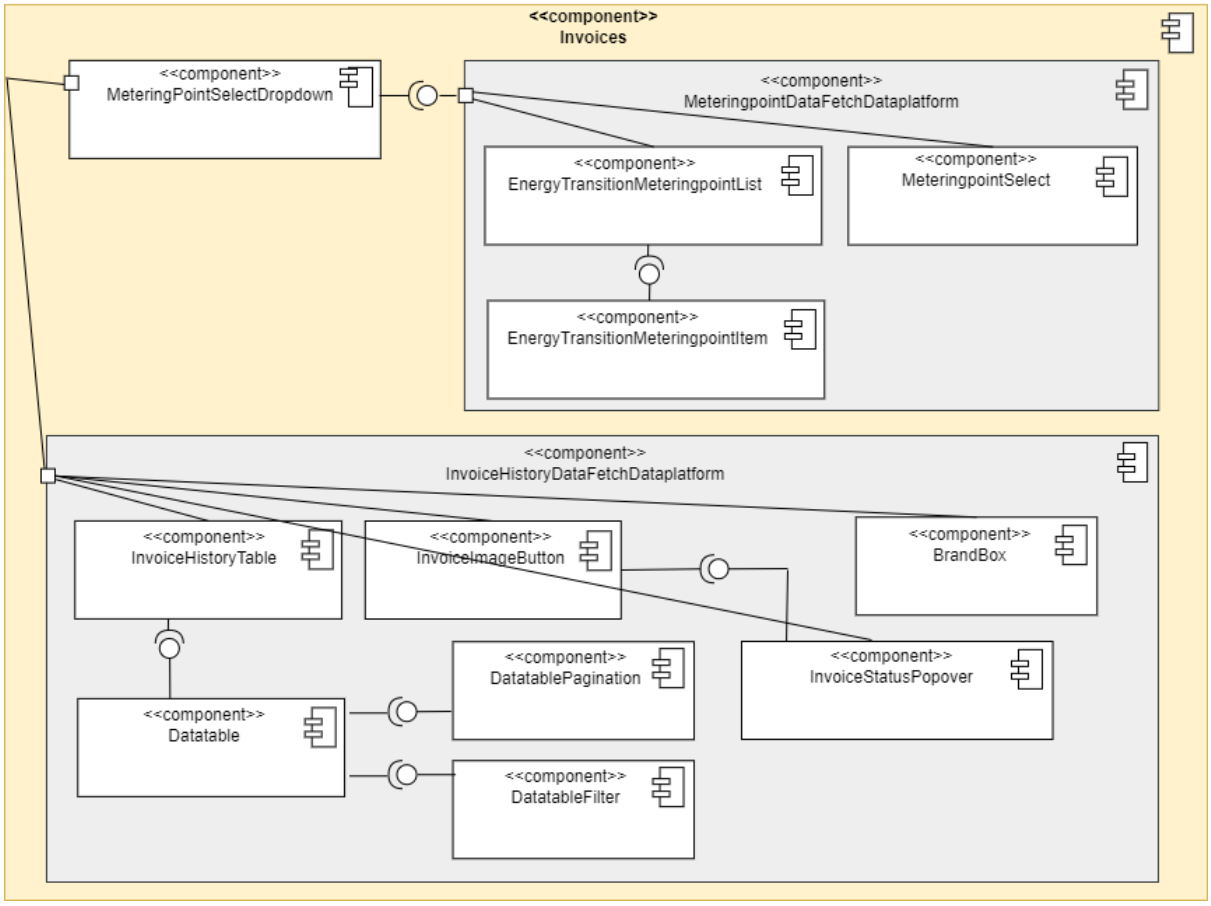
kön hintaan. Energiayhtiön ympäristöystävällisyydestä kertovat kaukolämmön alkuperä, päästöt ja hiilijalanjälki. Tuotteeseen on sisällytetty myös säätietoja.

Tarkastellaan esimerkinomaisesti asiakkaan laskusivun tuottavaa komponenttia `Invoices`, joka on esitetty kuviossa 5.



Kuvio 5. Asiakkaan laskusivun tuottava komponentti `Invoices`.

Tämän komponentin tuottamiseen tarvitaan yhteensä 14 kirjaston komponenttia, jotka vuorovaikuttavat keskenään. Kunkin komponentin graafinen esitys koostuu edelleen 0–16 Chakra UI -komponentista. Oheisesta UML-diagrammista (kuvio 6) nähdään komponentin rakennetta tarkemmin. Varsinainen `Invoices`-komponentti sisältää kaksi laajempaa, useamman komponentin kokonaisuutta (harmaat komponentit), joista ylempää käytetään myös muilla sivuilla. Keltaisen alueen `Invoices`-komponentti muodostaa raamit muille komponenteille. Se sisältää sivun otsikon ja kutsuu käyttöpaikan valintakomponenttia (alasvetovalikko) sekä datan esittävää taulukkoa, joilla puolestaan on muut UML-diagrammin mukaiset riippuvuudet.



Kuvio 6. `Invoices`-komponentin rakenne UML-kaaviona.

`Invoices`-komponentti on hyvä esimerkki tuoterungon yhdestä moduulista, joka toimii itsenäisesti. Se voidaan joko sisällyttää tai jättää pois asiakastoteutuksesta. Samalla periaatteella toimii esimerkiksi `Messages`-komponentti. Sen toteutukseen tarvitaan 19 kirjastossa olevan komponentin yhteistyötä. Samaan tapaan rakentuvat tuoterungon muutkin moduulit.

4.1.5 Komponenttien tyyli

Käyttöliittymän pääasiallinen ilme rakentuu tällä hetkellä Chakra UI -komponenttien ulkoasusta. Kuitenkin vain harvoin, on komponentteja käytetty sellaisenaan. Valtaosa Chakra-komponenteista on jollain tavalla tyylitelty: käytetty tiettyä varianttia, väriä, kokoa, marginaaleja tai välistyksiä. Kaikki nämä tyylimuotoilut on toteutettu kirjaston komponenttien sisällä niin, että jokaisella Chakra-instanssilla saattaa olla erilaiset tyylimääreet. Samat määreet osittain toistuvatkin, mutta toteutustapa selkeästi hankaloittaa tyylin ylläpitoa ja mahdollista muutosta. Esimerkiksi Chakra:n `Button`-komponentti on käytössä 42 eri komponentissa, joista lähes kaikkien yhteydessä on jonkinlainen muotoilu. Jos painikkeen ulkoasua halutaan yhtenäis-

tää, on muutos tehtävä jokaiseen `Button`-komponentin instanssiin. Käyttöliittymästä löytyy tällä hetkellä 17 erilaista painiketta, joista useimmilla on edelleen painikkeen tilaan liittyviä, osin toisistaan poikkeavia muotoiluja.

Sivujen asettelussa on käytetty monenlaista toteutustapaa. Vaikka yleisilme näyttää yhtenäiseltä, sen takaa löytyy eri komponentein toteutettuja rakenteita. Chakra UI:lla on valmiina mainio ryhmittelykomponentti `Stack`, josta edelleen on olemassa valmiit `VStack`- ja `HStack` -toteutukset (vertikaalinen ja horisontaalinen). Näitä on käytetty keskimäärin 30 komponentissa kutakin, mutta niiden lisäksi sivuasettelua on toteutettu hämmästyttävän usein `Flex`- ja `Box`-komponentein. Jälkimmäiset toimivat tässä tarkoituksessa kyllä, mutta vaativat enemmän tyylimäärityksiä näyttääkseen samalta kuin valmiit ryhmittelyyn tarkoitetut komponentit. Yhtenäisen ilmeen ja koodin luettavuuden vuoksi olisi hyvä käyttää samoja yhteisesti valittuja komponentteja.

Komponenttien responsiivisuus on toteutettu vaihtelevin tavoin. Useimmat komponentit toimivat eri kokoisilla näytöillä, mutta jälleen toteutustapa on erilainen. Osa komponenteista käyttää Chakra:n valmista hookia, `useBreakpointValue`, jolla elementtien tyyli sidotaan ruudun kokoon muuttujan avulla. Valtaosalla komponenteista responsiivisuus perustuu Chakra:n sisäänrakennettuihin taitekohtiin (engl. *breakpoint*), jotka voidaan määritellä kullekin projektille teeman yhteydessä. Komponenteille annetaan tällöin kutakin taitekohtaa vastaava tyylimääre. Jos teemassa ei näitä kohtia ole määritelty, käytetään Chakra:n oletusasetuksia. Tällaisen responsiivisuuden voi edelleen toteuttaa taulukko- tai olio-syntaksina. Molemmat tavat ovat käytössä tämän hetken komponenttikirjastossa. Jälleen, vähintäänkin koodin luettavuutta helpottaisi yhtenäinen tapa toimia.

Elementtien koko voidaan ilmaista useilla eri tavoilla ja mittayksiköillä. Kirjaston komponenteissa kokoja on määritelty ainakin yksiköissä `px`, `rem`, `vh` ja `%`. Lisäksi Chakra:n elementtikohtaisista koko-ominaisuuksista on käytössä välistyksille ja marginaaleille objektimuoto, esimerkiksi 2, joka vastaa kokoa `0.5rem` tai `8px`. Tekstin koko on yleensä määritelty Chakra:n oletuskokoja käyttäen, kuten `sm`, `md`, `lg`, mutta poikkeavia tapoja löytyy myös, jopa komponenttien sisältä. Myös tekstin paksuudessa, otsikoiden tyylissä sekä monissa muissa ominaisuuksissa on käytössä erilaisia tapoja saavuttaa jokin lopputulos.

Valtaosa komponenttien tyyleistä on siis määritelty kunkin instanssin yhteydessä. Mainittakoon vielä, että tämän lisäksi esimerkiksi `DatePicker`-komponentilla on täysin oma erillinen CSS-tiedostonsa, josta komponentin ulkoasu haetaan. Myös komponentit, joiden toteutuksessa

on käytetty charts.js- tai VisX-visualisointia aiheuttavat poikkeuksen toimintatavassa. Näiden osalta määritykset joko haetaan oletuksena käytettävästä kirjastosta tai annetaan vakioina komponentin toteutuksen yhteydessä.

Edellä mainitut epäjohdonmukaisuudet ja ylläpitoa hankaloittava rakenne kielii yhteisen tyylioppaan puutteesta. Suurena haasteena projektin onnistumiselle on se, että kirjastossa on jo valtava määrä valmista toimivaa koodia. Tätä tutkimusta tehdessä kirjasto on käytössä ja muuntuu koko ajan. Elävän tyylioppaan sisällyttäminen kirjastoon ja komponenttien nykyisten tyylien ohjaaminen yhteen paikkaan vaatii rakenteen massiivista uudelleen järjestelyä. Uuden toimintatavan hyväksyminen ja sen omaksuminen osaksi rutiinia vaatii organisaatiolta vahvaa sitoutumista ja joustoa. Tarkoitus on lopulta helpottaa kehittäjien työtä, ei hankaloittaa sitä.

4.2 Relevanssisykli: työkalun vaatimukset ja hyväksymiskriteerit

Tässä luvussa esitellään tyylioppaalle ja sen valvojalle asetetut vaatimukset. Toteutettavan työkalun tavoitetila määritellään vaatimusmäärittelyn avulla, joka jaetaan kahteen osaan, tyylioppaan ja tyylinvalvojan vaatimuksiin. Tyylioppaan osalta määritellään yleiset vaatimukset, tyyleihin liittyvät vaatimukset sekä toteutettavien komponenttien vaatimukset. Tyylinvalvojan osalta keskitytään työkalun valintaan ja sille määriteltäviin yleisiin vaatimuksiin. Luvussa esitellään myös, miten artefaktin onnistumista arvioidaan tutkimuksen päätteeksi ja mitkä ovat sen hyväksymiskriteerit.

4.2.1 Tyylioppaan vaatimukset

Kuviossa 7 on esitetty tyylioppaalle asetetut yleiset vaatimukset, jotka kattavat tyylioppaan sijainnin, rakenteen ja toteutuksen lisäksi vaatimuksia mm. komponenttien nimeämiseen, koodauskäytäntöön ja dokumentointiin liittyen. Myös asiakastoteutuksia varten tarvittava räätälöinti ja sen vaikutukset monikäyttöisiin graafisiin komponentteihin on otettava huomioon tavoitetilaa määriteltäessä. Yleisten vaatimusten yhteydessä asetetaan myös tyylioppaan hyväksymiskriteerit. Tämän tutkimuksen tavoitteena on toteuttaa sellainen osa tyyliopasta, jota voidaan käyttää esimerkkinä työn jatkamisessa kohti täydellistä tyyliopasta ja edelleen suunnittelujärjestelmää.

Nro	Nimi	Vaatimuksen alustus	Vaatimuksen kuvaus	Prioriteetti
1.1	Sijainti	Mikä on tyylioppaan toivottu sijainti kirjastoon nähden? Miten asettuu suhteessa valmiina oleviin komponentteihin?	Tyyliopas perustetaan komponenttikirjaston sisälle, Playground-projektiin, josta se on käytettävissä kaikille komponenteille.	Välttämätön
1.2	Rakenne	Millainen rakenne tyylioppaalla tulisi olla? Atomisuunnittelu, aakkosjärjestys vai joku muu?	Tyylikirjasto sisältää kolme pääryhmää: komponentit, tyylit ja sivumallit. Ryhmien sisältö järjestellään aakkosjärjestykseen.	Välttämätön
1.3	Toteutus	Mikä on tekninen toteutus? Käytetäänkö erillistä ohjelmistoa apuna (Storybook)? Toteutetaanko elävä vai staattinen tyyliopas? Mikä on tyylioppaan kieli?	Toteutetaan elävä tyyliopas (living style guide) komponenttikirjaston sisälle ilman erillistä työkalua. Tyylioppaan tulee olla suomenkielinen.	Välttämätön
1.4	Nimeäminen	Kuinka tyylikomponentit tulisi nimetä? Voivatko ne olla nimetty samoin kuin Chakrassa?	Tyyliopas esittelee komponenttien tyylin käyttäen yleisiä nimityksiä, kuten Button, Input tai Modal. Varsinaiset Akamon-spesifit komponentit nimetään käyttäen Akamon-alkuliitettä ja CamelCase -tapaa niin, että ne eroavat Chakran komponenteista.	Välttämätön
1.5	Dokumentointi	Missä tyylioppaan dokumentaatio on ja miten se hoidetaan, jos ei käytetä ulkopuolista työkalua, jossa dokumentointi on mukana?	Tyylioppaan toteutuksen yhteydessä kullekin sivulle lisätään lyhyt selite. Komponenttien lähdekoodi on oltava avattavissa tyylioppaan sivulta. Lisäksi komponenttien käyttämät ominaisuudet on esitettävä taulukkomuodossa tyylioppaan yhteydessä sekä Akamon-spesifien komponenttien osalta myös tarvittavat parametrit. Dokumentaatio on toteutettava mahdollisuuksien mukaan dynaamisesti.	Tärkeä
1.6	Koodauskäytäntö	Mitkä ovat oikeat tavat määritellä tyylejä, responsiivisuutta, sivujen asettelua tai elementtien kokoja? Millaista komponenttien tyyleihin liittyvää koodauskäytäntöä tulisi noudattaa?	Sovitaan yleisistä kirjauskäytännöistä.	Tärkeä
1.7	Räätälöinti	Miten asiakkaalle räätälöitävien tyyliominaisuuksien toteutus tapahtuu käytännössä? Halutaanko tyylien räätälöintiä rajoittaa?	Käyttöliittymään lisätään Sierran oletusteeman (SierraTheme) lisäksi customTheme -tyylitiedosto, jossa tyylien räätälöinti on mahdollista toteuttaa. Asiakkaan teemassa on mahdollista ylikirjoittaa lähes mikä tahansa oletustyylimääre. Yleisimpiä ovat kuitenkin värit ja fontit.	Välttämätön
1.8	Hyväksyminen	Mitkä ovat toteutetun tyylioppaan hyväksymiskriteerit, jotta tutkimuksen lopuksi voidaan sanoa sen toimivan toivotulla tavalla?	Tämän tutkimuksen puitteissa tarkoitus on saada toteutettua sellainen osa tyyliopasta, jota voidaan käyttää esimerkkinä työn jatkamiseen kohti kokonaista tyyliopasta. Toteutetaan muutama keskeinen komponentti ja niiden virtaus tyylikirjastosta visuaalisen testauksen kautta sovelluksen käyttöön. Tyylioppaalle asetetaan seuraavat laatuvaatimukset: toiminnallisuus, käytettävyys, tehokkuus ja ylläpidettävyys.	Välttämätön

Kuvio 7. Vaatimusmäärittely: tyylioppaan yleiset vaatimukset.

Tyylioppaan sijainti, rakenne ja tekninen toteutus olemassa olevaan komponenttikirjastoon nähden on tarkkaan mietittävä. Komponenttikirjasto sisältää jo runsaasti toimivaa koodia, jonka refaktorointi liittyy olennaisena osana tyylioppaan käyttöönottoon, kun toteutetaan elävää tyyliopasta. Sovelluksissa käytettävien komponenttien tulisi voida ongelmitta hyödyntää tyylikirjastoon luotuja komponentteja.

Ensimmäinen ajatus on luoda tyyliopas komponenttikirjaston Playground-projektin sisälle. Tyylikirjaston pohja luotaisiin Storybook-ohjelmistolla (Storybook 2024), joka on avoimen lähdekoodin käyttöliittymätyökalu. Storybook:illa voidaan luoda, tarkastella ja testata kom-

ponentteja eristyksissä muusta ohjelmistosta. Storybook:in keskeisin hyöty syntyy automaattisesti muodostuvasta komponenttien katalogimaisesta esityksestä käyttöliittymässä, jossa komponenttien eri variaatioita on helppo selata. Katalogi sisältää myös komponenttikohtaista dokumentaatiota, joka syntyy vaivatta komponenttien luonnin yhteydessä.

Kokeiltaessa tällaista toteutusta syntyy ajatus siitä, voisiko tyyliopas korvata koko Playground-projektin, joka tällä hetkellä toimii komponenttikirjaston ”näyteikkunana”. Nykyisessä Playground:issa ei esitetä yksittäisiä komponentteja vaan valmiita, monimutkaisia kokonaisuuksia, joista sovelluksen varsinaiset sivut muodostuvat. Storybook:in rakenne voitaisiin määritellä niin, että sen pienimmät komponentit (atomit) muodostaisivat suurempia kokonaisuuksia (molekyylit, organismit) ja edelleen malleja. Nämä moduulit edustaisivat tyylioppaan sisältöä ja olisivat komponenttikirjaston käytettävissä. Kun Storybookin malleihin lisättäisiin dataa, syntyisivät näkymät, jotka esittäisivät käytännössä saman asian, kuin Playground tällä hetkellä. Näin erillistä Playground:ia ei tarvittaisi, vaan Storybook toimisi sekä tyylioppaan, että valmiiden sivujen näyteikkunan roolissa.

Vaihtoehtona keskustellaan myös tyylioppaan toteutuksesta ilman erillistä työkalua. Myös tällöin komponenttikirjaston nykyiset komponentit pilkottaisiin pienempiin, uudelleenkäytettäviin osiin, joista suurempia kokonaisuuksia koottaisiin. Ongelmaksi muodostuisi tyylikirjaston komponenttien esittely ja dokumentointi, koska varsinainen katalogi olisi koottava komponenteista erikseen ja tehtävä joko manuaalisesti tai jonkinlaista koodigeneraattoria käyttäen. Rakenne olisi kuitenkin selkeä, komponentit olisivat helposti löydettävissä ilman erillisiä kansioita tai tarvetta erilliselle selainikkunalle. Tyyliopas olisi näin Playground:in sisällä, paikassa, johon kehittäjät ovat jo valmiiksi tottuneet. Lisäksi Playground:issa jo valmiina oleva autentikointi mahdollistaisi datan hakemisen tarvittaessa. Molemmilla tavoilla tyylikomponenttien hakeminen kirjaston monimutkaisimpien komponenttien käyttöön onnistuu.

Keskustelujen ja useiden demojen pohjalta päädyttiin lopulta ratkaisuun, jossa tyyliopas perustetaan ilman erillistä työkalua komponenttikirjaston juurikansioon, ja sen visuaalinen eritys osaksi nykyistä Playground-käyttöliittymää. Tyylikomponentit luodaan projektin juureen, *components* > *UI*-kansioon. Lisäksi kohdeorganisaation kanssa sovittiin, että tyylioppaan ensimmäinen toteutus olisi suomenkielinen, huolimatta siitä, että koodi on dokumentoitu englanniksi. Organisaation nykyisen rakenteen ja käyttäjäystävällisyyden vuoksi ohjeet on hyvä olla kehittäjien äidinkielellä.

4.2.2 Tyyleihin liittyvät vaatimukset

Tyylioppaan Tyylit-osiossa käsitellään typografia, ikonit, välistys ja värit. Kuviossa 8 on esitetty tyylioppaan tyyleihin liittyvät vaatimukset.

Nro	Nimi	Vaatimuksen alustus	Vaatimuksen kuvaus	Prioriteetti
2.1	Typografia/Fontti	Kirjasintyyppi (fontti), mitä fontteja käytetään ja missä ne tulee olla määritelty? Onko ominaisuus asiakkaan räätälöitävissä?	Sierran oletusteema (SierraTheme) sisältää erilliset fontit otsikoille ja leipätekstille. Nämä fontit määritellään oletusteematiedostossa. Asiakastoteutuksessa fontit voidaan ylikirjoittaa käyttäen customTheme -tiedostoa.	Välttämätön
2.2	Typografia/Leipä	Leipäteksti: Millaiset variantit leipätekstille on määriteltävä, tekstin koko ja kirjasimen korostus? Mikä on leipätekstin väri? Onko tarvetta määritellä erikseen kirjasin- tai riviväli? Ovatko ominaisuudet asiakkaan räätälöitävissä?	Leipätekstille määritellään fontti, sen oletuskoko ja kirjasimen korostus. Tyylioppaassa esitellään erilaiset kirjasimen korostusvaihtoehdot sekä tekstin eri koot. Kirjasin- ja riviväli käyttävät Chakran oletusasetuksia. Asiakastoteutuksessa nämä ominaisuudet voidaan ylikirjoittaa käyttäen customTheme -tiedostoa. Tekstin väri voidaan määritellä teematiedostossa, muutoin käytetään Chakran oletusväriä tekstille.	Välttämätön
2.3	Typografia/Otsikko	Otsikot: Millaiset variantit otsikolle on määriteltävä,tekstin koko ja kirjasimen korostus? Mikä on otsikon väri? Onko tarvetta määritellä erikseen kirjasin- tai riviväli? Ovatko ominaisuudet asiakkaan räätälöitävissä?	Otsikolle määritellään fontti, sen oletuskoko ja kirjasimen korostus. Tyylioppaassa esitellään erilaiset kirjasimen korostusvaihtoehdot sekä otsikon eri koot. Kirjasin- ja riviväli käyttävät Chakran oletusasetuksia. Asiakastoteutuksessa nämä ominaisuudet voidaan ylikirjoittaa käyttäen customTheme -tiedostoa. Otsikon väri voidaan määritellä teematiedostossa, muutoin käytetään Chakran oletusväriä otsikolle.	Välttämätön
2.4	Ikonit/Perus	Mistä perusikonikirjastoja on käytettävissä? Miten ikonin koko ja väritys määräytyy? Miten ikoni sijoitellaan?	Sovellus käyttää pääasiassa Chakra UI:n ikoneita. Lisäksi käytössä on muutama React-ikoni. Tyylioppaassa esitellään käytössä olevat ikonit, sekä tarjotaan mahdollisuus ikonin nimen kopioimiseen leikepöydälle. Linkki React-ikonien sivulle on myös saatavilla. Ikonin koko voidaan antaa ikonia kutsuttaessa, muutoin käytetään oletuskokoa. Sijainti määritellään käyttötarkoituksen mukaan. Ikonin oletusväri on harmaa, mutta se voidaan kutsuttaessa muuttaa.	Välttämätön
2.5	Ikonit/Kuvitus	Tarvitaanko kuvitusikoneita?	Perusikonit riittävät toteutuksen tässä vaiheessa.	Mahdollinen
2.6	Välistys/Sivu	Miten sivulla olevien elementtien välinen tila määritellään? Millaiset ovat marginaalit ja täytteet. Käytetäänkö Chakran arvoja tilankäytölle vai tulisiko sovellukselle määritellä omat? Missä kohtaa ja millä tavoin sivun elementtien välistys teknisesti määritellään?	Tyylioppaassa esitellään Sierran oletusteemaan tallennetut, tätä sovellusta varten ennalta määritellyt arvot tilankäytölle. Näitä arvoja voidaan soveltaa marginaaleihin, täytteisiin ja elementtien keskinäiseen sijoitteluun spacing-arvoa käytettäessä.	Välttämätön
2.7	Värit/Brändiväri	Millainen on Sierran oletusväripaletti? Miten brändiväriä (primary) käytetään? Ovatko värit asiakkaan räätälöitävissä?	Tyylioppaassa esitellään Sierran oletusväripaletti sävyineen sekä lyhyet ohjeet värien käyttöön. Kaikki värit ovat asiakkaalle määriteltävissä customTheme -tiedostoa käyttäen.	Välttämätön
2.8	Värit/Korostus	Missä määrin korostusvärejä tarvitaan (secondary, tertiary)? Onko tilanteita, joissa väriä tarvitaan luomaan eroavaisuutta muista elementeistä? Tarvitaanko enemmän värejä (kaaviot, diagrammit, donitsit)? Onko tarvetta korostaa sovelluksessa väreillä jotain muuta?	Brändivärin lisäksi annetaan kaksi korostusväriä (secondary ja tertiary) harkittuun käyttöön. Kaavioissa ja diagrammeissa voidaan tarvita enemmän värejä, niitä varten teematiedostoon määritellään chartcolors-väripaletti.	Välttämätön
2.9	Värit/Neutraalit	Miten tekstin, taustojen ja reunusten väri ja sovelluksessa käytetyt muut harmaan sävyt määritellään?	Tyylioppaassa määritellään väripaletin lisäksi neutraalit sävyt, jota käytetään johdonmukaisesti taustoissa, tekstissä ja reunuksissa.	Välttämätön
2.10	Värit/Semantic	Millaiset liikennevalovärit teema sisältää? Käytetäänkö Chakran oletusvärejä vai määritelläänkö omat liikennevalovärit?	Tyylioppaassa määritellään väripaletin lisäksi semanttiset värit, kaksi sävyä kullekin. Värejä käytetään ilmaisemaan onnistumista, epäonnistumista, varoitusta ja infoa.	Välttämätön

Kuvio 8. Vaatimusmäärittely: tyylioppaan tyyleihin liittyvät vaatimukset.

Typografia sisältää käytettävät fontit, sekä ohjeet leipätekstin ja otsikoiden johdonmukaiseen käyttöön. Ikonien osalta vaatimuksia määritellään ainoastaan perusikoneille, koska kuvitusi-

koneita ei toteutuksen tässä vaiheessa tarvita. Välistyksen osalta määritellään tilankäyttöön ja elementtien sijoitteluun liittyvät seikat ja kuinka määriteltyjä arvoja käytetään. Värien käytölle määritellään erilaisia tarpeita, annetaan ehdotus väripalettien käytöstä ja esitellään sovelluksessa tarvittavat väripaletit. Kunkin tyylielementin osalta otetaan myös kantaa teematiedoston räätälöintiin asiakastoteutuksia varten.

4.2.3 Tyylioppaaseen sisällytettävät komponentit ja niiden vaatimukset

Keskusteluissa todetaan, ettei tyylioppaassa ole järkevää toistaa kaikkia Chakra UI:n komponentteja, vaan tulee keskittyä niihin, joita halutaan muotoilla eniten. Valtaosa Chakra UI:n komponenteista on käytössä miltei sellaisenaan ja olisi tässä vaiheessa hyödytöntä kopioida jokaista tyylioppaaseen. Toki pidemmälle mietittynä ja riippuvuuden mahdollisen purkamisen vuoksi jokaisen tarvittavan komponentin tulisi olla osana tyyliopasta. Tässä vaiheessa päätetään keskittyä tyylin kannalta olennaisiin komponentteihin.

Kuviossa 9 on esitetty tyylioppaan alustavat vaatimukset komponenttien osalta. Joukkoon on valittu vain ne komponentit, joilla arvellaan olevan eniten merkitystä yhtenäisen tyylin luomisessa. Esimerkkikomponentiksi valitaan yksiselitteisesti painikekomponentti `Button`, jonka merkitys yhtiön ilmeen kannalta on olennaisin. Toteutusvaiheessa voidaan vaatimuksia muokata, jos huomataan jonkin olennaisen komponentin puuttuvan tai jonkin olevan lisäarvoltaan tarpeeton.

Painikkeen ja typografian yhteydessä mainittujen otsikon (`Heading`) ja leipätekstin (`Text`) lisäksi mukaan valitaan vaatimusmäärittelyvaiheessa heräte (`Alert`), valikko (`Menu`), dialogi (`Menu`), ponnahdusikkuna (`Popover`) ja lisätietokomponentti `Tooltip`. Lomakkeen (`Form`) yhteydessä määritellään erilaiset syötekentät. Lisäksi vaatimuksiin kirjataan tietysti myös taulukkokomponentti (`Table`).

Graafisten, organisaatiospesifien komponenttien toteutus on olennainen osa tyylioppaan suunnittelu- ja toteutustyötä. Komponenttien monikäyttöisyys erilaisissa yhteyksissä halutaan varmistaa tutkimalla tarkkaan niiden nykyistä käyttötarkoitusta, rajapintoja ja vuorovaikutusta toisten komponenttien kanssa. Tältä osin suunnitteluun varataan enemmän aikaa.

Nro	Nimi	Vaatimuksen alustus	Vaatimuksen kuvaus	Prioriteetti
3.1	Alert	Sisältää Chakra UI -komponentit: Alert, AlertIcon, AlertDescription, AlertTitle. Tarvittavat variantit: info, success, warning, error, loading.	Toteutetaan tyylitelty AkamonAlert, joka käyttää pohjana Chakran Alert-komponenttia. Parametreina annetaan otsikko ja sisältö. Lisäksi annetaan herätteen tyyppi, joita ovat virhe, varoitus, onnistuminen, info ja tarvittaessa lataustila.	Tärkeä
3.2	Button	Sisältää Chakra UI -komponentit: Button. Tarvittavat variantit: primary, secondary, outline, ghost, link.	Painike eli Button. Ei toteuteta erillistä AkamonButton-komponenttia, varianttien toteutus tehdään teematiedostossa, joka on muunneltavissa asiakastoteutuksissa. Varianttien lisäksi tyylioppaassa tulee esittää painikkeen lataustila sekä estetty tila.	Välttämätön
3.3	Form	Sisältää Chakra UI -komponentit: FormControl, FormLabel, FormErrorMessage, FormHelperText.	Toteutetaan tyylitelty AkamonForm, joka käyttää pohjana Chakran FormControl-komponenttia. AkamonForm toimii lomakegeneraattorina, joka saa lomakkeen kentät objektina. Jokainen kenttä sisältää tiedot: otsikko, kenttätyyppi, paikanvaraaja ja oletusarvo. Lisäksi komponentille annetaan parametrina painikkeen teksti. Lomakegeneraattori tunnistaa kenttätyypin ja laatii sen perusteella oikeanlaisen syötekentän.	Välttämätön
3.4	Heading	Sisältää Chakra UI -komponentit: Heading.	Heading-komponentti käyttää suoraan Chakran Heading-komponenttia ja sen muotoilut asetetaan teematiedostossa. Erillistä AkamonHeading-komponenttia ei tarvita. Heading esitellään tyylioppaassa typografian alla.	Välttämätön
3.5	Input	Sisältää Chakra UI -komponentit: Input. Tarvittavat variantit: text, email, password, number, textarea, checkbox, radio, select.	Erilaiset syötekentät esitellään tyylioppaassa. Ei ole tarvetta luoda erillistä AkamonInput-komponenttia. Input-komponentit käyttävät Chakran Input-komponenttia sekä tarvittaessa Select-komponenttia. Jos syötekentän ulkoasua on tarvetta muokata, se tapahtuu teematiedostossa.	Tärkeä
3.6	Menu	Sisältää Chakra UI -komponentit: Menu, MenuButton, MenuList, MenuItem. Käytössä Selectin tapaan (tarvitaanko molempia?).	Luodaan tyylitelty AkamonMenu, joka toimii Select-komponentin tapaan, mutta on ulkoasultaan vapaammin muotoiltavissa. Menun pohjana toimii Chakran Menu-komponentti. Menun valintavaihtoehdot annetaan objektijoukkona.	Tärkeä
3.7	Modal	Sisältää Chakra UI -komponentit: Modal, ModalOverlay, ModalContent, ModalHeader, ModalFooter, ModalBody, ModalCloseButton.	Luodaan tyylitelty AkamonModal, jonka pohjana toimii Chakran Modal-komponentti. AkamonModal ottaa vastaan seuraavat tiedot: otsikko, sisältö (vaihteleva), alaviite (vaihteleva) sekä toimintopainikkeet, jotka annetaan objektijoukkona.	Välttämätön
3.8	Popover	Sisältää Chakra UI -komponentit: Popover, PopoverTrigger, PopoverContent, PopoverHeader, PopoverBody, PopoverFooter, PopoverArrow, PopoverCloseButton	Ponnahdusikkunasta luodaan oma tyylitelty AkamonPopover-komponentti, joka painiketta painamalla. Komponentti ottaa vastaan seuraavat tiedot: otsikko, sisältö ja objektijoukkona annettavat toimintopainikkeet.	Tärkeä
3.9	Table	Sisältää Chakra UI -komponentit: Table, Thead, Tbody, Tr, Th, Td, TableCaption, TableContainer. Variantit: simple ja striped.	Luodaan AkamonTable-komponentti datan esitystä varten. AkamonTable ottaa vastaan datan objektijoukkona sekä vapaaehtoisen kuvaustekstin ja käyttää pohjana Chakran Table-komponenttia. Muotoilut tapahtuvat AkamonTable-komponentin yhteydessä. Luodaan lisäksi AkamonBrandTable käytettäväksi tarvittaessa (sopimukset, profiilitiedot).	Välttämätön
3.10	Text	Sisältää Chakra UI -komponentit: Text.	Text-komponentti käyttää suoraan Chakran Text-komponenttia ja sen muotoilut asetetaan teematiedostossa. Erillistä AkamonText-komponenttia ei tarvita. Text-komponentti esitellään typografian alla.	Välttämätön
3.11	Tooltip	Sisältää Chakra UI -komponentit: Tooltip.	Lisätiedon antamista varten luodaan oma tyylitelty AkamonTooltip-komponentti, joka avautuu viemällä hiiri teeman pääväreillä muotoillun infoikonin päälle. Komponentti ottaa vastaan infotekstin kutsun yhteydessä.	Tärkeä

Kuvio 9. Vaatimusmäärittely: tyylioppaan komponentteihin liittyvät vaatimukset.

4.2.4 Malleihin liittyvät vaatimukset

Kuviossa 10 on esitetty tyylioppaan sivumalleihin liittyvät vaatimukset. Määritellään sivulle raamit ja se, miten sisältö muodostuu. Tyylioppaassa esitellään esimerkinomaisesti erilaisia

sivumalleja, joiden responsiivista esitystapaa voi kokeilla pienentämällä ruutukokoa.

Nro	Nimi	Vaatimuksen alustus	Vaatimuksen kuvaus	Prioriteetti
4.1	Mallit / Sivu	Mistä osista sivu koostuu ja miten raamit muodostuvat?	Sivua varten luodaan erityinen komponentti, malli, jota jokainen sovelluksen sivu noudattaa. Malli sisältää otsikon ja sisältöosion. Mallin yhteydessä määritellään otsikon ja sisällön keskinäinen välistys sekä sivumarginaalit käyttäen tyylioppaassa esiteltyjä arvoja.	Välttämätön
4.2	Mallit / Sisältö	Miten sivun sisältöosio rakennetaan? Sivuilla voi olla hyvin erilaista sisältöä ja erilainen määrä elementtejä. Kuinka tämä mahdollistetaan?	Sivun sisältöä varten luodaan erityinen komponentti, jota käyttämällä sivu voidaan koostaa dynaamisesti yhdestä tai useammasta sisältöosiosta. Elementtien keskinäisestä välistyksestä huolehtivat sivu- sekä sisältömalli.	Välttämätön

Kuvio 10. Vaatimusmäärittely: tyylioppaan malleihin liittyvät vaatimukset.

4.2.5 Tyylinvalvojan vaatimukset

Tyylinvalvojalle asetetut vaatimukset on esitetty kuviossa 11. Vaatimukset kattavat tyylinvalvojan sijainnin, toiminnan ja toteutuksen lisäksi vaatimuksia mm. testauksen ajoitukseen, virhemarginaaliin ja koodikattavuuteen liittyen. Vaatimusmäärittelyn yhteydessä asetetaan tyylinvalvojalle myös hyväksymiskriteerit.

Nro	Nimi	Vaatimuksen alustus	Vaatimuksen kuvaus	Prioriteetti
5.1	Sijainti	Missä tyylinvalvoja sijaitsee suhteessa komponenttikirjastoon?	Tyylinvalvoja sijoitetaan komponenttikirjaston juurikansioon, jossa toiminnallinen testaus myös tapahtuu. Sekä tyylikomponenttien että playgroundissa sijaitsevan esimerkkisivun testaus onnistuu tällä tavalla.	Välttämätön
5.2	Toiminta	Mitä asioita tyylinvalvojan tulee käyttöliittymästä tarkastaa? Sisällytetäänkö myös CSS-koodin testausta?	Tyylioppaan komponeneista kootaan esimerkkisivu, jolle suoritetaan visuaalinen regressiotestaus. Komponenteista tarkistetaan niiden asemointi, koko, väri, tekstisisältö sekä muut layoutin ja tyylin muutokset. CCS-testausta ei tähän toteutukseen sisällytetä.	Välttämätön
5.3	Työkalu	Mikä on tekninen toteutus? Mitä valmiina löytyviä työkaluja otetaan vertailuun? Onko syytä yrittää rakentaa oma, tekoälyyn / koneoppimiseen perustuva ratkaisu?	Koska valmiita testaustyökaluja on olemassa runsaasti, otetaan käyttöön jokin niistä. Keskitytään snapshot-tekniikalla toimiviin avoimen lähdekoodin työkaluihin. Valitaan vertailuun kolme ohjelmistoa ja valitaan niistä sopivin. Valinnassa huomioidaan myös se, että työkalun on tarvittaessa sovelluttava myös toiminnalliseen testaukseen.	Välttämätön
5.4	Ajankohta	Missä vaiheessa/vaiheissa kehitystä tyylinvalvonta tapahtuu? Kuinka usein?	Testaus tulee suorittaa aina, kun sovellukseen tehdään muutoksia.	Välttämätön
5.5	Toleranssi	Minkä verran virheitä tulisi sallia, jotta kehitystä voi jatkaa?	Vertailuvaiheessa työkalun herkkyydestä riippuen asetetaan virheille toleranssi, jotta visuaalinen yhtenäisyys säilyy. Lopullisen työkalun valinnan myötä arvioidaan, kuinka tarkka pikselivertailun tulee olla valitulla työkalulla.	Välttämätön
5.6	Koodikattavuus	Kuinka paljon koodista on oltava testattu, että se voidaan hyväksyä?	Koodikattavuutta ei mitata tässä vaiheessa, koska toteutetaan vain esimerkkitestit. Mittari on kehityksen kannalta hyödyllinen mutta jää työn seuraaviin vaiheisiin.	Ei toteuteta
5.7	Hyväksyminen	Mitkä ovat toteutetun tyylinvalvojan hyväksymiskriteerit, jotta tutkimuksen lopuksi voidaan sanoa sen toimivan toivotulla tavalla?	Tämän tutkimuksen puitteissa toteutetaan sellainen osa testausta, jota voidaan käyttää esimerkkinä työn jatkamiseen. Toteutetaan testaus muutamalle tyylioppaassa tutkimushetkellä olevalle komponenteille.	Välttämätön

Kuvio 11. Vaatimusmäärittely: tyylinvalvojan vaatimukset.

4.3 Suunnittelusykli: tyylioppaan toteutus

Vaatimusmäärittelyyn perustuen tyyliopas toteutetaan ilman erillistä työkalua osaksi Playground-käyttöliittymää. Toteutus aloitetaan sivuvalikon muutoksilla. Käyttöliittymän sivuvalikkoon lisätään väliotsikot `UI-Komponentit`, `UI-Tyylit` ja `UI-Mallit`, jotka sisältävät kaikki tyylioppaan osat. Playground:in sivut löytyvät edelleen `Sierra-components`-valikosta.

4.3.1 Komponentit

Jotta komponenttien tunnistus valikosta olisi kehittäjille helppoa, niiden nimet jätetään englannin kielelle. Tyylioppaan sivut halutaan pitää yksinkertaisena ja keskenään samanlaisina. Sivun otsakkeen alla on lyhyt kuvaus komponentin tarkoituksesta. Kohdeorganisaation toive on, ettei liian pitkiä dokumentaatioita lisättäisi, koska esimerkit puhuvat paremmin puolestaan. Kehittäjät haluavat löytää hakemansa nopeasti ja visuaalinen versio edesauttaa tätä paremmin. Komponenttien sivulla on kolme välilehteä: `Esimerkit`, joka esittelee kaiken tarpeellisen komponenttiin liittyvän tekstein ja visuaalisessa muodossa, `Propsit`, jossa on taulukkomuodossa esitettynä kaikki komponentilla olevat ominaisuudet sekä `Parametrit`, jossa esitellään komponentin organisaatiospesifit parametrit. Kunkin esimerkin jälkeen käyttäjän on mahdollisuus avata lähdekoodi ja kopioida se leikepöydälle.

`Button`. Esimerkkikomponentiksi valitun `Button`-komponentin luominen aloittaa tyylioppaan sisällön muodostuksen. `Button`:ista esitellään variantit, lataustila ja estetty tila. `Button`:in osalta ajatus organisaatiospesifistä komponentista ei toimi. On helpompi soveltaa Chakra:n `Button`-komponenttia sellaisenaan ja muodostaa siihen erilaisia variantteja käyttöä varten. Tällainen lähestymistapa on kohdeorganisaation toive, vaikka se rikkoo yhtenäisen linjan toteutustavan suhteen. Alun perin ajatuksena oli mahdollisuus tarvittaessa purkaa vahva Chakra-riippuvuus ja tämä tapa ei tue tuota periaatetta. Chakra:n tarjoama varianttimenettely on kieltämättä toimiva ja tällä voidaan välttää jokaiselle erilaiselle painikkeille oman komponentin luominen. Varianttien muotoilu toteutetaan ohjelmiston teematiedostossa. Kullakin variantilla on nimi, joka annetaan komponentin kutsun yhteydessä. Teematiedosto huolehtii, että variantti on halutunlainen. Variantteja ovat `primary`, `secondary`, `outline`, `ghost` ja `link`. Lataus- ja estetty tila näytetään kaikille varianteille. Teematiedosto on muokattavissa, jolloin myös painikkeille saadaan haluttaessa asiakkaan toiveiden mukainen ulkoasu.

`Input ja Form`. Eräs kohdeorganisaatiolle päänvaivaa aiheuttanut seikka on ollut lomak-

keiden epäyhtenäinen ulkoasu. Tyylioppaassa esitellään erilaiset syötekentät sellaisessa muodossa, kuin niiden halutaan sovelluksissa esiintyvän. `Input`-sivulla on vain esimerkinomainen listaus saatavilla olevista kentistä lähdekoodeineen. `Form`-sivulla sen sijaan esitellään myös `AkamonForm`, komponentti, jolla lomakkeen muodostus tapahtuu dynaamisesti. React-hook-form:illa toteutettu lomakekomponentti käyttää Zod-kirjaston avulla toteutettua skeemaa. Lomakkeen kentät ja niihin liittyvät validointitiedot annetaan Zod-objektina. Käytettäessä aina samaa lomakkeenmuodostuskomponenttia, lomakkeet kautta sovelluksen ovat yhtenäisen näköisiä. Tässä tullaan jo lähemmäs sitä alkuperäistä ajatusta organisaatiospesifeistä komponenteista. Haluttaessa kenties joskus kauemmas Chakra UI:sta, muutokset tehtäisiin keskitetysti lomakegeneraattoriin, eikä jokaista instanssia tarvitsisi muokata erikseen.

`Modal`. Sivulle avautuva dialogi, `Modal`, on myös eräs epäyhtenäistä ilmettä aiheuttava asia, koska se on ahkerasti käytössä sovelluksen sivuilla. Tämän komponentin osalta halutaan myös luoda `AkamonModal`, joka näyttää aina samalta ja toimii halutulla lailla. Primääripainikkeen painallus saa aikaan dialogin avautumisen. Dialogi sisältää aina samat elementit: otsikon, sulkupainikkeen, sisällön ja alaviitteen painikkeineen. `AkamonModal` on komponentti, jolle tarvittavat tiedot annetaan kutsuvaiheessa. Komponentti luo aina samanlaisen dialogin huolimatta vaihtelevasta sisällöstä. Johdonmukaisuus käyttöliittymäkomponenttien toiminnassa luo selkeyttä ja opittavuutta, mitkä vaikuttavat merkittävästi käyttäjäkokemukseen.

`Table`. Myös taulukosta luodaan oma, pelkistetty `AkamonTable`-komponenttinsa, joka esittää datan aina samanlaisessa muodossa. Sarakeotsikot käyttävät isoja kirjaimia ja brändiväriä, solut on tasattu vasemmalle. Taulukon responsiivinen muoto esittää informaation korttimuodossa. Taulukon taustalla on jälleen generaattori, jolle taulukon sisältö annetaan objektijoukkona. Sovelluksessa tarvitaan kuitenkin myös toisenlainen taulukkomalli, joka komponenttikirjastossa jo on olemassa. Tästä `BrandTable`:sta muokataan vielä spesifi `AkamonBrandTable`, joka on tarkoitettu esimerkiksi sopimus- tai profiilitietojen näyttämiseen, tarkoitukseen, jossa yhden asian tai henkilön tietoja näytetään kerrallaan. Taulukolla on otsake, sarakenimet vasemmalla ja tiedot oikealla. Tyylioppaan sivulla esitellään myös tämä taulukkomuoto. Kuitenkin ehkä käyttökelpoisin taulukkomuoto on `AkamonDatatable`, jolla voidaan esittää suuria määriä dataa. Komponentti hyödyntää TanStack-kirjastoa taulukon muodostuksessa ja sisältää suodatus- ja sivutusominaisuudet.

`Alert`. Käyttöliittymässä on usein tarpeellista informoida käyttäjää sovelluksen tilasta. Tähän tarkoitukseen käytetään herätteitä. Tyylioppaassa luodaan `AkamonAlert`-komponentti, joka

on valmiiksi tyylitelty komponentti kaikenlaisten herätteiden käyttöön. Herätteitä ovat virhe, varoitus, onnistuminen, info ja tarvittaessa sitä voidaan käyttää myös lataustilasta ilmoittamiseen. `AkamonAlert` käyttää teematiedostossa annettuja semanttisen paletin värejä, jotka perustuvat herätteen tyyppiin. Nämä värit ovat muokattavissa asiakkaan omassa teemassa.

`Menu`. Valintakomponentti eli `Menu` soveltuu alasvetovalikoiden käyttöön varsinaisen `Select`-komponentin tapaan. Valintakomponentille annetaan vaihtoehdot, joista käyttäjä valitsee yhden. `Select`-komponentti on tyylimuotoilultaan hyvin rajoittunut, siksi `Menu` soveltuu paremmin tilanteeseen, jossa käyttöliittymän yhtenäistä ilmettä haetaan. Tyylioppaassa luodaan `AkamonMenu`-komponentti, joka on ulkoasultaan brändin mukainen. Vaihtoehdot valintaa varten annetaan objektijoukkona.

`Popover`. Ponnahdusikkunalla voidaan käyttäjän huomio kiinnittää hetkeksi tiettyyn asiaan. Tyylioppaassa luodaan `AkamonPopover`-komponentti, joka sisältää otsikon, sulkupainikkeen, vaihtelevan sisällön sekä alaviitteen painikkeineen. Ponnahdusikkunan avaus tapahtuu primäärimuotoillusta painikkeesta. Komponentti saa tarvittavat tietonsa kutsuvaiheessa.

`Tooltip`. Lisätietoikkuna eli `Tooltip` antaa käyttäjälle lisätietoa sovelluksen toiminnasta. Tyylioppaassa luodaan `AkamonTooltip`-komponentti, jossa lisätieto avautuu infoikonin päälle siirryttäessä. Ikonin väritys vastaa pääväriä ja harmaalla pohjalla oleva valkoinen infoteksti avautuu ikonin alapuolelle. Infoteksti annetaan komponentille kutsuvaiheessa.

4.3.2 Tyylit

Tyylioppaan Tyylit-osiossa määritellään yleisiä visuaaliseen ilmeeseen vaikuttavia seikkoja, kuten typografia, värit, välistys ja ikonit. Millaisin variantein otsikoita ja leipätekstiä tulee käyttää ja miten värien käyttö tulee teknisesti toteuttaa, niin että se on tehokkaasti räätälöitävissä asiakastoteutuksiin? Välistyksellä luodaan vaikutelma harmoniasta ja myös tähän haetaan ratkaisua.

Typografia. Typografia sisältää kirjasintyypit, niiden koot ja muotoilut. Tyylioppaassa esitellään organisaation typografiateema, sisältäen kaksi fonttia `Sans-serif`-fonttiperheestä, jotka on määritelty sovelluksen teematiedostossa. Tekstityypeistä esitellään otsikot ja leipäteksti. Otsikoissa käytetään kirjasinta `Ridley Grotesk Extra Bold` ja brändiväriä. Otsikon osalta esitellään kirjasimen paksuus- ja kokovaihtoehdot. Leipätekstin fontti on `Aaux Next`. Samoin leipätekstille on olemassa erilaisia kirjasimen paksuus- ja kokovaihtoehtoja. Molem-

pien tekstityyppien osalta esitellään myös kirjasimen oletustyylit. Kirjasin- sekä rivivälien osalta ei esitellä erilaisia vaihtoehtoja, ne seuraavat Chakra:n `Heading`- ja `Text`-komponenttien oletusasetuksia. Typografiaan liittyvät tyylit ovat muokattavissa asiakaskohtaisessa teematiedostossa. Kirjasimen väritykseen otetaan kantaa tyylioppaan Värit-sivulla.

Värit. Väreillä on tärkeä rooli yhtiön visuaalisessa ilmeessä. Ne luovat persoonallisuutta, herättävät käyttäjän huomion ja parantavat luettavuutta. Tämän tyylioppaan tarkoitus ei ole luoda uutta värimaailmaa, vaan esitellä olemassa olevat värit ja antaa ohjeita niiden käyttöön. Väripaletti on poikkeuksetta asiakaskohtainen, jolloin tämän leikkikenttäsovelluksen värityksillä ei ole niin valtavan suurta merkitystä. Värien suhteilla toisiinsa sen sijaan on. Jotta asiakkaan toiveiden mukainen väritys on helposti toteutettavissa, on määriteltävä tarkkaan, kuinka sovelluksessa pää- tai korostevärejä käytetään. Tyylioppaassa esitellään primääri- ja sekundääripalettien lisäksi neutraali väripaletti sekä semanttiset, kommunikointiin tarkoitetut värit (mm. liikennevalovärit).

Välistys. Erittäin tärkeä osa tyylikästä käyttöliittymää on elementtien asettelu sivulle sekä niiden keskinäinen välistys. Tämä on yksi suurimmista ongelmista tämän hetken sovelluksessa, jossa välistyksiin ei ole kiinnitetty riittävästi huomiota. Tyylioppaassa luodaan yksinkertaiset ennalta määritellyt arvot tilankäytölle. Näitä teematiedostoon tallennettuja arvoja voidaan käyttää marginaaleissa, täytteissä sekä elementtien väleissä. Myöhemmin kuvattavat sivumallit hyödyntävät juuri näitä välistysarvoja, jotta yleisilme pysyy harmonisena.

Ikonit. Kuvakkeilla eli ikoneilla voidaan ohjata käyttäjää tietyissä toiminnoissa ja helpottaa laajojen kokonaisuuksien hahmottamista. Tyylioppaan ikonisivulla esitellään kaikki Chakra UI -ikonit sekä muutama jo käytössä oleva React-ikoni. Ikonia painettaessa nimi kopioituu leikepöydälle, josta se on helppo siirtää sovellukseen. Sivulla tarjotaan myös linkki React Icons -sivulle, siltä varalta, ettei sopivaa ikonia esitellystä joukosta löydy.

4.3.3 Mallit

Monesti käyttöliittymä sisältää useita keskenään erilaista sisältöä esitteleviä sivuja. Sivujen keskinäinen yhteys ja hyvä käyttäjäkokemus rakennetaan sivumalleilla, jotka toistuvat läpi sovelluksen. Tyylioppaassa luodaan `AkamonPageTemplate`, joka muodostaa sivun raamit. Komponentti sisältää pinorakenteeseen (`Stack`) sijoitetun, horisontaalisesti keskitetyn otsikon ja sisältöosan. Kutsuttaessa komponenttia sille voidaan antaa yksi välistysarvo, jota käytetään sivumarginaalien lisäksi `Stack`-komponentin `Spacing`-arvona. Jos välistysarvoa ei

erikseen anneta, käytetään ennalta määriteltyä oletusarvoa. Varsinaista sisältöä varten luodaan `AkamonPageSection`-komponentti, joita voidaan sivulle asetella joko yksi tai useita. Tämä komponentti vastaa yhden elementtirivin sisällöstä ja on rakennettu edelleen `Stack`-komponenttia käyttäen. `AkamonPageSection` ottaa vastaan yhden tai useamman elementin ja on responsiivinen siten, että pienemmillä näytöillä rivin elementit muotoutuvat sarakkeeksi. Kun sivun sisältö rakentuu erikseen määriteltävistä riveistä, voidaan paremmin kontrolloida sisällön sijoittumista sivulle erikokoisilla näytöillä. Tällainen malli taipuu paremmin myös keskenään hyvin erilaisten asiakastarpeiden hallintaan.

4.4 Suunnittelusykli: komponenttikirjaston muutokset

Komponenttikirjastossa on rekonstruktion hetkellä 251 komponenttia, kun niitä tutkimuksen alussa oli 174. Kirjasto siis elää jatkuvasti, mikä luonnollisesti vaikeuttaa työtä. Jotta tyylikomponentit voitaisiin ottaa käyttöön, täytyy ensin merkitä ne komponenttikirjaston kohdat, joissa vastaavia komponentteja on käytössä. Tämän jälkeen instanssi kerrallaan arvioidaan, kuinka tyylikomponentin käyttöönotto on mahdollista kyseisessä tapauksessa. Monet kirjaston komponenteista sisältävät tyylimuotoiluja, joita ei luonnollisesti haluta siirtää monikäyttöiseen tyylikomponenttiin, koska tavoitteena on yhtenäinen ilme. Sen sijaan tyylikomponenttia täytyy muokata, jos huomataan käyttötarkoituksen vaativan esimerkiksi lisää parametreja tai ominaisuuksia, joita ei ole osattu huomioida tyylikomponenttia luotaessa. Ongelmaksi voi muodostua se, että komponentteja käytetään hyvin erilaisiin tarkoituksiin. Riittävän universaalin mallin rakentaminen voi olla haasteellista, jolloin täytyy harkita, onko tyylikomponentti syytä eriyttää useammaksi erilaiseksi versioksi.

Komponenttikirjaston yhteyteen luodaan kansio *reconstructionexamples*, johon kootaan joukko esimerkkinä käytettyjä komponentteja muutoksineen. Koska täydellistä rekonstruktiota ei tämän tutkimuksen puitteissa ole mahdollista tehdä, kohdeorganisaatio voi halutessaan käyttää annettuja esimerkkejä ja jatkaa työtä laadittujen ehdotusten mukaan. Käsitellään seuraavissa kappaleissa kutakin tyylikomponenttia erikseen edellä mainitut asiat silmällä pitäen.

4.4.1 Komponentit

`Alert`. Chakra UI:n `Alert`-komponentti on käytössä neljässä kirjaston komponenteista, joista yksi on olemassa oleva, monikäyttöiseksi tarkoitettu malli. Tätä mallia kutsuu edelleen 10 muuta komponenttia. Chakra UI:n `Alert`:ia käyttävät komponentit on esitetty taulukossa 1.

Komponentin sijaintipolku	Komponentin nimi
src/components/	Contracts
src/components/...	ConsumptionTimeRange
playground/src/components/	Login
src/components/	Alert

Taulukko 1. Chakra:n `Alert`-komponenttia käyttävät komponentit.

Esimerkkinä muokataan kirjastossa oleva `Alert`-komponentti käyttämään tyylikomponenttia `AkamonAlert` koodilohkoissa 1 ja 2 esitetyllä tavalla, jolloin siihen sidoksissa olevien komponenttien tulisi korjautua kerralla. Toinen vaihtoehto olisi muokata jokainen `Alert`-komponenttia kutsuva esiintymä kutsumaan `AkamonAlert`-komponenttia. Jälkimmäinen vaatii enemmän työtä, mutta olisi jatkon kannalta suoraviivaisempi tapa ja turhat komponenttikutsut jäisivät välistä pois.

```
...
      <ChakraAlert w={'fit-content'} {...rest}>
        <AlertIcon />
        <AlertTitle>{title}</AlertTitle>
        <AlertDescription>{description}</AlertDescription>
      </ChakraAlert>
...
```

Koodilohko 1. Korvattava `Alert`-esiintymä.

```
...
      <AkamonAlert
        title={title}
        description={description}
        {...rest} />
...
```

Koodilohko 2. `Alert`-esiintymän korvaava koodi.

Pohdittavaksi jää vaatiiko jokin kirjaston `Alert`-komponenttia kutsuvasta 10 komponentista muutoksia tyylikomponenttiin, vai riittävätkö `Alert`-isäntäkomponenttiin tehdyt muutokset. Nämä `Alert`-komponenttia kutsuvat komponentit on esitetty taulukossa 2.

Komponentin sijaintipolku	Komponentin nimi
src/components/.../	AveragePriceContent
src/components/.../	ContractProductListing
src/components/.../	MessagesTableWrapper
src/components/	Messages
src/components/	ProductComparison
src/components/	Profile
src/components/.../	ConsumptionChange
src/components/.../	ConsumptionDiagram
src/components/.../	NormalizedHeatingEnergyMonthly
src/components/.../	NormalizedHeatingEnergyYearly

Taulukko 2. Kirjaston `Alert`-komponenttia kutsuvat komponentit.

`Button`. Chakra UI:n `Button`-komponentti on käytössä kirjaston 66 komponentissa, joissa on esiintymiä yhteensä 104. Painikkeen rekonstruktio toimii eri tavalla kuin muiden tyylikomponenttien, koska painikkeesta ei päätetty tehdä organisaatiospesifiä komponenttia. Sen sijaan toimitaan tyylitiedostoon kirjoitetuilla varianteilla. Kirjaston painike-esiintymät käyttävät koodin yhteydessä annettuja variantteja, jotka kuitenkin tyylioppaan puitteissa on määritelty uudestaan. Jos varianttia ei ole annettu, käytetään teematiedostoon määriteltyä oletusta. Monessa esiintymässä käytetään lisäksi erilaisia muotoiluja, jotka saattavat häiritä yhteistä ilmettä. Tutkitaan jokaista esiintymää kerrallaan ja määritellään tarvittavat muutokset. `Button`-komponentin esiintymät lähdekoodeineen on koottu erilliseen tiedostoon jatkojalostusta varten (*Button-instanssit-AkamonSierra.xlsx*). Alla olevassa koodilohkossa 3 on esimerkki eräästä `Button`-komponentin esiintymästä (*ContractProductCalculator.tsx*).

```
...
      <Button
        height={45}
        w={{ base: '100%' }}
        bgColor="brand.700"
        color="white"
        fontSize="xl"
        fontWeight={400}
        _hover={{ color: 'white', bg: 'brand.300' }}
```

```
        _active={{ color: 'white', bg: 'brand.300' }}
        _focus={{
          boxShadow: 'none',
          color: 'white',
          bg: 'brand.300',
        }}
        onClick={onClickReturn}
      >
        Takaisin tuotevalintaan
      </Button>
...
```

Koodilohko 3. `Button`-esiintymä.

Painikkeen leveys on yleisesti annettu ominaisuus, yleensä haetaan täyttä leveyttä. Kannattaa pohtia, olisiko hyötyä variantista, jolle olisi asetettu täysi leveys jo valmiiksi. Oletuksena painike ottaa tekstinsä vaatiman leveyden. Jos halutaan painikkeen täyttävän koko käytössä oleva tila, leveys on annettava erikseen. Painikkeen variantit on kirjattu tiedostoon *akamon-chakra-base-theme.ts* ja niitä voidaan tarpeen vaatiessa muokata tai luoda lisää.

`Form`. Lomakkeita esiintyy kirjaston 19 komponentissa, joista kahdella on edelleen riippuvuuksia muualla. `FormInput`-komponenttia käyttää 2 komponenttia ja `FormRadio`-komponenttia 5 muuta kirjaston komponenttia. Tyylioppaan lomakegeneraattorin sopivuus täytyy tarkistaa komponentti kerrallaan. Tavoitetila on, että kaikki sovelluksen lomakkeet voitaisiin luoda yhteistä lomakegeneraattoria käyttäen eikä erillisiä monikäyttöiseksi tarkoitettuja lomakkeen osia tarvittaisi. Chakra UI:n `Form`:ia käyttävät komponentit on esitetty taulukossa 3.

Komponentin sijaintipolku	Komponentin nimi
playground/src/components/	AveragePrice
src/components/	BasicInformation
src/components/.../	AutomaticSignUpCheckbox
src/components/.../	ContractInformation
src/components/.../	TermsOfSale
src/components/.../	ConsumptionInputs
src/components/.../	ContractTerminationForm
src/components/.../	DatePickerCustomInput
src/components/.../	YearlySavings
src/components/.../	EtrmFormInput
src/components/.../	EtrmManualGsrnSelectModal
src/components/.../	EtrmMeteringPointFormSectionConsumption
src/components/.../	EtrmOfferSummaryCardSendToEmailModal
src/components/.../	FormInput
src/components/.../	FormRadio
playground/src/components/	Login
src/components/.../	MessageForm
src/components/.../	NewContactModal
src/components/	ProductComparison

Taulukko 3. Chakra:n `Form`-komponenttia käyttävät komponentit.

Esimerkkinä muokataan `NewContactModal`-komponentin `Form`:ia käyttävää osuutta alla olevan koodilohkon 4 mukaan (lähdekoodin rivimäärän vuoksi lohkossa näytetään vain muutettu osuus). Komponentin käyttämä skeematiedosto löytyy *UI/schemas*-kansiosta nimellä *NewContactSchema.ts*. Tyylikomponenttiin tarvittavat lisäparametrit on esitetty jäljempänä taulukossa 4.

```
...
      <AkamonForm
        buttonText={t('button:submit')}
        columns={2}
        onFormSubmit={() => {
          console.log('add handleSubmit-method here');
        }}
```

```
          schemadata={{
          validationSchema,
          fieldMetadata,
        }}
      />
...
```

Koodilohko 4. `Form`-esiintymän korvaava koodi.

Parametri	**Selitys**
`tschemadata: validationSchema: T; fieldMetadata: Record<string, any>;`	Skeeman tuonti oli aiemmin rakennettu `AkamonForm`:in sisälle. Nyt se haetaan komponenttia kutsuttaessa ja välitetään parametrina lomakegeneraattorille.

Taulukko 4. `AkamonForm`-komponenttiin tarvittavat lisäparametrit.

Muutosten yhteydessä erilaisten skeemojen käyttö on mahdollistettu `AkamonForm`-komponentin geneerisellä tyypillä. Näin TypeScript osaa dynaamisesti päätellä tarvittavat tyypit skeeman perusteella. Myös `renderFormControl`-komponentin parametreihin ja niiden tyyppeihin on tehty muutoksia, jotta erilaisten skeemojen käyttö on mahdollista. Lisäksi `Input`-tyyppi `switch` on lisätty `renderFormControl`-komponenttiin.

Pohdittavaksi jää, tarvitaanko olemassa olevien lisäksi muita `input`-tyyppejä. Tarpeellisia voisivat olla ainakin liitteen lataaminen (`file`) ja liukusäädin (`range slider`). Tarvittaessa lisätään erilaisia tyyppejä `renderFormControl`-komponentin `case`-listaan. `onSubmit` -tapahtuman käynnistää `AkamonForm`:illa oleva painike. Jos lomake on sisällytetty dialogiin, onko tarkoitus, että dialogi sulkeutuu lomakkeen lähetyksen yhteydessä? Esimerkkikomponentissa lomakkeen lähetys on liitetty modaalin toimintopainikkeisiin. Olisiko lomake monikäyttöisempi, jos se ei sisältäisi lähetyspainiketta? Lisäksi `AkamonForm`:illa olevan peruutuspainikkeen teksti on kovakoodattu. Sen voisi tarvittaessa tuoda myös parametrina samaan tapaan kuin lähetyspainikkeen tekstin. Tyylioppaan toisen evaluoinnin jälkeen lomakekomponenttia on muutettu edelleen niin, että skeematiedosto on entistä geneerisempi ja sisältää tiedon tarvittavista sarakkeista.

`Menu`. Kirjaston komponenteista 3 käyttää Chakra UI:n `Menu`-komponenttia. Pohtia kannattai-

si lisäksi, voisiko joitakin `Select`-komponentteja muuntaa `AkamonMenu`-komponenteiksi, jolloin tyyli noudattaisi paremmin yhtenäistä ilmettä. `Select`-komponentti on käytössä kirjaston 12 komponentissa ja sen muotoilu on huomattavasti rajoitetumpaa, kuin `Menu`-komponentin. Chakra UI:n `Menu`:a käyttävät komponentit on esitetty taulukossa 5 .

Komponentin sijaintipolku	Komponentin nimi
src/components/.../	AsyncMeteringPointList
src/components/.../	EnergyTransitionMeteringPointList
playground/src/components/.../	LanguageSelector

Taulukko 5. Chakra:n `Menu`-komponenttia käyttävät komponentit.

Esimerkkinä muokataan `LanguageSelector`-komponenttia alla olevien koodilohkojen 5 ja 6 mukaan. Tyylikomponenttiin tarvitaan lisäparametreja, jotka on esitetty jäljempänä taulukossa 6. Samalla lisätään tyylimuotoilu `textTransform: capitalize` painikkeen sekä valintavaihtoehtojen tekstiin, jotta tekstit alkavat isolla kirjaimella. Esimerkkikomponentissa on lisäksi tyylimuotoiluja, jotka tässä muutoksessa on jätetty toteuttamatta. Näitä ovat ikonien väritys ja painikkeen kulmien pyöristykset.

```
...
      <Menu colorScheme="purple">
        <MenuButton
          w="full"
          as={Button}
          variant="outline"
          borderRadius="sm"
          leftIcon={
        <MdOutlineLanguage color={light || 'white'} />
          }
          color={color || 'gray.700'}
          rightIcon={
        <ChevronDownIcon color={light || 'white'} />
          }
         >
          {languages.find((lng) =>
```

```
          lng.value === selectedLocale)?.label}
        </MenuButton>
        <MenuList>
          {languages.sort().map((lng) => (
            <MenuItem
              isDisabled={lng.value === selectedLocale}
              key={lng.value}
              onClick={() => handleLocaleChange(lng.value)}
              style={{ textTransform: 'capitalize' }}
            >
              {lng.label}
            </MenuItem>
          ))}
        </MenuList>
      </Menu>
...
```

Koodilohko 5. Korvattava `Menu`-esiintymä.

```
...
      <AkamonMenu
        menuOptions={optionsList}
        buttonWidth={'full'}
        buttonState={
          languages.find((lng)
          => lng.value === selectedLocale)?.label
        }
        leftIcon={<MdOutlineLanguage />}
        variant={'outline'}
      />
...
```

Koodilohko 6. `Menu`-esiintymän korvaava koodi.

Parametri	Selitys
`buttonWidth?: string`	Valinnainen painikkeen leveys, jos halutaan esimerkiksi koko raamin levyinen painike. Oletuksena painikkeen leveys mukautuu tekstin leveyden mukaan.
`buttonState?: string`	Valinnainen painikkeessa aloitushetkellä näkyvä teksti, oletuksena ei mitään.
`leftIcon?: ReactElement`	Valinnainen ikoni menupainikkeen vasempaan laitaan. Oikeassa laidassa on automaattisesti alasvetovalikon nuoli, vasemmassa laidassa ei mitään.
`variant?: string`	Painikkeen valinnainen variantti tyylimuotoilua varten. Oletuksena ei varianttia, jolloin käytetään `primary`-muotoilua.

Taulukko 6. `AkamonMenu`-komponenttiin tarvittavat lisäparametrit.

`Modal`. Dialogeja käytetään kirjaston 10 komponentissa. Jokainen esiintymä tulee pyrkiä korvaamaan `AkamonModal`-komponentilla, joka sisällön osalta taipuu moneen eri tarkoitukseen. Komponenttia voidaan muokata, jos huomataan, että lisäominaisuuksille on tarvetta. Chakra UI:n `Modal`:ia käyttävät alla olevassa taulukossa 7 esitetyt komponentit.

Komponentin sijaintipolku	Komponentin nimi
src/components/	AgeRestriction
src/components/	CookieModal
src/components/.../	EtrmNavigationBlocker
src/components/.../	EtrmManualGsrnSelectModal
src/components/.../	EtrmOfferSummaryCardSendToEmailModal
src/components/.../	MessageModal
src/components/.../	NewContactModal
src/components/.../	WeatherStationSelect
src/components/.../	ReportModal
src/components/	SelectCustomer

Taulukko 7. Chakra:n `Modal`-komponenttia käyttävät komponentit.

Esimerkkinä muokataan `AgeRestriction`-komponenttia koodilohkojen 7 ja 8 mukaan. Esimerkki `AkamonModal`-komponentin tarvitsemasta toimintopainikejoukosta on esitetty alinna lohkossa 9. Tyylikomponenttiin tarvitaan lisäparametreja, jotka on esitetty taulukossa 8.

```
...
      <Modal
        isOpen={isOpen}  onClose={onClose}
        closeOnOverlayClick={false}>
        <ModalOverlay />
        <ModalContent>
          <ModalHeader>
            {t('header', { ns: 'ageRestriction' })}
          </ModalHeader>
          <ModalBody>
            {t('info', { ns: 'ageRestriction' })}
          </ModalBody>
          <ModalFooter>
            <Button  {...buttonProps}
              w="full"
              onClick={() => onClickClose()}>
              {t('confirm', { ns: 'button' })}
            </Button>
          </ModalFooter>
        </ModalContent>
      </Modal>
...
```

Koodilohko 7. Korvattava `Modal`-esiintymä.

```
...
      <AkamonModal
        isOpen={isOpen}
        onClose={onClose}
        closeOnOverlayClick={false}
        title={t('header', { ns: 'ageRestriction' })}
        content={t('info', { ns: 'ageRestriction' })}
        actionButtons={actionButtons}
      />
...
```

Koodilohko 8. `Modal`-esiintymän korvaava koodi.

```
...
      const actionButtons = [
        {
          variant: 'primary',
          text: t('confirm', { ns: 'button' }),
          onActionButtonClick: () => {
            onClickClose();
          },
          shouldClose: true,
          width: 'full',
        },
      ];
...
```

Koodilohko 9. Komponentin parametrina saama toimintopainikejoukko.

Parametri	**Selitys**
`triggerButton: ReactNode`	Valinnainen dialogin avaava elementti. Parametrina annettaessa voi olla joko painike, ikoni tai jokin muu elementti. `Modal`:in avautuminen voi perustua myös ehtolauseeseen, jolloin erillistä triggeriä ei tarvita.
`onClose: () => void`	Dialogin sulkeva toiminto annetaan kutsun yhteydessä, koska voi olla tarpeen suorittaa erilaisia toimintoja dialogin sulkeutuessa.

Taulukko 8. `AkamonModal`-komponenttiin tarvittavat lisäparametrit.

Toimintopainikkeiden leveys voidaan antaa valinnaisena tietona painikkeiden objektijoukossa, ominaisuus on lisätty `ActionButton`-tyyppiin. Samoin Dialogin avaava painike on poistettu `AkamonModal`-komponentin sisältä. On myös lisätty mahdollisuus antaa `closeOnOverlayClick`-ominaisuus komponenttikutsun yhteydessä. Chakra:n `Modal`-komponentin oletus on `true`. Samoin dialogin koko voidaan nyt välittää komponenttikutsussa. Chakra:n oletus on `md`. Voisi olla hyvä pohtia, mille muille dialogin ominaisuuksille saattaisi olla tarvetta. `AkamonModal`-komponenttia voidaan muokata käyttötarkoitusten mukaan, kuitenkin niin, että toteutus pysyy monikäyttöisenä.

`Popover`. Ponnahdusikkunoita käytetään neljässä kirjaston komponentissa. Yksi näistä on monikäyttöiseksi komponentiksi tarkoitettu `Popover`, jota edelleen kutsuu 7 muuta kirjaston komponenttia. Chakra UI:n `Popover`:ia käyttävät komponentit on esitetty taulukossa 9.

Komponentin sijaintipolku	Komponentin nimi
src/components/.../	PopoverButton
src/components/.../	InvoiceStatusPopover
src/components/	MarketingApproval
src/components/	Popover

Taulukko 9. Chakra:n `Popover`-komponenttia käyttävät komponentit.

Esimerkkinä muokataan kirjaston `Popover`-komponentti käyttämään `AkamonPopover`-komponenttia alla olevien koodilohkojen 10 ja 11 mukaan, jolloin siihen sidoksissa olevien komponenttien tulisi korjautua kerralla. Tyylikomponenttiin tarvittavat lisäparametrit on esitetty jäljempänä taulukossa 10.

```
...
      <Popover trigger="hover" {...popoverProps}>
        <PopoverTrigger>
          {children ||
          <InfoOutlineIcon
            data-testid="default-popover-icon" />}
        </PopoverTrigger>
        <PopoverContent>
          <PopoverHeader>
            {header}
          </PopoverHeader>
          <PopoverCloseButton />
          <PopoverBody textAlign="left">
            <Stack>
              {tooltips?.map((tooltip) => (
                <Text key={tooltip}>{tooltip}</Text>
              ))}
            </Stack>
```

```
          </PopoverBody>
        </PopoverContent>
      </Popover>
...
```

Koodilohko 10. Korvattava `Popover`-esiintymä.

```
...
      <AkamonPopover
        trigger="hover"
        triggerButton={
          <InfoOutlineIcon
             margin={0}
             boxSize="30px" />
        }
        title={header}
        content={
          <Stack>
            {tooltips?.map((tooltip) => (
              <Text key={tooltip}>{tooltip}</Text>
            ))}
          </Stack>
        }
        {...popoverProps}
      />
...
```

Koodilohko 11. `Popover`-esiintymän korvaava koodi.

Parametri	**Selitys**
`triggerButton: ReactNode`	Ponnahdusikkunan avaava painike. Parametrina annettaessa voi olla joko painike, ikoni tai jokin muu elementti.
`trigger: string`	Toiminto, joka avaa ponnahdusikkunan (`click/hover`). Tämä parametri sisältyy Chakra:n `PopoverProps`:iin, joten sitä ei tarvitse erikseen esitellä. Silti, jos halutaan ylikirjoittaa oletustriggeri, se on annettava komponentille kutsun yhteydessä.

Taulukko 10. `AkamonPopover`-komponenttiin tarvittavat lisäparametrit.

Kirjaston `Popover`-komponenttia kutsuvat seuraavassa taulukossa 11 esitellyt komponentit. On syytä selvittää, vaatiiko jokin niistä edelleen muutoksia tyylikomponenttiin.

Komponentin sijaintipolku	Komponentin nimi
src/components/.../	CarConsumption
src/components/.../	ChargePhases
src/components/.../	KmEstimation
src/components/.../	PowerControl
src/components/.../	ConsumptionTimeRange
src/components/.../	YearlySavings
src/components/.../	StatsCard

Taulukko 11. Kirjaston `Popover`-komponenttia käyttävät komponentit.

`Table`. Chakra:n taulukkokomponentteja on käytössä kirjaston neljässä komponentissa, joista `Datatable`-komponenttia käyttää edelleen yksi ja `BrandTable`-komponenttia kaksi muuta kirjaston komponenttia. Organisaatiospesifit `AkamonTable`, `AkamonBrandTable` ja `AkamonDatatable` ovat erilaisiin käyttötarkoituksiin tehtyjä malleja. Tarkastellaan jokaista kirjaston esiintymää erikseen ja valitaan tarkoitukseen parhaiten sopiva organisaatiospesifi komponentti. Chakra UI:n `Table`:a käyttävät komponentit on esitetty alla olevassa taulukossa 12.

Komponentin sijaintipolku	Komponentin nimi
src/components/	Datatable
src/components/.../	MessagesTable
src/components/.../	ReportTable
src/components/.../	BrandTable

Taulukko 12. Chakra:n `Table`-komponenttia käyttävät komponentit.

`BrandTable`-komponenttia kutsuvat edelleen komponentit `ContractsItem` ja `Profile`, jotka muokataan käyttämään `AkamonBrandTable`-komponenttia. Muutos on suhteellisen yksinkertainen. Alla olevissa koodilohkoissa 12 ja 13 on esitetty esimerkkinä olleen `Profile`-komponentin muutos.

```
...
      <BrandTable
         title=""
         caption={t('caption')}
         tableItems={tableRows}
       />
...
```

Koodilohko 12. Korvattava `BrandTable`-esiintymä.

```
...
      <AkamonBrandTable
         title=""
         caption={t('caption') || ' '}
         tableItems={tableRows}
       />
...
```

Koodilohko 13. `BrandTable`-esiintymän korvaava koodi.

`Datatable`-komponenttia kutsuu edelleen `InvoiceHistoryTable`-komponentti, jonka muokkaaminen jätetään tämän rekonstruktion ulkopuolelle. Koska `AkamonDatatable` ottaa parametrina ainoastaan datan ja saraketieto muodostetaan komponentin sisällä, se luonnollisesti esittää kaikki datassa olevat sarakkeet. Jos esitettäviä sarakkeita halutaan rajoittaa, `columns`-parametri on annettava `AkamonDatatable`-komponentille kutsun yhteydessä. Näin ollen tyylikomponenttia on muokattava sen mukaan.

Muut Chakra:n `Table`-komponenttia käyttävät taulukot muokataan käyttämään `AkamonTable`-komponenttia. Esimerkkinä käytetään `MessagesTable`-komponenttia ja käykin ilmi, että `AkamonTable`-komponenttiin tarvitaan lukuisia muutoksia, jotta `MessagesTable`-komponentti voi tätä kutsua. Taulukossa 13 on esitetty komponentin tarvitsemat lisäparametrit.

Parametri	Selitys
`columns: [key: string]: any []`	Taulukossa näytettävät sarakkeet on mainittava, jos datasta ei haluta sisällyttää taulukkoon kaikkea.
`onActionButtonClick?: (item: any) => ReactElement`	`MessageTable`-komponentissa käytetään toimintopainiketta. Painikkeen toimintamekanismi on esiteltävä parametrina kutsuvalle komponentille.
`columnNames: string[]`	Taulukko sarakkeiden nimistä, jotta voidaan muodostaa korttimuotoinen esitys ja erotella otsikko, toimintopainike ja muu sisältö toisistaan.

Taulukko 13. `AkamonTable`-komponenttiin tarvittavat lisäparametrit.

Muutosten yhteydessä on indeksi lisätty `key`-parametriin, koska aiemmin käytetty avain, ensimmäisen solun arvo, voi olla sama. `columns`-parametri muodostettiin aiemmin komponentin sisällä ja se sisälsi sarakkeiden nimet. Nyt se tulee parametrina kutsun yhteydessä ja sen muoto on erilainen. Lisäksi `columnNames`-taulukko muodostetaan komponentin sisällä ja siirretään korttimuodon esitystä varten `renderCards`-funktiolle, jossa sitä tarvitaan erottelemaan otsikossa ja sisältönä näytettävä data.

Taulukonmuodostus on pitänyt miettiä uudelleen, koska `columns`-parametri sisältää nyt sarakkeiden nimien sijaan objektitaulukon. Riveille renderöidään nyt myös toimintoelementti, jos sellainen on parametrina annettu. `renderCards`-funktioon on tehty vastaavat muutokset. Toimintoelementti näytetään otsikossa ja se erotellaan varsinaisen otsikon lisäksi muusta sisällöstä. Korttimuotoisesta esityksestä on otettu pois avain, jolloin sivulla näytetään pelkkä arvo. `AkamonTable`-komponentti on ahkerasti käytössä tyylioppaan sivuilla, koska Chakra:n komponenttien ominaisuudet esitetään tätä taulukkoa käyttämällä. Näitä kaikkia esiintymiä on jouduttu muokkaamaan tämän rekonstruktion myötä.

`MessagesTable`-esimerkistä puuttuu edelleen lukemattomien viestien lihavoitu kirjoitusasu. Tämän ominaisuuden rakentaminen monikäyttöiseen malliin täytyy harkita erikseen. Riittävän universaalin mallin rakentaminen on haastavaa, koska taulukoiden käyttötarkoitus voi olla keskenään hyvin erilainen. On syytä tarkastaa jokaisen esiintymän toimivuus kerrallaan ja harkita tarvitaanko vielä lisää monikäyttöisiä mallitaulukoita.

`Tooltip`. Ohjeen tai lisätiedon antamista varten tarkoitettu `Tooltip`-komponentti on käytössä kirjaston 7 komponentissa. Vaikka kaikkien esiintymien on tarkoitus antaa käyt-

täjälle lisätietoa, ovat avausmekanismit keskenään erilaiset. Käytössä ovat mm. infoikoni, kysymysmerkki, painike, `Box`- tai `Flex`-elementti. Infotekstilaatikon muotoilu olisi hyvä olla yhtenäinen huolimatta siitä mikä elementti sen avaa. Chakra UI:n `Tooltip`:ia käyttävät komponentit on esitelty taulukossa 14.

Komponentin sijaintipolku	Komponentin nimi
src/components/.../	AutomaticSignUpCheckbox
src/components/.../	HousingRadioButtons
src/components/.../	ChargerPhases
src/components/.../	EtrmTopVatSelector
src/components/.../	EtrmOfferRemainingTime
playground/src/components/	TenantConfigurationSelect
playground/src/components/.../	LanguageSelector

Taulukko 14. Kirjaston `Tooltip`-komponenttia käyttävät komponentit.

Esimerkkinä alla olevissa koodilohkoissa 14 ja 15 esitetään `LanguageSelector`-komponentin muuntaminen, joka käy yksinkertaisesti korvaamalla Chakra UI:n komponentti `AkamonTooltip`-komponentilla. Tyylikomponenttiin tarvittavat lisäparametrit on esitetty jäljempänä taulukossa 15.

```
...
      <Tooltip
            label={`Translations are still in progress.
         See Select customer-component and
            Home page for partial implementation.`}
       >
         <InfoIcon fontSize="2xl" mr={0} color={color} />
      </Tooltip>
...
```

Koodilohko 14. Korvattava `Tooltip`-esiintymä.

```
...
      <AkamonTooltip
         label={`Translations are still in progress.
```

```
          See Select customer-component and
          Home page for partial implementation.`}
        >
          <InfoIcon fontSize="2xl" mr={0} color={color} />
        </AkamonTooltip>
...
```

Koodilohko 15. `Tooltip`-esiintymän korvaava koodi.

Parametri	Selitys
`children: ReactNode`	Jotta infotekstin näyttävä elementti voi olla mikä tahansa, on se annettava komponenttikutsun yhteydessä.

Taulukko 15. `AkamonTooltip`-komponenttiin tarvittavat lisäparametrit.

Käyttöliittymän intuitiivisuutta ja opittavuutta sekä yleisilmettä ajatellen, olisi hyvä käyttää yhteistä käytäntöä ohjeen näyttämiselle. Infotekstilaatikko avautuu nyt aina samalla tavalla vasemmalle alas. Voiko olla tarpeen muokata sen sijaintia käyttötarkoituksen mukaan.

4.4.2 Tyylit

`Typografia`. Tekstielementtien muotoilut määritellään teematiedostossa (*akamon-chakra-base-theme.ts*) ja ne voidaan ylikirjoittaa tai täydentää asiakaskohtaisessa teematiedostossa (*customTheme.ts*). Kirjaston komponenteista `Text` esiintyy 93:ssa ja `Heading` 67 komponentissa, joista monesti löytyy myös useampi esiintymä. Valtaosaa näistä on muotoiltu esiintymän yhteydessä erilaisin muotoiluin, joista osa on tyylioppaan puitteissa hyväksyttäviä, kuten koko ja paksuus, mutta myös esimerkiksi väriä on muokattu 13 otsikko- ja 30 tekstiesiintymässä. Täysin vailla muotoilua ovat 18 tekstiä ja 9 otsikkoa. `Text`- ja `Heading`-komponenttien esiintymät muotoiluineen on koottu erilliseen tiedostoon jatkojalostusta varten (*Text-Heading-instanssit-AkamonSierra.xlsx*).

Yhtenäiseen ulkoasua edesauttavat merkittävästi keskenään samanlaiset otsakkeet ja tekstit. Teematiedostoon merkittyjen yhteisten käytänteiden ja tyylien tulisi ohjata eri kehittäjiä samaan lopputulokseen. Tyylioppaaseen on valittu tietyt koot ja paksuudet typografiaelementeille, joista kehittäjä valitsee tarkoitukseen sopivan, sen sijaan, että muokkaa esiintymää mielivaltaisesti. Usein muotoilut ovat kuitenkin tarpeellisia, kun halutaan korostaa toimintoa tai infoa värillä tai vaikkapa sovittaa teksti painikkeeseen. Tämän vuoksi esiintymiä on syytä tarkastella

yksitellen ja korjata ne, jotka vaikuttavat yhtenäiseen ilmeeseen häiritsevästi. Pohdittavaksi jää, voisiko varianttimenettelystä olla hyötyä myös typografiaelementtien yhtenäistämisessä?

`Värit`. Väripaletti määritellään teematiedostossa (*akamon-chakra-base-theme.ts*) ja se voidaan ylikirjoittaa tai täydentää asiakaskohtaisessa teematiedostossa (*customTheme.ts*). Tällä hetkellä kirjaston komponenteissa väriä käytetään vaihtelevasti niin, että osa esiintymistä käyttää teematiedostossa määriteltyjä värikoodeja (esim. `brand.300`), osa Chakra UI:n värikoodeja (esim. `white`, `red.500`), osassa väri on annettu hexa- tai oklch-koodina. Jotta ilme ja ennen kaikkea ylläpidettävyys säilyisi, tulisi jokaisen väriesiintymän nojata teematiedostoon. Samalla asiakaskonfiguraatiot helpottuisivat värien osalta huomattavasti.

Tällä hetkellä `color`-ominaisuus sisältyy jossain muodossaan kirjaston 191 komponenttiin ja mainintoja on yhteensä reilu tuhat. Värien esiintymät komponenttikirjastossa on koottu erilliseen tiedostoon (*Color-instanssit-AkamonSierra.xlsx*). Värien käytön yhtenäistäminen vaatii työtä, mutta suoraviivaistaa asiakastoteutuksia jatkossa, jos teematiedostoon tehdyt muutokset uivat kerralla kaikkialle sovelluksessa. Tyylioppaaseen on valittu eräänlainen tapa nimetä värejä (`primary`, `secondary` jne.). Väripaletit voi toki nimetä muutenkin ja niitä voidaan lisätä tarpeen mukaan, tärkeintä on kuitenkin, että sovelluksen kaikki väriesiintymät nojaisivat näihin paletteihin.

`Ikonit`. Ikoneita käyttää kirjaston 61 komponenttia, joissa on yhteensä 87 esiintymää. Ikoneita on maltillisesti muotoiltu, eivätkä ne itsessään aiheuta suurta epäyhtenäisyyttä sovelluksen ulkoasuun. Värin käyttämistä ikoneissa kannattaa harkita tilanteen mukaan. Samoin ikonin koko (`boxSize`) on esiintymä kerrallaan sovitettava tarkoitustaan vastaavaksi. Ikonien esiintymät komponenttikirjastossa on koottu erilliseen tiedostoon (*Icon-instanssit-AkamonSierra.xlsx*).

4.4.3 Mallit

`Välistykset ja sivumallit`. Sivumallin käyttöönotto täytyy miettiä tarkkaan, koska se vaatinee eniten töitä. Sivuille sijoitettavat komponentit sisältävät erilaisia raameja, marginaaleja ja täytteitä, jotka sotkevat sivumallin käyttöä. Malli on rakennettu niin, että komponentit voidaan sijoittaa siihen sellaisenaan ilman erillisiä raameja tai välistyksiä. Idea on, että malli hoitaa välistykset, jotta ilme säilyy yhtenäisenä.

Käytännössä asia ei ole näin suoraviivainen, ainakaan olemassa olevien komponenttien osalta.

Monet komponenteista tarvitsevat ympärilleen erilaisia raameja, jotta komponentin sisäisten elementtien asettelu toimii halutulla tavalla. Vaikka komponentin ulommainen raami asetettaisiin sivumalliin ilman erillistä välistystä, taustan ollessa väritön, itse sisältö ei täytä koko tilaa. Komponentit näyttävät olevan aseteltu erilaisin välistyksin ja yhtenäinen ilme hajoaa. Alla olevissa kuvioissa 12 ja 13 on esimerkki samasta komponentista, joista ylemmän raami sisältää vertikaalista marginaalia ja alemman ei. Otsikon ja sivusisällön välistys on tämän vuoksi erilainen ja johtaisi kahden sivun epäyhtenäiseen ulkoasuun.

Kuvio 12. Yhden sisällön sivumalli komponentin sisäisen marginaalin kanssa.

Kuvio 13. Yhden sisällön sivumalli ilman komponentin sisäistä marginaalia.

Myös rinnakkain aseteltuna otsikon ja sisällön väliseen eroon vaikuttaa komponenttiin rakennettu marginaali, mutta rinnakkaiset sisällöt asettuvat samalla lailla keskenään, koska

suuremman elementin korkeus määrää sisältörivin korkeuden ja sisältö on keskitetty. Kuvioissa 14, 15 ja 16 vasen komponentti ei sisällä vertikaalista marginaalia ja oikea sisältää.

Kuvio 14. Kahden sisällön sivumalli toisen komponentin sisäisen marginaalin kanssa.

Kuvio 15. Kahden sisällön sivumalli marginaaliton komponentti korostettuna.

Kuvio 16. Kahden sisällön sivumalli marginaalia sisältävä komponentti korostettuna.

Sivumallin käyttöönotto ja sen toiminnan hyödyntäminen vaatii sivujen läpikäyntiä yksi kerrallaan. Komponenttien uloimmissa kääreissä olevat marginaalit tulee mahdollisuuksien mukaan poistaa ja sisempien kääreiden välistysominaisuuksien merkitys tarkastella erikseen. Tällä hetkellä sisältörivin elementit on keskitetty vertikaalisesti, jolloin rivi näyttää hyvältä, kun sisällöt ovat keskenään samankaltaiset. Kahden hyvin erilaisen ja erikokoisen elementin sijoittaminen samalle riville voi vaatia keskityksen poistamista. Hyödyllistä olisi myös tutkia, missä kaikkialla välistys-ominaisuutta (engl. *spacing*) käytetään ja korvata esiintymät tyylioppaassa esitellyillä välistysarvoilla.

4.5 Suunnittelusykli: tyylinvalvojan toteutus

Markkinoilla on runsaasti valmiita visuaaliseen regressiotestaukseen tarkoitettuja ohjelmistoja. Tyylioppaan suunnittelun ja toteutuksen viedessä suunniteltua kauemmin todetaan, ettei tämän tutkimuksen puitteissa ole mahdollista toteuttaa tekoälyyn, tietokonenäköön ja koneoppimiseen perustuvaa visuaalista testausta. Sen sijaan yritetään löytää paras snapshot-tekniikalla toimiva työkalu, joka suorittaa tarvittavat visuaaliset regressiotestit ja taipuu jatkossa mahdollisesti myös toiminnalliseen testaukseen.

4.5.1 Testaustyökalujen esittely

Esitellään kolme snapshot-tekniikkaan ja pikselivertailuun perustuvaa työkalua vaihtoehtoina tyylinvalvojaksi. Sen lisäksi kerrotaan lyhyesti myös eräästä teköälyä käyttävästä työkalusta, koska on hyödyllistä hieman pohtia myös tekoälyyn perustuvan mallin toimintalogiikkaa, jotta ymmärrys saatavilla olevista vaihtoehdoista lisääntyy. Kohdeorganisaatio voi halutessaan harkita sellaiseen malliin siirtymistä tulevaisuudessa.

Vitest image snapshots. Komponenttikirjaston toiminnallista testausta hoidetaan tällä hetkellä Vitest-ohjelmistolla, mikä on erityisesti Vite-projektien käyttöön suunniteltu testausympäristö. Tämän vuoksi Vitest:in taipuvuus visuaaliseen testaukseen halutaan tutkia ensimmäisenä. Vitest:issä on olemassa snapshot-ominaisuus, jossa ohjelmisto ottaa tilannekuvan annetusta arvosta ja vertaa sitä sitten testin rinnalle tallennettuun referenssitiedostoon. Vaikka tekniikkaa käytetään myös koodirivien ja tiedostojen vertailuun, tämän tutkimuksen puitteissa keskitytään nimenomaan kuvien vertailuun keskenään. Kuvien snapshot-testaukseen tarvitaan avuksi Jest-laajennus, jest-image-snapshots, joka luo Puppeteer-kirjastoa käyttäen kuvan testattavasta käyttöliittymästä tai yksittäisestä komponentista ja vertaa sitä pikselitasolla aiemmalla testiajolla muodostettuun referenssikuvaan. Testi epäonnistuu, jos kaksi tilannekuvaa eivät täsmää: joko muutos on odottamaton tai referenssikuva on päivitettävä uuteen versioon. (You 2024b)

Käytännössä toiminta on seuraavanlainen:

- **Tallennetaan referenssikuva.** Ensimmäisellä testiajolla Puppeteer navigoi käyttöliittymässä halutulle sivulle ja ottaa kuvan testin kohteena olevasta näkymästä tai yksittäisestä komponentista. Kuva tallennetaan referenssikuvaksi seuraaville testiajoille.
- **Otetaan kuvakaappaus.** Seuraavilla testiajoilla Puppeteer tallentaa uuden kuvan testikuvana, jota jest-image-snapshot sitten vertaa aiemmin tallennettuun referenssikuvaan.
- **Suoritetaan pikselivertailu.** Uutta testikuvaa verrataan referenssikuvaan pikseli pikseliltä. Työkalu laskee pikselierot ja vertaa tulosta annettuun toleranssiin. Toleranssi voidaan määritellä testi kerrallaan ja näin asettaa tiukemmat tai löysemmät rajat testauksen kohteen mukaan.
- **Esitetään tulokset.** Jos toleranssirajat ylittyvät, työkalu tallentaa poikkeamakuvan, jossa muutokset näkyvät korostettuna. Kuva on kolmen kuvan sarja, jossa esitetään vasemmalla odotettu kuva (referenssikuva), oikealla todellinen kuva ja keskellä virheet.

Tämän jälkeen kehittäjän täytyy tarkastella tuloksia ja arvioida, onko kyse tahattomasta poik-

keamasta vai halutaanko referenssikuvaa muuttaa. Jos muutos on tarkoituksellinen, referenssikuva päivitetään vastaamaan uutta tilannetta komennolla `-u`. Koska Vitest on jo käytössä kirjaston toiminnallisessa testauksessa, on luonnollista, että se valitaan yhdeksi vaihtoehdoksi myös visuaaliseen testaukseen.

Playwright visual comparisons. Toinen Vite-projektien kanssa hyvin yhteensopiva testausympäristö on Microsoftin kehittämä Playwright Test, joka on tehokas useita selaimia tukeva toiminnalliseen sekä end-to-end -testaukseen soveltuva työkalu. Kohdeorganisaatio on tutkimuksen edetessä esittänyt kiinnostusta tätä ohjelmistoa kohtaan, joten sen taipumista visuaaliseen testaukseen halutaan myös tutkia. Playwright Test tarjoaa samanlaiset toiminnallisuudet snapshot-testaukseen kuin Vitest, joskin erilaisin kirjastoin. Playwright ottaa itse kuvakaappaukset ja käyttää kuvien vertailussa Pixelmatch-kirjastoa.

Käytännössä toiminta on seuraavanlainen:

- **Tallennetaan referenssikuvat.** Ensimmäisellä ajolla Playwright luo referenssikuvat ottaen useita kuvia ja kahden peräkkäisen kuvan vastatessa toisiaan, jälkimmäinen tallennetaan referenssikuvaksi.
- **Otetaan testikuvat.** Seuraavilla testiajoilla otetaan uusia kuvakaappauksia verrattavaksi referenssikuviin. Koska kuvakaappaukset vaihtelevat erilaisissa selaimissa ja alustoissa, Playwright-dokumentaatiossa suositellaan ottamaan erilliset kuvat erilaisille ympäristöille.
- **Suoritetaan vertailu.** Pikselivertailun suorittaa Pixelmatch-kirjasto, jolle voidaan määrittää toleranssirajat. Esimerkiksi erilaisten ympäristöjen testaamisessa on hyödyllistä sallia pieniä eroja ilman, että testi epäonnistuu.
- **Raportoidaan tulokset.** Testin epäonnistuessa virheet raportoidaan kolmena kuvana, joista ensimmäinen on referenssikuva, toinen testiajon yhteydessä otettu kuva ja kolmannessa kuvassa esitetään visuaaliset poikkeamat korostettuna. Kuvat on tallennettu erillisinä .png-tiedostoina *test-results*-kansioon.

Näiden vaiheiden jälkeen testaajan on arvioitava testaukset tulokset ja päätettävä korjataanko virhe vai halutaanko referenssikuvaa muuttaa. Uuden referenssikuvan tallentaminen onnistuu vaivatta testiajon yhteydessä annettavalla komennolla `-update-snapshots`. (Microsoft 2024a)

Cypress image snapshot. React-sovellusten kanssa hyvin toimiva ja Vite-sovelluksia tukeva

Cypress tarjoaa laajan tuen end-to-end- ja toiminalliselle testaukselle. Lisäosien myötä myös visuaalinen regressiotestaus on mahdollista. Cypress käyttää snapshot-kuvien ottamiseen lisäosaa cypress-image-snapshot, jolla otetaan kuva koko testattavasta sovelluksesta tai tietystä elementistä. Kuvaa verrataan aiemmin hyväksyttyyn peruskuvaan, kuten edellä esitetyissä työkaluissa. Vaihtoehtona on olemassa myös kaupallinen työkalu, Cypress Visual Testing (Cypress Cloud), joka suorittaa snapshot-vertailun pilvessä ja sisällyttää visuaalisen regressiotestauksen osaksi CI/CD-putkea. (Cypress.io 2024)

Käytännön toiminta testauksessa on aivan samanlainen, kuin edellä esiteltyjen työkalujen osalta. Ensimmäisellä ajolla muodostetaan referenssikuva, jota vastaan varsinaisten testiajojen kuvakaappauksia arvioidaan. Virheet testiajossa tuottavat poikkeamakuvan, jossa vasemmalla esitellään odotettu tulos ja oikealla varsinainen testin tulos, erot korostettuina keskimmäisessä kuvassa. Edellisistä työkaluista poiketen Cypressiin kuuluu käyttöliittymätyökalu, jossa testit voidaan ajaa ja niiden tuloksia tarkastella.

Cypress:iin, kuten muihinkin suosittuihin testausalustoihin on saatavilla myös visuaaliseen testaukseen suunniteltuja Plugineita. Näistä mainittakoon esimerkkinä Applitools Eyes, jonka toimintaa tarkastellaan seuraavassa.

Applitools Eyes. Applitools on suosittu tekoälyavusteinen testausratkaisu, joka integroituu helposti useimpiin nykyaikaisiin testausalustoihin, kuten yllä mainitut Vitest, Playwright ja Cypress. Sen keskeinen komponentti on Applitools Eyes, joka käyttää Visual AI-tekoälyä ja kuvantunnistustekniikoita visuaalisten erojen havaitsemiseksi käyttöliittymän kuvakaappauksissa, jotta niitä voidaan tarkastella ihmissilmän tavoin. Kuvakaappauksia voidaan arvioida useilla eri alustoilla ja laitteilla yhden testiajosarjan aikana. Tekoäly osaa erottaa merkitykselliset visuaaliset erot pienistä renderöintieroista, joihin pikselivertailussa usein takerrutaan. Applitools Eyes:ille voidaan määrittää alueita, esim. päivämääriä ja ilmoituksia, jotka se voi ohittaa. Vaihtoehtoisesti dynaaminen sisältö voidaan ohittaa tekoälyn avulla. Pääasiassa testaus tapahtuu hyvin samalla tavalla kuin perinteisessä snapshot-pikselivertailumallissa, mutta tulokset ovat luotettavampia ja manuaalisen työn määrä vähäisempi.

Testauksen vaiheet Applitools Eyes:illa:

- **Integrointi.** Työkalu integroidaan käytössä olevaan testausympäristöön ja testit määritellään ottamaan kuvakaappauksia joko kokonaisista sivuista tai yksittäisistä komponenteista.

- **Otetaan kuvakaappaukset.** Testiajoilla Applitools Eyes tallentaa kuvakaappaukset ja vertaa niitä referenssikuviin. Eroavaisuudet raportoidaan pilvipohjaisessa käyttöliittymässä poikkeamakuvina.
- **Raportoidaan tulokset.** Applitools:in raportointityökalu, Test Manager tarjoaa käyttäjästävällisen tavan analysoida testien tuloksia. Eyes on jo valmiiksi poistanut vääriä positiivisia hälytyksiä, joten tarkasteltavaa jää kehittäjälle vähemmän.
- **Hyväksytään tai hylätään muutokset.** Uusien referenssikuvien tallentaminen tapahtuu helposti Task Managerin kautta.

Tällaisen tekoälyavusteisen työkalun merkitys korostuu siinä, että kehittäjille jää vähemmän manuaalista työtä. Useiden erilaisten laite- selain- ja alustayhdistelmien sekä erilaisten resoluutioiden testaaminen käy tekoälyltä vaivatta. Visual AI osaa suodattaa merkityksettömät erot oikeista virheistä, tehdä alustariippumattomia vertailuja, huomioida käyttöliittymän mukautumisen erikokoisille näytöille ja ohittaa dynaamisen sisällön aiheuttamat virheet. Tällä tavalla testauksen tarkkuus paranee, testisyklien läpimeno nopeutuu ja voidaan testata yhä suurempia määriä käyttöliittymäkomponentteja samalla ajolla. (Applitools 2024)

Markkinoilla on runsaasti myös muita tekoälyyn ja koneoppimiseen perustuvia testaustyökaluja, kuten LambdaTest, Functionize, Testim.io ja Percy sekä lukuisia muita. Ne käyttävät tekoälyä testausprosessin eri vaiheissa testien kirjoittamisesta prosessin automatisointiin. Tätä tutkimusta varten tehty 18 kaupalliseen verkkosivustoon perustuva katsaus kuitenkin nosti Applitools:in useimmin kolmen kärkeen, kun analysoitiin 70 erilaista visuaalisen testauksen työkalua niistä tehtyjen paremmuusvertailujen avulla. Työkaluja ilmestyy markkinoille koko ajan ja käytetyn tekoälyn osuus kasvaa myös visuaalisen testauksen saralla.

4.5.2 Työkalujen vertailua ja tyylinvalvojana käytettävän testausympäristön valinta

Työkalujen vertailua varten luodaan uusi React Vite-sovellus, jossa on yksinkertainen profiilitietoja tallentava käyttöliittymäsivu sisältäen lomakkeen syötekenttineen ja tallennuspainikkeineen. Kaikki kolme testaustyökalua asennetaan ja konfiguroidaan tarkoitusta varten tähän sovellukseen, jotta vertailu voidaan suorittaa samanlaisella sisällöllä. Konfigurointi on haastavinta Cypress:issä, Playwright:issa asennus ja käyttöönotto sujuu helpoiten. Luodaan kullakin työkalulla kaksi testiä, toinen, joka tarkistaa kokonaisen verkkosivun ja toinen, joka tutkii yksittäisen painike-komponentin. Testien syntaksi on hyvin samanlainen kaikissa työkaluissa. Koska Vitest käyttää Puppeteer-kirjastoa apuna, on siinä sen vuoksi muutama

koodirivi enemmän. Lisäksi snapshotille on Vitest:issa annettava tarkempia parametreja, kuin muiden työkalujen tapauksessa. Koko sivun testissä navigoidaan ensin halutulle verkkosivulle ja otetaan sivusta kuvakaappaus. Alla olevissa koodilohkoissa 16, 17 ja 18 esitetään kullakin työkalulla kirjoitetut testitapaukset.

```
it('profile', async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('http://localhost:5173/profile');
  const imageBuffer = await page.screenshot({
    type: 'png',
    encoding: 'base64',
    fullPage: true });
  expect(Buffer.from(imageBuffer, 'base64'))
   .toMatchImageSnapshot();
});
```

Koodilohko 16. Vitest-syntaksi, koko sivun testi.

```
test("profile visual test", async ({ page }) => {
  await page.goto("http://localhost:5173/profile");
  expect(await page.screenshot())
    .toMatchSnapshot("profile.png");
});
```

Koodilohko 17. Plawright-syntaksi, koko sivun testi.

```
it("should match the profile snapshot", () => {
  cy.visit("/profile");
  cy.matchImageSnapshot("profile");
});
```

Koodilohko 18. Cypress-syntaksi, koko sivun testi.

Testatessa yksittäistä komponenttia, hakeudutaan ensin halutulle verkkosivulle, etsitään sieltä testattava html-elementti, josta kuvakaappaus otetaan. Elementtiä voidaan html-tagin lisäksi etsiä myös luokan tai id:n perusteella. Alla olevissa koodilohkoissa 19, 20 ja 21 esitetään syntaksi eri työkaluilla.

```
it('profilebutton', async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto('http://localhost:5173/profile');
  const buttonComponent = await page.$('button');
  if (buttonComponent) {
    const image = await buttonComponent.screenshot({
      type: 'png',
      encoding: 'base64' });
  expect(Buffer.from(image, 'base64'))
    .toMatchImageSnapshot();
  } else {
    throw new Error("Button not found on the page.");
  }
  await browser.close();
});
```

Koodilohko 19. Vitest-syntaksi, komponentin testi.

```
test("profile button component", async ({ page }) => {
  await page.goto("http://localhost:5173/profile");
  const buttonComponent = await page.locator("button");
  await expect(buttonComponent)
    .toHaveScreenshot("profileButton.png");
});
```

Koodilohko 20. Plawright-syntaksi, komponentin testi.

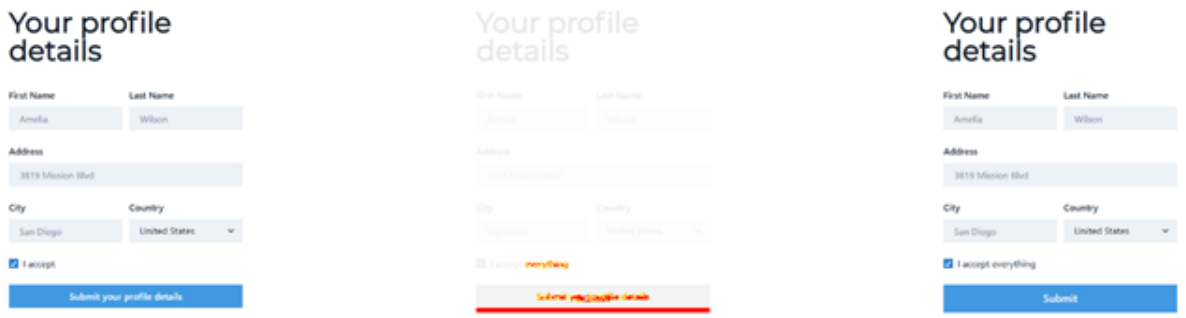
```
it("should match the button snapshot", () => {
  cy.visit("/profile");
  cy.get("button")
    .matchImageSnapshot("profilebutton-snapshot");
});
```

Koodilohko 21. Cypress-syntaksi, komponentin testi.

Cypress DevTools -käyttöliittymä on informatiivinen, testien ajo sujuu näppärästi sitä kautta. Muissa työkaluissa ei sellaista ole, vaan testien ajo tapahtuu komentoriviltä. Ensimmäisellä testiajolla kukin ohjelma tallentaa referenssikuvan profiilisivusta ja sillä olevasta painikkees-

ta. Tehdään käyttöliittymään ulkoasuun vaikuttavia muutoksia: kasvatetaan painikkeen kokoa, muutetaan sen tekstiä ja lisätään lomakkeen valintaruutuun tekstiä. Tämän jälkeen testit ajetaan uudelleen. Odotetusti testit epäonnistuvat. Poikkeamakuvat tallentuvat kaikkien ohjelmistojen tapauksessa omaan erilliseen kansioonsa, josta ne ovat helposti löydettävissä. Vitest ja Cypress esittävät poikkeamat kolmen kuvan sarjana, kun Playwright:issa kuvat tallentuvat erillisinä tiedostoina. Kolmen kuvan sarjasta poikkeamat löytyvät helpommin, kun kaikkia kuvia voi katsella samanaikaisesti.

Testien tarkkuudessa on eroja. Vitest:in ja Playwright:in ilmoittamat virheet olivat todellisten muutosten aiheuttamia, kun Cypress ilmoitti lisäksi virheistä, jotka eivät olleet todellisia. Kaikissa ohjelmistoissa on mahdollista määritellä toleranssi, jota testiajoissa sovelletaan. Vitest:in poikkeamakuvat profiilisivulle ja painikkeelle on esitetty kuvioissa 17 ja 18.

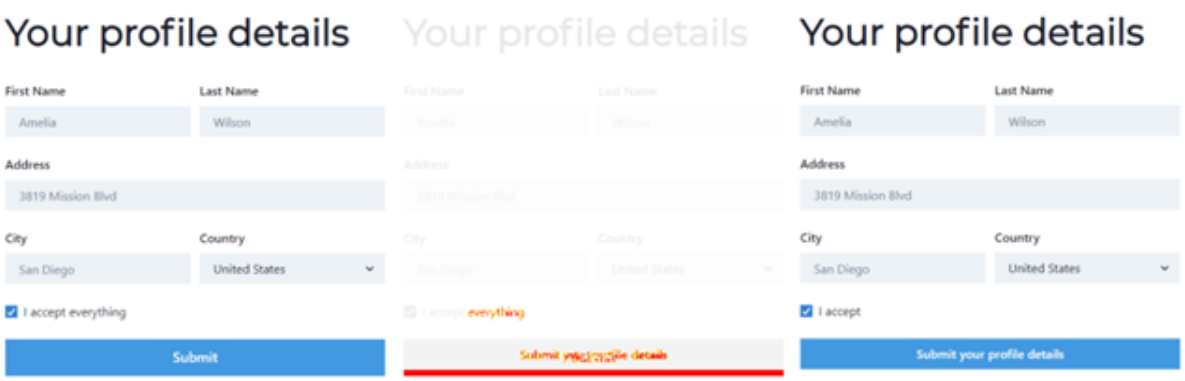


Kuvio 17. Vitest, koko sivun testi.

Kuvio 18. Vitest, komponentin testi.

Vitest:issä vasemmalle tulostuu referenssikuva, oikealle todellinen kuva ja keskellä näkyvät korostettuna virheet näiden välillä. Testi odotetusti korostaa painikkeen muuttuneen korkeuden, painikkeessa olevan tekstin ja valintaruutuun lisätyn tekstin.

Playwright tallensi seuraavanlaiset kolme erillistä kuvaa epäonnistuneiden testien pohjalta (kuviot 19 ja 20). Vasemmalla on todellinen kuva, oikealla referenssikuva ja keskellä näkyvät virheet. Tulos on hyvin samanlainen kuin Vitest:illä.



Kuvio 19. Playwright, koko sivun testi.

Kuvio 20. Playwright, komponentin testi.

Cypress esitti virheet alla olevissa kuvioissa 21 ja 22 esitetyllä tavalla. Vasemmalla on referenssikuva, oikealla todellinen kuva ja keskellä virheet. Koko sivun kuvasta huomataan, että Cypress ilmoitti myös virheestä, joka ei ollut todellinen. Kuvasta, eikä DevTools -käyttöliittymästä selviä käyttäjälle, miksi osoitekenttä on korostettu. Todennäköisesti kyse on väärästä hälytyksestä ja toleranssin kasvattaminen auttaisi tällaisten eliminoimisessa.

Kuvio 21. Cypress, koko sivun testi.

Kuvio 22. Cypress, komponentin testi.

Jos testin ilmoittamat muutokset todetaan tarkoituksenmukaisiksi, täytyy referenssikuvat päivittää vastaamaan muuttunutta tilannetta. Uusien referenssikuvien tallentaminen onnistuu Vi-

test:issä ja Playwright:issa helposti komentoriviltä antamalla komento `update`. Sen sijaan Cypress:issä vanha referenssikuva on joko manuaalisesti poistettava, nimettävä uudelleen tai siirrettävä toiseen kansioon, jotta uusi referenssikuva voidaan ottaa.

Koska ohjelmistojen välille ei vertailussa syntynyt suurta eroa ja ottaen huomioon Vitest:in olevan käytössä jo toiminnallisessa testauksessa, valitaan myös visuaalisten regressiotestien työkaluksi Vitest. Kohdeorganisaatiolle jää myös kaksi muuta toimivaksi todettua vaihtoehtoa, jos testausympäristöä joskus halutaan muuttaa.

4.5.3 Testaustyökalun käyttöönotto komponenttikirjastossa

Ensin on suoritettava testausta varten tarvittavien pakettien asennus. Koska Vitest on jo käytössä, sitä ei tarvitse asentaa uudelleen. Lisätään sen sijaan jest-image-snapshot-laajennus, joka suorittaa pikselivertailun ja kuvan ottamista varten tarvittava Puppeteer-kirjasto. TypeScript vaatii lisäksi vielä tyypityksiä, joten lisätään myös @types/jest-image-snapshot. Vite:n konfigurointitiedosto *vite.config.ts* sisältää jo testausta varten tarvittavat tiedot ja *setup.ts* -tiedosto on jo olemassa. Lisätään sinne visuaalista testausta varten globaali työkalu jest-image-snapshot.

Testausta varten on jo luotu kansio *__tests__* projektin juurikansioon. Avataan sinne uusi kansio *visual*, jotta visuaaliset testit on helpompi erotella toiminnallisista testeistä. Tähän kansioon luodaan testitiedosto, *Visual.test.ts* ja siihen testien lähdekoodi. Vitest:in ollessa jo käytössä, testien käynnistys komennolla `yarn test` aiheuttaa kaikkien testien ajon kerralla. Jotta visuaaliset testit voidaan ajaa erillään toiminnallisista testeistä, lisätään *package.json*-tiedostoon uusi visuaalisten testien käynnistys-scripti `test:visual:vitest -dir ./src/__tests__/visual`, joka etsii testitiedostoja *visual*-kansiosta ja käynnistää ne nyt komennolla `yarn test:visual`.

Tyylioppaan sivut kokoavat yhteen komponenttikirjastoon lisätyt organisaatiospesifit komponentit. Nämä sivut voidaan testata tällä menetelmällä yksi kerrallaan. Kuitenkin tyylioppaan sivujen testausta tärkeämpää on tyylikomponenttien ja niiden esiintymien testaus. Esimerkkinä käytetään sivua, johon on koottu muutamia tyylioppaan komponentteja. Sivun pohjana käytetään tietenkin tyylioppaaseen kuuluvaa `AkamonPageTemplate`-komponenttia ja sisällön muodostuksessa `AkamonPageSection`-komponenttia.

Testien tarkkuus voidaan määrittää seuraavilla `configureToMatchImageSnapshot`-funktion ominaisuuksilla *setup.ts*-tiedostossa :

`customDiffConfig: {threshold: 0.1}` määrittää, kuinka vertailualgoritmi havaitsee ja mittaa erot yksittäisissä pikseleissä. Tälle annettu `threshold`-arvo kertoo pikselikohtaisen toleranssin. Pienempi arvo tekee vertailusta tiukemman ja havaitsee pienetkin muutokset pikseleissä. Esimerkiksi arvo 0.1 tarkoittaa, että kaikki alle 10% erot pikselissä sallitaan.
`failureThreshold: 0.01 & failureThresholdType: pixel / percent` asettaa suurimman sallitun eron referenssikuvan ja tilannekuvan välillä eli kuinka suuri kokonaisero kuvassa hyväksytään sille annetun mittaustyypin perustella joko pikseleinä tai prosentteina. Esimerkiksi arvo 0.01 tyypillä *percent* tarkoittaa, että jos yli 1% kuvan kokonaispikselien määrästä eroaa, testi epäonnistuu. Arvo 0.01 tyypillä *pixel* tarkoittaa, että testi epäonnistuu, jos referenssikuvan ja uuden kuvan välillä on enemmän kuin 0.01 pikselin ero. Koska pikselit mitataan kokonaislukuina, tällainen toleranssi ei käytännössä salli yhdenkään pikselin eroa ja kaikki havaitut poikkeamat johtavat testin epäonnistumiseen. Tarkoituksenmukaisempaa olisi määrittää pikselitoleranssi kokonaislukuna. Esimerkiksi hyvin pienille kuville kuten ikoneille voi olla järkevää käyttää kiinteää pikselitoleranssia. Sen sijaan suuremmille komponenteille ja kokonaisille sivuille on järkevämpää käyttää pikselien määrän sijaan niiden prosenttiosuutta. *Setup.ts*-tiedostossa olevat yleiset toleranssit voidaan ylikirjoittaa testikohtaisesti antamalla testin lähdekoodin yhteydessä `toMatchImageSnapshot`-funktiolle kyseiset arvot.

4.5.4 Käyttöliittymän visuaaliset testit

Ennen jokaista testiä suoritetaan kaikille testeille yhteiset toimenpiteet, jotka määritellään `beforeEach`-kutsulla. Sen tarkoituksena on asettaa testauksen lähtötilanne ja alustaa tarvittavat testeille yhteiset resurssit. Ensin käynnistetään Puppeteer-selain ja avataan uusi välilehti. Seuraavaksi määritellään testeille selainikkunan koko, joka simuloi tiettyä näyttöresoluutiota. Lopuksi navigoidaan annetulle verkkosivulle, käyttöliittymän sivulle, jossa esimerkkisivu sijaitsee. Toimenpiteet voivat olla myös testikohtaisia, jolloin yhteistä alustusta ei tarvita. Testien ryhmittelyyn käytetään Vitest:in `describe`-lohkoja. Jokainen lohko määrittelee testijoukon, joka liittyy tiettyyn toiminnallisuuteen, komponenttiin tai ominaisuuteen. Näin lähdekoodi pysyy luettavana ja tietyn testijoukon ajaminen kerrallaan on mahdollista. Jaotellaan esimerkkitestit kolmeen lohkoon:

Visual fullpage snapshot tests (3 testiä) sisältää kokonaisten testisivujen snapshot-testauksen. Kirjoitetaan testit aiemmin määritellylle, työpöytäversiota vastaavalle selainikkunan koolle se-

kä kahdelle pienemmälle ruudulle, mobile- ja tablet-kokoisille näytöille.

Visual component snapshot tests (7 testiä) sisältää erillisten komponenttien testauksen. Tutkitaan primääripainikkeen, otsikon, taulukon ja primäärivärin olemassaoloa sivulla, infoboksin `hover`-muotoa ja dialogin sekä ponnahdusikkunan visuaalista tilaa painikkeen painalluksen jälkeen.

Visual component tests without snapshots (3 testiä) ei käytä kuvakaappauksia, mutta tutkii komponenttien visuaalista ilmettä. Tässä osiossa tutkitaan otsikon korkeutta, väriä ja siinä olevaa tekstiä erilaisin metodein.

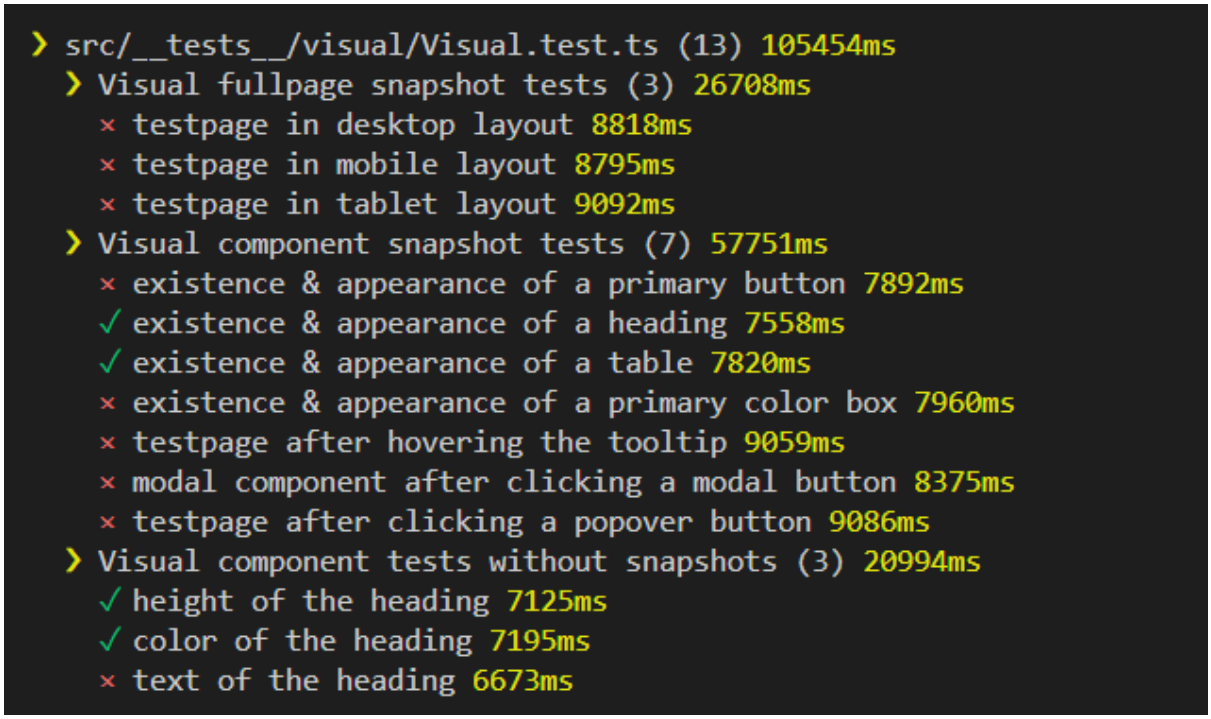
Jotta testikomponentit voi löytää, niillä täytyy olla uniikki tunniste. Esimerkiksi `button`-elementtejä saattaa olla sivulla monta ja testi valitsee näistä ensimmäisen. Elementtien sijaan kannattaakin käyttää tarkempaa tunnistetta, kuten id:tä tai luokkaa. Tämä täytyy ottaa huomioon tyylioppaan komponenteissa ja lisätä testausta varten tarvittavat tunnisteet.

Ensimmäisellä testiajolla syntyvät snapshot-testien osalta referenssikuvat. Tehdään sivun komponentteihin muutoksia ja tarkastellaan ohjelman löytämiä tarkoituksellisia virheitä. Käytännössä koko sivun testien pitäisi ilmoittaa kaikista snapshot-poikkeamista, koska testattavat komponentit sijaitsevat kaikki samalla sivulla. Yksittäisten komponenttien testeillä päästään kuitenkin lähemmin tarkastelemaan niissä olevia virheitä. Tarkoitukselliset virheet ovat seuraavat:

- lisätään otsikkoon sanojen väliin kaksi välilyöntiä
- lisätään tekstikappaleen perään lisää tekstiä
- muutetaan ensimmäisen värilaatikon korkeutta
- lisätään primääripainikkeeseen täytettä
- lisätään taulukolle ylämarginaalia
- poistetaan dialogista `isCentered` ja muutetaan yhden painikkeen tekstiä
- kasvatetaan ponnahdusikkunan avauspainikkeen kokoa ja yhden painikkeen tekstiä
- vaihdetaan viimeisen herätteen status ja otsake.

Ajetaan kaikki testit uudelleen ja odotetusti suurin osa testeistä epäonnistuu. Otsikon ulkoasutestin *existence and appearance of a heading* olisi luullut aiheuttavan virheen, mutta Chakra:n `Heading`-komponentti poistaa ylimääräiset välilyönnit renderöintivaiheessa. Otsikon tekstisisältö kuitenkin on poikkeava, siitä kertoo epäonnistunut *text of the heading* -testi. Taulukolle annettu marginaali ei aiheuta muutosta komponentin itsensä ulkoasuun, mutta virhe jää kiinni

koko sivun testeissä. Kuviossa 23 ovat ensimmäisen testiajon tulokset.



Kuvio 23. Ensimmäisen varsinaisen testiajon tulokset.

Tarkastellaan epäonnistuneita testejä ja niistä syntyneitä poikkeamakuvia ja virheviestejä seuraavassa. Suurin osa viesteistä kertoo kuvakoon poikkeamasta, koska useat tehdyistä muutoksista vaikuttavat kuvan kokoon. Vitest raportoi virheistä siinä järjestyksessä, kuin ne testiajossa esiintyvät. Jos testissä on useampi virhe, niistä raportoidaan ensimmäinen. Pikselieroista raportointi toisi käyttäjälle lisäarvoa. Kuvakoon poikkeaman voi määritellä hyväksyttäväksi virheeksi antamalla `configureToMatchImageSnapshot`-funktiolle ominaisuus `allowSizeMismatch: true`. Tällöin testi ohittaa kokoeron ja raportoi käyttäjän kannalta ehkä tärkeämpää tietoa. Kuvista voidaan kuitenkin havaita kaikki testin virheet.

T1. testpage in desktop layout. *Error: Expected image to be the same size as the snapshot (1200x1300), but was different (1200x1312).* Kuviossa 24 on esitetty testin poikkeamat, vasemmalla referenssikuva, oikealla todellinen kuva ja keskellä virheet. Koko sivua koskeva työpöytänäkymätesti paljastaa lähes kaikki virheet. Komponenttien kokoon on tullut muutoksia, mikä vaikuttaa sivun asetteluun. Sisältömuutokset näkyvät kuvassa myös, lukuun ottamatta otsikon ylimääräisiä välilyöntejä, koska ne eivät kuvassa toteudu.

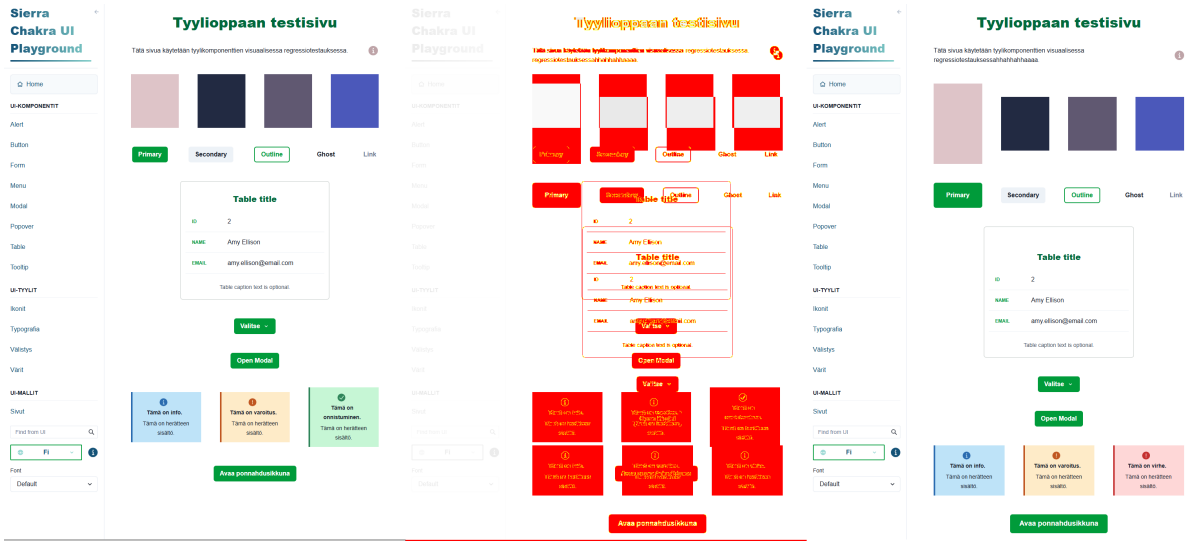
Kuvio 24. Koko testisivun työpöytä-näkymän poikkeamakuva (T1).

T2. testpage in mobile layout *Error: Expected image to be the same size as the snapshot (375x2199), but was different (375x2320).* Mobiilinäkymän testi näyttää samat virheet kuin koko kuvassa. Poikkeamakuvasta (kuvio 25) voidaan myös havaita mahdolliset ongelmat sivun responsiivisuudessa.

Kuvio 25. Koko testisivun mobile-näkymän poikkeamakuva (T2).

T3. testpage in tablet layout *Error: Expected image to be the same size as the snapshot (1024x1366), but was different (1024x1370).* Testin virheviesti koskee kuvan kokoa, joka on

vain 4 pikselin ero korkeudessa. Sallimalla kuvan kokoerot, saataisiin selville tärkeämpää tietoa pikselipoikkeamista. Responsiivinen asettelu voidaan kuitenkin tarkastaa kuviosta 26.

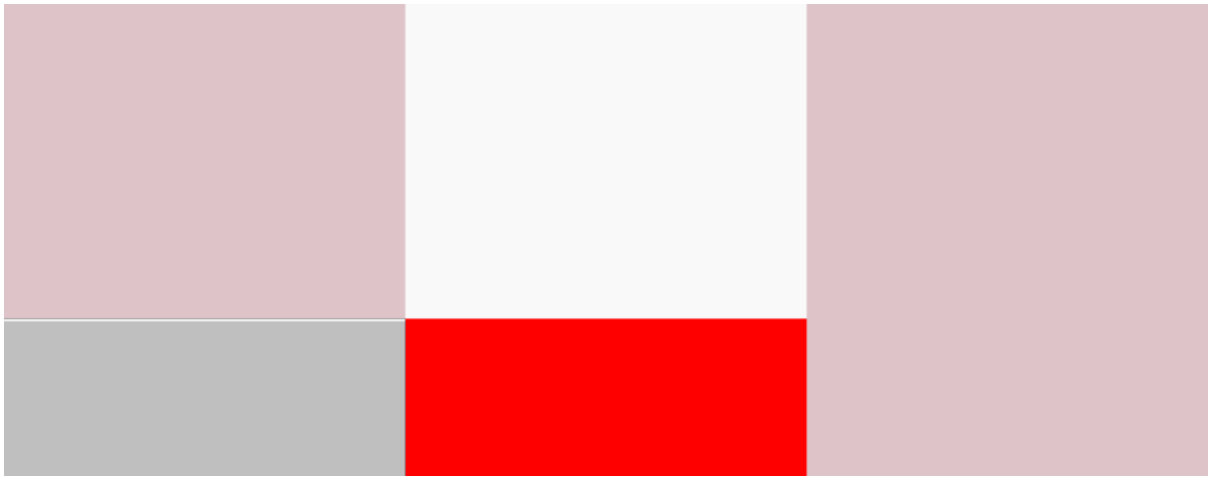


Kuvio 26. Koko testisivun tablet-näkymän poikkeamakuva (T3).

T4. existence & appearance of a primary button *Error: Expected image to be the same size as the snapshot (92x40), but was different (124x64).* Painikkeelle annettu täyte näkyy selkeästi kuviossa 27. Painikkeen koko ja tekstin paikka muuttuu.

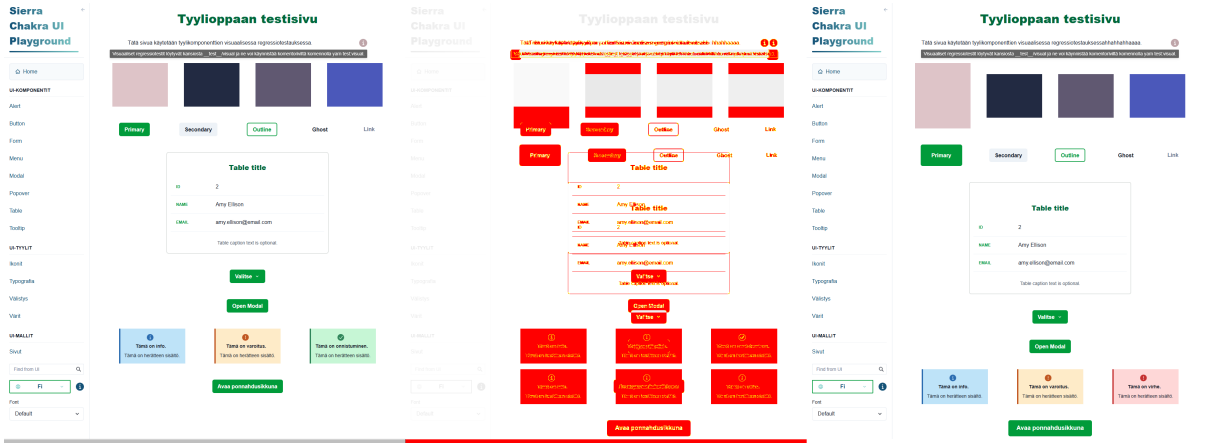
Kuvio 27. Primääripainikkeen ulkoasutestin (T4) poikkeamakuva.

T7. existence & appearance of a primary color box *Error: Expected image to be the same size as the snapshot (166x130), but was different (166x195).* Elementin muuttunut koko nähdään selkeästi myös kuviosta 28. Tähän testiin voisi yhdistää myös komponentin värin tarkastuksen ilman kuvakaappausta, kuten testissä T12. Snapshot-testi paljastaisi muuttuneen värin kuitenkin myös, mutta erillinen väritesti antaisi lisäinformaatiota.



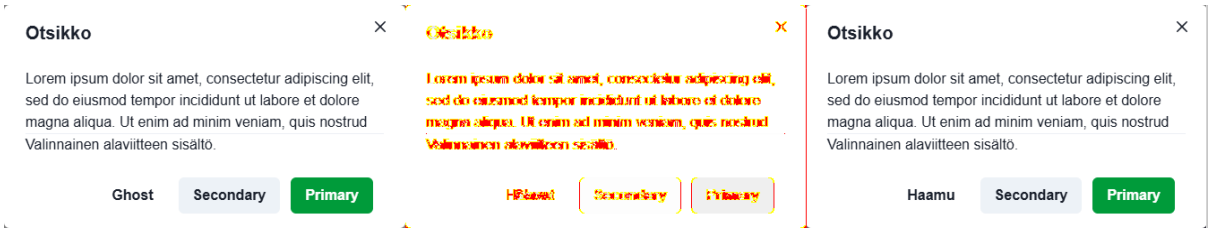
Kuvio 28. Primäärivärilaatikon ulkoasutestin (T7) poikkeamakuva.

T8. testpage after hovering the tooltip *Error: Expected image to be the same size as the snapshot (1200x1300), but was different (1200x1312).* Testin tulos on esitetty kuviossa 29. Animaatioiden osalta kannattaa testiin lisätä ajastin tai jollain muulla tavalla varmistaa, että animaatio on valmis kuvaa otettaessa. Luonnollisesti sivulla olevat muutkin virheet näkyvät kuvassa, tarkoituksenmukaisempaa olisi tarkastella komponenttia erillään.



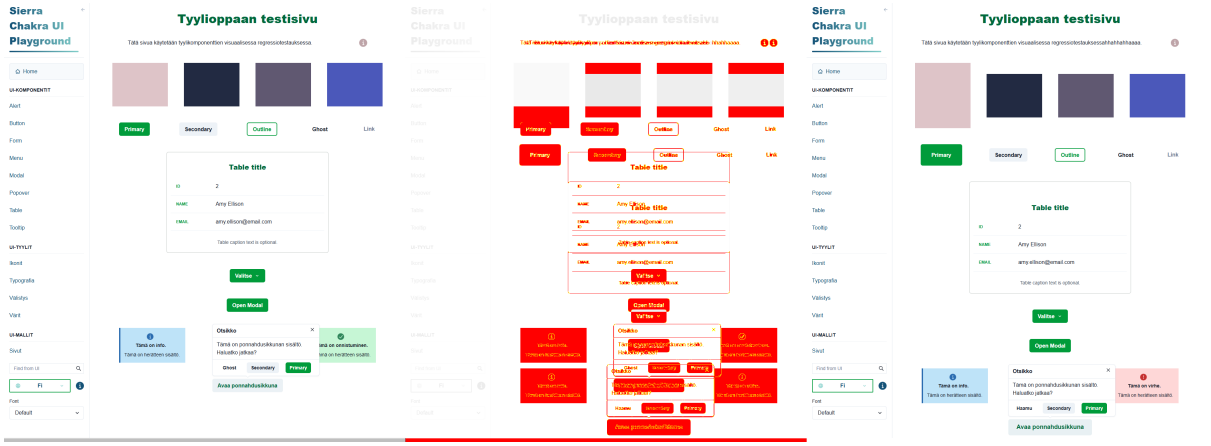
Kuvio 29. Infolaatikon hover-testin (T8) poikkeamakuva.

T9. modal component after clicking a modal button *Error: Expected image to be the same size as the snapshot (448x247), but was different (446x247).* Dialogille tehty sijainnin muutos ei paljastu poikkeamakuvassa (kuvio 30), koska testin kohteena on pelkkä komponentti. Kuvaan piirtyy sen sijaan paljon ylimääräistä. Ensimmäisen painikkeen tekstin muutos valitettavasti hukkuu kohinaan.



Kuvio 30. Dialogitestin (T9) poikkeamakuva.

T10. testpage after clicking a popover button *Error: Expected image to be the same size as the snapshot (1200x1300), but was different (1200x1312).* Ponnahdusikkunatestin lisäarvo visuaalisen testauksen kannalta on vähäinen, koska testataan jälleen koko sivua, kuten kuviosta 31 nähdään. Toiminnallisuuden osalta voidaan kuitenkin todeta, että ikkuna on avautunut painikkeen painalluksen myötä. Jos muita virheitä ei sivulla olisi, tulos olisi huomattavasti käyttökelpoisempi.

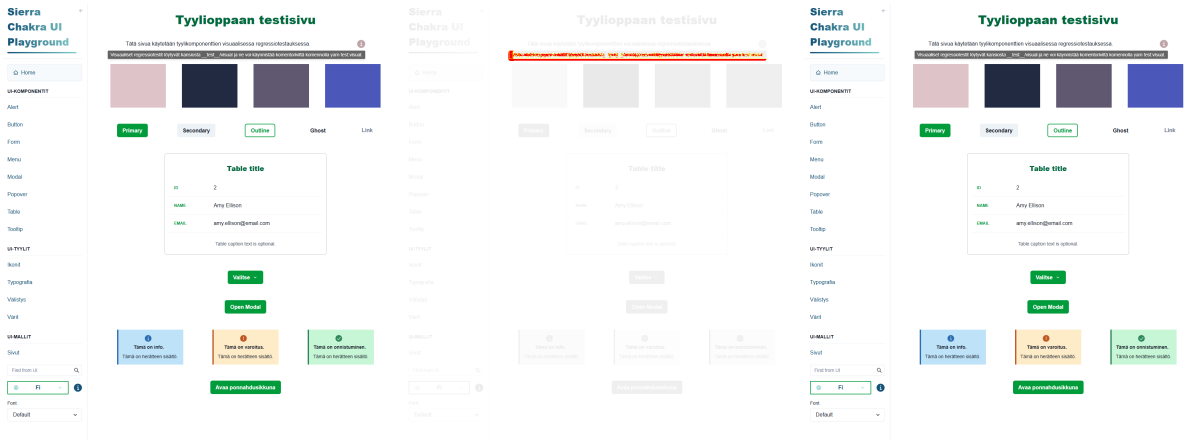


Kuvio 31. Ponnahdusikkunatestin (T10) poikkeamakuva.

T13. text of the heading *AssertionError: expected 'Tyylioppaan testisivu' to be 'Tyylioppaan testisivu'.* Ilman kuvakaappausta toteutetut testit kertovat virheestä sanallisesti. Virheviesti on informatiivinen ja sen avulla päästään virheeseen helposti käsiksi.

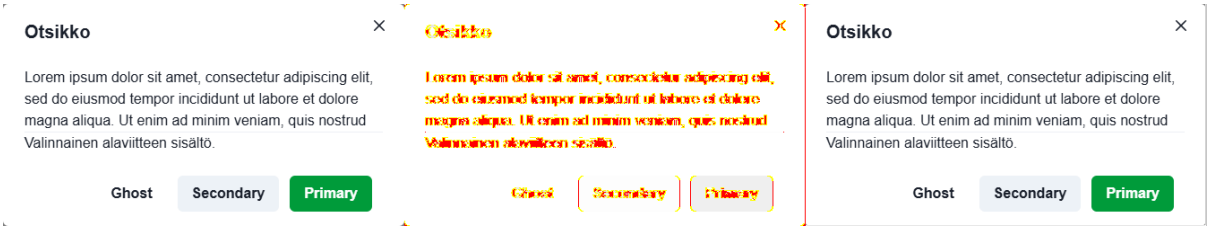
Korjataan tehdyt muutokset takaisin alkuasetelmaan ja ajetaan testit uudelleen. Nyt kaikkien testien odotetaan onnistuvan. Kuitenkin kolme tapahtumia sisältävää testiä epäonnistuu edelleen (testit T8, T9 ja T10).

T8. testpage after hovering the tooltip *Error: Expected image to match or be a close match to snapshot but was 0.4953846153846154% different from snapshot (7728 differing pixels).* Nyt Vitest raportoi vajaan puolen prosentin pikselipoikkeamasta. Komponentissa ei pitäisi enää olla virhettä, mutta kuviossa 32 kuitenkin korostuu virhe komponentin hover-tilassa.



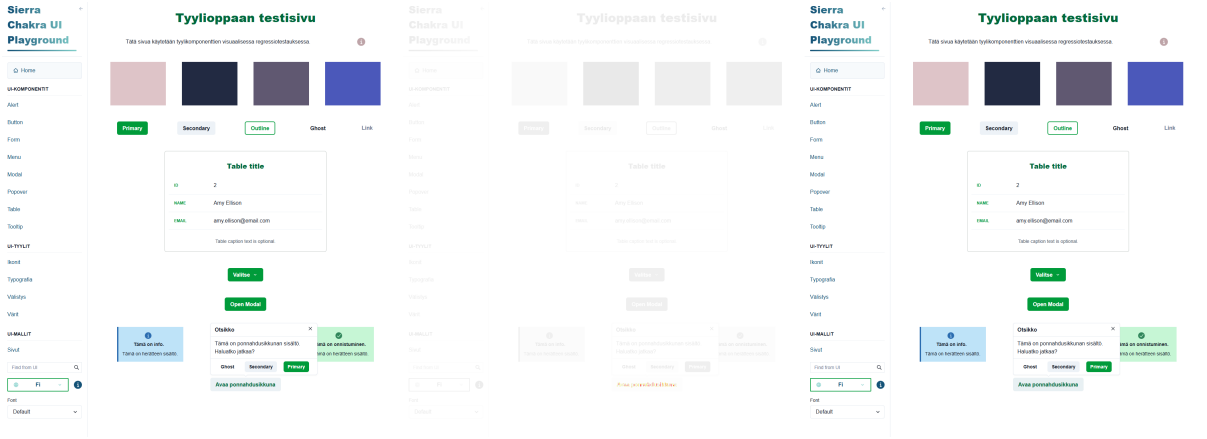
Kuvio 32. Infolaatikon hover-testin (T8) toinen poikkeamakuva.

T9. modal component after clicking a modal button *Error: Expected image to be the same size as the snapshot (448x247), but was different (446x246).* Samoin dialogin osalta raportoidaan edelleen poikkeamasta, jota ei pitäisi enää olla (kuvio 33).



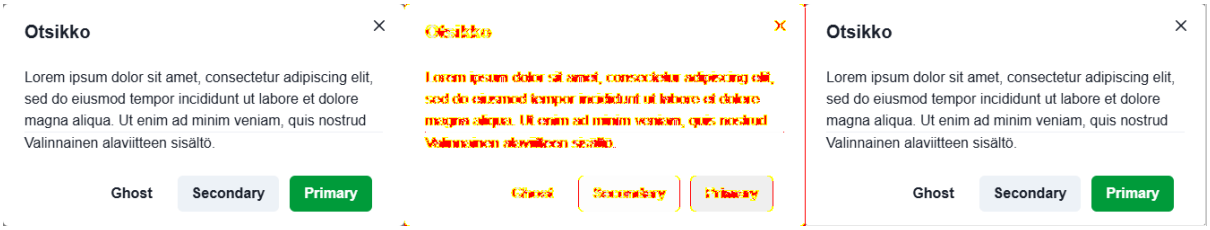
Kuvio 33. Dialogitestin (T9) toinen poikkeamakuva.

T10. testpage after clicking a popover button *Error: Expected image to match or be a close match to snapshot but was 0.008397435897435% different from snapshot (131 differing pixels)* Ponnahdusikkunan tapauksessa virhe on niin pieni, että se on tuskin havaittavissa (kuvio 34).



Kuvio 34. Ponnahdusikkunatestin (T10) toinen poikkeamakuva.

Toisen testiajon kuvista päätellen poikkeamat ovat vääriä hälytyksiä. Virheraportista nähdään myös erojen olevan hyvin pieniä. Kasvatetaan näiden testien osalta virhemarginaalia ja ajetaan testit uudelleen. `Modal`-komponentin testi (T9) epäonnistuu edelleen, muut menevät läpi. Epäonnistuneen testin poikkeamakuvasta (kuvio 35) kuitenkin nähdään, ettei kyse ole todellisesta virheestä.



Kuvio 35. Dialogitestin (T9) kolmas poikkeamakuva.

4.5.5 Työkalun soveltuvuus tyylinvalvojaksi

Vitest, jest-image-snapshot:in ja Puppeteer-kirjaston avustuksella voisi toimia käyttöliittymäkomponenttien visuaalisessa regressiotestauksessa, mutta kohdeorganisaation tapauksessa jatkotutkimusta vaaditaan. Epäjohdonmukaiset väärät hälytykset ja vajavainen raportointi vaikuttavat testien luotettavuuteen. Testien syntaksi on kuitenkin selkeää ja esimerkkejä löytyy dokumentaatiosta. Vaikka suurin osa virheistä jää kiinni kokonaisten sivujen kuvissa, on komponentteja kuitenkin syytä tarkastella myös erikseen. Tällöin poikkeamat tulevat selkeämmin esiin ja testiajoja voidaan täsmentää ongelmakohtiin.

Tätä tutkimusta varten tehty simulaatio on toki keinotekoinen, koska virheet ovat ennalta tie-

dossa. Väärien virheiden erottaminen todellisista virheistä saattaa ainakin aluksi tuntua työläältä. Toleranssin kasvattaminen testikohtaisesti voi auttaa, kun opitaan tunnistamaan virheet, jotka ovat ihmissilmälläkin havaittavissa. Tekoälypohjaiset ratkaisut vastaavat juuri tähän ongelmaan ja osaavat erotella merkitykselliset visuaaliset erot pienistä renderöintieroista.

Vitest soveltuu myös toiminnalliseen testaukseen ja onkin siltä osin jo kohdeorganisaation käytössä. Se on myös integroitavissa yleisesti käytettyihin CI/CD-työkaluihin. Visualisen testaustyökalun tehokkuus ja suorituskyky jatkuvan ohjelmistokehityksen putkessa on kuitenkin arvioitava erikseen.

5 Artefaktin evaluointi ja tulokset iteraatioittain

Suunnittelusyklien iteraatioiden päätteeksi suoritetaan artefaktin evaluointi. Tässä luvussa esitellään arvioinnin tulokset ja niiden perusteella suoritettavat jatkotoimenpiteet.

5.1 Tyyliopas, ensimmäisen evaluoinnin tulokset

Alla kuvataan ensimmäisen evaluoinnin tuloksia laatuvaatimusten täyttymisen ja tyylioppaan lähdekoodin osalta. Tyylioppaalle asetettiin seuraavat laatuvaatimukset: toiminnallisuus, käytettävyys, tehokkuus ja ylläpidettävyys. Toiminnallisuudella tarkoitetaan sitä, että tyyliopas soveltuu käyttöön, johon se on tarkoitettu ja hoitaa sille määritellyt toimenpiteet. Käytettävyys puolestaan tarkoittaa tyylioppaan helppokäyttöisyyttä, eli sitä, että se on riittävän intuitiivinen ja opittava, tehokas ja yksinkertainen käyttää. Tehokkuus tarkoittaa, että tyyliopas ei hukkaa käyttäjänsä aikaa tai järjestelmän resursseja ja toimii tehokkaasti siinä tarkoituksessa, johon se on luotu. Ylläpidettävyys liittyy tyylioppaan elinkaareen ja siihen, ettei se ole kertaluontoinen projekti vaan elää sovelluksen mukana ja pysyy ajan tasalla.

Näitä asioita tarkastellaan tyylioppaan toteutusta evaluoitaessa, vaatimusmäärittely mielessä pitäen. Arvioinnin suorittaa kohdeorganisaation ohjaaja. Ensimmäisen evaluoinnin pohjalta toteutetaan toinen iteraatio tyylioppaan kehityksestä, jossa alla mainitut asiat pyritään korjaamaan. Iteraation päätteeksi tyyliopas ja sen lähdekoodi arvioidaan uudelleen.

5.1.1 Laatuvaatimusten täyttyminen

Toiminnallisuuden osalta arvioija pitää tyyliopasta selkeänä, uskoo käyttäjän saavan nopeasti yleiskuvan siitä, mitä on tarjolla ja tuovan käyttöliittymäkehittäjien työhön selkeyttä. Organisaatiotasoisten komponenttien valmiiksi rajoitetun spesifioinnin uskotaan helpottavan kaikkien työtä.

Jatkotoimenpiteitä aiheuttavat ensimmäisen evaluoinnin perusteella muutamat seikat. Tyylioppaan olisi hyvä olla näkyvillä myös kirjautumattomille käyttäjille. Ensimmäisessä tyylioppaan versiossa esitellään myös joitakin ei-spesifejä komponentteja, kuten `Input`, joiden tarpeellisuus jää vähäiseksi. Jotta kokonaisuus olisi tiivis, nämä on syytä poistaa. Lisäksi ensimmäisen version lomakekomponentti, vaikkakin toimii yksinkertaisen lomakkeen luontiin, vaatii muokkausta. Käyttökelpoisempi olisi skeemapohjainen lomakegeneraattori, jossa myös sarakkeiden

käyttäminen olisi mahdollista. Arvioija mainitsee lyhyesti myös testauksesta, jossa tyylioppaan esimerkkisivun mallia vastaan voitaisiin testata komponenteissa tehtyjä muutoksia. Kyseessä olisi näin ollen visuaalinen regressiotestaus snapshot-tekniikalla. Väripalettiin ja sen käyttöönoton vaikutuksiin on palattava seuraavissa vaiheissa. Ajatuksena on, että Sierra-teemassa määritellään sovelluksen perusvärit käyttäen tyylioppaan jaottelua (primary, secondary jne), jotka voidaan ylikirjoittaa asiakaskohtaisesti teematiedostossa (`customTheme`). Näiden väripalettien käyttöönotto vaatii merkittäviä muutoksia komponenttikirjaston nykyisessä rakenteessa ja herättää keskustelua uuden, organisaation brändiin perustuvan demosovelluksen luomisesta. Kuviossa 36 on esitetty evaluoinnin tulokset toiminnallisuuden osalta, sekä vaadittavat jatkotoimenpiteet ja toteutus.

Huomiot	Jatkotoimenpiteet ja toteutus
Tyyliopas on selkeä, sekä antaa ensisilmäyksellä kokonaiskuvan siitä mitä työkaluja ja sisältöä käyttäjälle on tarjolla.	Ei jatkotoimenpiteitä.
Sisältö ottaa huomioon organisaatiotasoiset komponentit niin graafisine esittelyineen kuin tarkempine ominaisuuksineen. Tyyliopas luo hyvän rungon käyttöliittymäkehittäjien työhön selkeyttäen tekemistä yksinkertaistamalla komponenttitasolla tarvittavaa konfigurointia.	Ei jatkotoimenpiteitä.
Organisaatiotasoisten komponenttien rajoittaminen vain vaadittuihin ominaisuuksiin helpottaa kaikkien työtä ja tätä lähestymiskulmaa on jo hyvin hyödynnetty mm. sivu-templaattien yhteydessä.	Ei jatkotoimenpiteitä.
Tyylioppaan on hyvä olla näkyvissä myös kirjautumattomille käyttäjille. Kirjautuminen on tarkoitettu niitä organisaation komponentteja varten, jotka tarvitsevat dataa organisaation sisäisistä järjestelmistä.	Tyylioppaan sivut siirretty näkyville aloitusvalikkoon, eivätkä ne vaadi enää kirjautumista.
Nykyisessä versiossa on tiettyjä komponentteja, jotka eivät ole ns. organisaatio-spesifejä. Esimerkkinä <Input>-komponentti, jonka esittämisen tarpeellisuutta tulee arvioida, sillä kyseessä on vakiotasoinen kolmannen osapuolen komponentti, jonka dokumentaatio on saatavilla lähteen kautta.	Poistettu komponentit, jotka eivät ole Akamon-spesifejä. Input on ainoa sellainen. Voidaan keskustella Alertin merkityksestä, onko siinä jotain muuta specifiä kuin muotoilu, joka voitaisiin hoitaa teematiedostossa?
Nykyinen lomake-työkalu toimii hyvänä pohjana. Jatkossa tässä tulee huomioida skeema-pohjainen lomakkeen luonti skeeman validointi mukaan lukien, sekä mahdollistaa mm. sarakkeet.	Skeemapohjainen lomake on toteutettu käyttäen react-hook-formia ja zod-kirjastoa. AkamonFormille annetaan myös sarakkeiden lkm, jolloin kenttiä voidaan asetella myös rinnakkain.
Testausta voidaan hyödyntää luomalla esim. snapshot esimerkkisivusta, jossa yleisimmät komponentit käytössä. Testaus tätä kuvaa / kuvia vasten	Keskustellaan tästä tarkemmin, kun testausta aletaan luomaan. Esimerkkisivulle kootaan tyylioppaan komponentit ja jokaisen muutoksen yhteydessä suoritetaan visuaalinen regressiotestaus.
Väripaletit ja niiden käyttöönotto. Vaatii muutoksia sekä komponenttikirjastoon, että kaikkiin sitä käyttäviin sovelluksiin.	Väripalettien suunnittelu ja niiden käyttöönotto on mietittävä erikseen. Tyylioppaassa esitellään sovelluksen perusvärit, jotka voidaan ylikirjoittaa asiakastoteutuksissa.

Kuvio 36. Evaluointi I, toiminnallisuus: huomiot ja jatkotoimenpiteet

Käytettävyydestä projektin tässä vaiheessa mainitaan vain, että tyyliopas on selkeä käyttää. Ohjaaja mainitsee dokumentaation löydettävyyden kuitenkin kärsivän tyyliohjeen ollessa pelkästään Github-repositoriossa. Julkaisua ulkoverkkoon voidaan harkita, jolloin tulee arvioida uudestaan myös `Webscraper`-komponentin käyttö ja alkuperäiset tekijänoikeudet. Tyylioppaan komponenttien esittelyistä puuttuvat koodiesimerkit. Nämä ovat edellytyksenä tyylioppaan käytettävyydelle. Kuviossa 37 on esitetty evaluoinnin tulokset käytettävyyden osalta, sekä vaadittavat jatkotoimenpiteet ja toteutus.

Huomiot	Jatkotoimenpiteet ja toteutus
Tyyliopas on projektin tässä vaiheessa selkeä käyttää.	Ei jatkotoimenpiteitä.
Dokumentaation löydettävyys kärsii ohjeen ollessa pelkästään GitHub repositoriossa. Löydettävyyden lisäämiseksi tulee arvioida komponenttikirjaston playground-projektin julkaisua ulkoverkkoon. Tällöin kysymysmerkkinä webscraperin käyttö ja alkuperäisen sisällön tekijänoikeudet.	Aihe jää keskusteltavaksi ja lopulta organisaation päätettäväksi.
Komponenttien koodiesimerkit puuttuvat.	Dynaamisesti komponenttikoodin perusteella syntyvät koodiesimerkit ovat ehdoton edellytys tyylioppaan käytettävyydelle. Ratkaisu on toteutettu tuomalla komponenttikoodi merkkijonona käyttäen Viten '?raw'-liitettä. Koodin korostukseen ja tyylittelyyn käytetään 'react-syntax-highlighter'ia.

Kuvio 37. Evaluointi I, käytettävyys: huomiot ja jatkotoimenpiteet

Tehokkuudesta arvioija mainitsee, että kokonaisuus on nopeasti selattavissa ja haettava tieto on helppo löytää. Sivujen jakaminen kehittäjien välillä on mahdollista, koska sovelluksen reitit kulkevat osoiterivillä. Lisäksi arvioija esittää toiveen tyylioppaan sisällön indeksoinnista ja hakutoiminnosta, kun ominaisuuksien määrä kasvaa. Kuviossa 38 on esitetty evaluoinnin tulokset tehokkuuden osalta, sekä vaadittavat jatkotoimenpiteet ja toteutus.

Huomiot	Jatkotoimenpiteet ja toteutus
Kokonaisuus on nopeasti selattavissa ja haettavan tiedon paikkaa ei tarvitse arpoa. Eri sivuja voidaan jakaa kehittäjien välillä, sillä sovelluksen reitit kulkevat osoiterivillä.	Ei jatkotoimenpiteitä.
Ominaisuuksien kasvaessa sisällön indeksointi & haku olisi tarpeellinen.	Sivuvalikkoon lisätty hakutoiminnallisuus UI-osion sivuihin. Pitäisikö haun koskea kaikkea kirjaston sisältöä? Hakukenttä on nyt sivun lopussa, onko se oikea paikka?

Kuvio 38. Evaluointi I, tehokkuus: huomiot ja jatkotoimenpiteet

Tyylioppaan **ylläpidon** arvioija uskoo olevan helppoa, koska kaikki sisältö löytyy saman repositorion sisältä. Hän huomioi kuitenkin ylläpidon olevan manuaalista, johon helpotusta voisi tuoda esimerkiksi automaattinen koodin generointi lähdekoodin perusteella. Kun tyylioppaaseen lisätään komponentteja, niiden esitys ja dokumentaatio on luotava manuaalisesti. Evaluoinnin tuloksia läpikäydessä keskustellaan siitä, mikä tämän ominaisuuden merkitys on. Vaikkei tyyliopasta voitaisi pitää ajan tasalla päivittäin, ajoitettaisi säännöllisesti tyylikomponenttien ja oppaan tarkistus, jolloin kerralla päivitettäisiin vastaamaan toisiaan. Samalla tehtäisiin muutoksia jo olemassa oleviin tyylioppaan sivuihin, mikäli komponentteihin on tehty muutoksia. Kuviossa 39 on esitetty evaluoinnin tulokset ylläpidon osalta, sekä vaadittavat jatkotoimenpiteet ja toteutus.

Huomiot	Jatkotoimenpiteet ja toteutus
Tyylioppaan ylläpito on yksinkertaista repositorion sisältäessä sekä tyylioppaan, kehitystyökalut kuin myös organisaatiotasoiset komponentit. Näiden välillä hyödynnetään aliaksia, joilla tiedostojen polkuja voidaan hyödyntää dynaamisesti ei projektien välillä.	Ei jatkotoimenpiteitä.
Ylimääräistä helpotusta voidaan lisätä luomalla automaattista koodin generointia lähdekoodin perusteella. Generoitavaa koodia voi olla esim. koodiesimerkkien sisältö.	Tyylioppaan esityksen osalta turha toisto on poistettu. Jatkoa ajatellen, kun tyylioppaaseen lisätään komponentteja, niiden esitys on toteutettava manuaalisesti. Tämä oli tietoinen valinta, kun Storybook työkaluna hylättiin.

Kuvio 39. Evaluointi I, ylläpito: huomiot ja jatkotoimenpiteet

5.1.2 Lähdekoodin evaluointi

Myös tyylioppaan komponenttien lähdekoodi arvioidaan. Osa arvioiduista tiedostoista sisältää tyylioppaan esitykseen liittyvää koodia, mutta valtaosa arvioista kohdistuu kuitenkin organisaatiospesifeihin komponentteihin. Templaattien osalta arvioija ehdottaa oletusvälistyksen määrittelemistä, jolloin komponenttia kutsuttaessa ei välttämättä tarvitsisi antaa välistysarvoa. Herätteestä arvioija mainitsee, ettei status-parametria tarvitsisi erikseen määritellä, koska se kuuluu Chakra UI:n parametreihin. Näin ollen spesifi status-parametri ei anna mitään lisäarvoa. Lisäksi arvioija ehdottaa tyypin `RequireAtLeastOne` käyttämistä otsikon ja kuvauksen osalta. On kuulemma tavallista, että joissain yhteyksissä käytetään vain toista, jolloin joko otsikko tai kuvaus annetaan. `BrandTablen` value-ominaisuuteen on jäänyt tyypiksi `any`, mikä saattaa aiheuttaa ongelmia. Tämä on syytä korjata vastaamaan sallittuja tyyppejä. Lomakekomponentista arvioija kehottaa tekemään skeemapohjaisen version, kuten toiminnallisuuden arvioinnissa mainitaan. Taulukon osalta avaintietona käytetty indeksi saattaa myös aiheuttaa ongelmia. Arvioija ehdottaa kentän nimen käyttämistä avaimena indeksin sijaan. Tyylioppaan esityssivuilla on jonkin verran toistoa, josta on pyrittävä eroon. Samoin kehitysvaiheessa asennetut riippuvuudet, jotka eivät tässä versiossa ole enää käytössä, on syytä poistaa. Esityksessä olevien esimerkkitaulukoiden testidata olisi hyvä muuntaa JSON-muotoon ja siirtää omaan kansioonsa. Näin samaa dataa voitaisiin käyttää myös sovellusta testatessa. Tyylitiedoston osalta toivotaan jaettujen ominaisuuksien määrittelemistä ainoastaan yhdessä paikassa, jolloin niiden ylläpito helpottuu.

5.2 Tyyliopas, toisen evaluoinnin tulokset

Toinen evaluointi on luonnollisesti ensimmäistä kevyempi, koska arvioidaan pääasiassa ensimmäisen evaluoinnin perusteella tehtyjä muutoksia. Laatuvaatimusten osalta merkittävimpiä muutoksia ovat seuraavat:

Toiminnallisuus:

- Tyylioppaan näkyvyys kirjautumattomalle käyttäjälle: tyylioppaan käyttö ei enää vaadi kirjautumista. **Toteutus hyväksytty.**
- Ei organisaatio -spesifien komponenttien poistaminen tyylioppaasta: kohdeorganisaatio edelleen ehdottaa painikekomponentin poistamista tyylioppaasta. Tosiasia on, ettei painike ole organisaatiospesifi komponentti, mutta varianttivaihtoehtojen esittely auttaa kehittäjiä hahmottamaan, millaista painiketta kulloinkin tulee käyttää. Variantit ovat kuitenkin organisaatiospesifejä, vaikka itse komponentti ei olekaan. Tämän vuoksi painike jätetään osaksi tyyliopasta. **Toteutus hyväksytty.**
- Skeemapohjaisen lomakkeen luonti: kohdeorganisaatio esittää muutoksia ensimmäisen evaluoinnin perusteella toteutetulle komponentille. Sarakkeiden olisi hyvä olla tietomallissa mukana. Esimerkkinä annetaan JSON-skeema ja ehdotetaan sen käyttämistä rekursiolla. **Toteutusta muutetaan.**
- Väripaletin suunnittelu ja sen käyttöönotto: ehdotettu toteutus väripaletin käytöstä antaa selkeän suuntaviivan toteutukselle. **Toteutus hyväksytty.**

Käytettävyys:

- Komponenttien koodiesimerkkien dynaaminen luonti: tyylioppaan sivuilla on näkyvillä kunkin komponentin alla dynaamisesti lähdekoodista luotu koodiosa. **Toteutus hyväksytty.**

Tehokkuus:

- Hakutoiminnallisuuden toteuttaminen tyylioppaan komponenteille: haku komponenttien nimillä on kohdeorganisaation mukaan riittävä. **Toteutus hyväksytty.**
- Hakutoiminnallisuuden sijainti tyylioppaassa: haku on paremmin tarjolla ja löydettävissä jos se on heti listan alussa. **Toteutusta muutetaan.**

Ylläpidettävyys:

- Tyylioppaan esityksen manuaalinen luonti: todetaan, että uusien komponenttien tapauksessa esitysmuoto on toteutettava manuaalisesti. **Toteutus hyväksytty.**

Koska laatuvaatimusten todetaan täyttyneen riittävällä tasolla, toisen evaluoinnin pääpaino on lähdekoodin arvioinnissa. Arvioidaan mm. toisteisuutta, jota tyylikomponenttien esitysmuo-

dossa paikoitellen on. Tämän perusteella rakennetaan `FetchPropsHook`, joka suorittaa ominaisuuksien noudon `Webscraper`-komponenttia käyttäen Chakra UI:n dokumentaatiota ja palauttaa ne järjestelmään kunkin komponentin esitystä varten.

Arvioinnissa huomioidaan myös Chakra:n ominaisuuksien liian vapaa käyttö, kun toteutuksessa käytetään tapaa laajentaa Chakra:n ominaisuuksia tarkoituksen kulloinkin vaatimilla tapauskohtaisilla ominaisuuksilla (`...extends AlertProps`). Tämä valitettavasti sallii tarpeettomien ominaisuuksien lisäämisen vahingossa ja kohdeorganisaatio ehdottaakin komponentin omien ominaisuuksien lisäksi valinnaista `customProps`:ia, jolle annetaan tyypiksi Chakra:n ominaisuudet (`customAlertProps? : AlertProps`). Näin ollen kehittäjän täytyy halutessaan tietoisesti valita ominaisuus komponentille määriteltyjen omien ominaisuuksien ulkopuolelta. Muutos vaikuttaa järkevältä ja se toteutetaan viiteen tyylioppaan komponenttiin samalla lailla.

Edellisten lisäksi kohdeorganisaatio mainitsee kymmenkunta muutoskohtaa, jotka ovat luonteeltaan keskenään erilaisia: rakennetaan takaisinkutsuja, eriytetään komponenttien sisällä olevia komponentteja omiksi tiedostoikseen, tarkistetaan responsiivisuutta ja pyritään mahdollisuuksien mukaan luopumaan kaikesta kovakoodatusta sisällöstä. Toisen evaluoinnin pohjalta toteutettuja muutoksia ei tämän tutkimuksen puitteissa evaluoida enää uudelleen. Kohdeorganisaatio arvioi lopullisen toteutuksen kokonaisuutena siinä vaiheessa, kun seuraavat vaiheet, ohjeet rekonstruktiosta sekä tyylinvalvonta on suoritettu.

5.3 Tyylinvalvoja, ensimmäisen evaluoinnin tulokset

Kohdeorganisaation edustaja arvioi tyylinvalvojalla olevan tarvetta jatkokehitykselle. Kaiken kaikkiaan hän pitää tutkimuksen tätä osaa kattavana kokonaisuutena eri visuaalista testausta hyödyntävistä testauskirjastoista ja niiden lähtökohtaisesta toiminnallisuudesta. Tutkimus antaa selkeän kuvan niin toiminnallisesta puolesta kuin myös kirjastojen mukanaan tuomista haasteista.

Vitest:in ollessa käytössä järjestelmän toiminnallisessa testauksessa, on loogista hyödyntää ja tarkastella syvemmin saman kirjaston aliohjelmaa. Arvioija kokee esimerkkitapausten mahdollistavan tarvittaessa kevyen siirtymän jatkokehityksen sekä testauskirjaston konfiguroinnin osalta.

Huolenaiheena hän nostaa esiin väärät hälytykset. Ilmaan jää myös kysymys siitä, olisiko

CSS-tyylien ohjelmallinen testaus varmempi lähestymistapa komponenttien ja sivujen tyylien regressiotestaukseen. Nykyisessä projektissa hyödynnetään suurimmaksi osaksi ohjelmallista integraatio-testausta, jota pidetään luotettavana. Arvioija korostaa myös, että CI/CD-putken tuomat haasteet tulee kartoittaa, jos projektissa edetään visuaalisten testien kanssa. Haasteeksi voi muodostua esimerkiksi suorituskyvyn puute.

Tyylinvalvojan osalta ei kuitenkaan toteuteta uutta iteraatioita. Tutkimus antaa jo riittävästi informaatiota kohdeorganisaatiolle jatkokehitystä varten. Esimerkkitapaukset ovat käytettävissä komponenttikirjaston testitiedostojen lähdekoodina.

5.4 Artefaktin kokonaisarvio

Kohdeorganisaation edustajat kokivat, että tutkimus toi heille lisäarvoa ja ovat pääasiassa tyytyväisiä tutkimuksen tuloksiin, joskin kehitettävääkin on. Tutkimuksen ohjaaja ja kohdeorganisaation osoittama arvioija luonnehtii tutkimuksen antia näin:

"Työkalu tarjoaa vankan pohjan käyttöliittymäkehityksen yksinkertaistamiseen, sillä se vähentää kehittäjän tarvetta määritellä tyylejä ja ominaisuuksia komponenttitasolla. Tämä auttaa pienentämään virhemarginaalia ja tehostaa kehitystyötä.

Komponenteista erityismaininnan ansaitsee lomakekomponentti, joka vakuuttaa vähäisellä konfiguraatiotarpeellaan ja tarjoaa selkeät suuntaviivat yhtenäisten lomaketoteutusten luomiseen. Komponentin lähdekoodi on helposti ymmärrettävää, yksinkertaista ylläpitää ja mahdollistaa jatkokehityksen ilman merkittävää regressioriskiä.

Huomionarvoista on kuitenkin komponenttikohtaisten "propsien"eli ominaisuuksien hyödyntäminen Chakra UI lähdesivustolta. Koska kyseessä on tekijänoikeuksien alainen sisältö, on tämä otettava huomioon, jos komponenttikirjaston Playground-sovellus julkaistaan avoimesti verkossa. Lisäksi mukana on ominaisuuksia, jotka eivät välttämättä vastaa tilaajaorganisaation tarpeita. Esimerkkinä voidaan mainita painikekomponentin väriskeemat.

Työkalu soveltuu ehdotetussa muodossaan myös tilaajan asiakaskohtaisiin käyttötapauksiin, joissa esimerkiksi painikkeiden tyylit poikkeavat oletusasetuksista. Näissä tapauksissa asiakaskohtaiset vaatimukset voidaan toteuttaa joko päivittämällä teemakonfiguraatioita tai luomalla asiakasympäristöön päätason komponentteja, jotka hyödyntävät olemassa olevia komponenttikirjaston ratkaisuja.

Visuaalinen testaus ei nykyisellään ole tuotantokelpoinen ja vaatii tilaajaorganisaatiolta lisäarviointia sen osalta, voidaanko ehdotettu lähestymistapa toteuttaa riittävän luotettavasti sekä paikallisessa kehityksessä että CI/CD-putkessa.

Sekä olemassa olevien että uusien komponenttien määrittely on tehty yksinkertaiseksi, muun muassa esittelysivujen avulla, joissa komponenttien parametrit ovat helposti tarkasteltavissa. Tämä mahdollistaa tehokkaan työskentelyn työkalun parissa.

Kaiken kaikkiaan kyseessä on työkalu, joka nykyisessä muodossaan helpottaa merkittävästi kehitystyötä. Se tarjoaa organisaatiotasoisten komponenttien lisäksi kattavan dokumentaation valmiiksi määritellyistä tuotekohtaisista peruspilareista."
- Sauli Purhonen, Senior Software Developer, Akamon Innovations Oy

Kohdeorganisaation johtaja ja tutkimusaiheen alustaja näkee työkalulla olevan myös kaupallista merkitystä tuoden tuotteisiin vaikutelman hyvästä laadusta. Hän kuvailee tutkimuksen merkitystä näin:

"Käyttöliittymät ovat merkittävä tekijä tarjoamiemme ratkaisujen käytettävyydessä, ja yhtenäinen visuaalinen ilme parantaa tuotteidemme kaupallista menestystä. Viimeistelty ja mobiililaitteille skaalautuva käyttöliittymä luo asiakkaalle ammattimaisen vaikutelman ja helpottaa tuotteiden markkinointia ja myyntiä. Tämä on erityisen tärkeää kaltaisellemme yritykselle, jossa tuotteiden modulaarisuus ja asiakasräätälöinnin tehokkuus ovat keskeisiä kilpailuetuja.

Yhtenäisten ja huolellisesti suunniteltujen käyttöliittymien kaupallinen arvo korostuu erityisesti myyntitilanteissa: ne antavat hyvän ensivaikutelman ja lisäävät asiakkaiden luottamusta. Helppokäyttöiset ja huolitellut käyttöliittymät parantavat asiakastyytyväisyyttä sekä vähentävät asiakastuen tarvetta, mikä osaltaan lisää asiakaspysyvyyttä.

Tulemme hyödyntämään tämän tutkielman lopputuotoksia osana yrityksemme ohjelmistokehitystä, erityisesti käyttöliittymäkomponenttien suunnittelussa ja hallinnassa, varmistaaksemme korkealaatuisen ja yhtenäisen lopputuloksen jatkossa."
- Juuso Kari, toimitusjohtaja, Akamon Innovations Oy

6 Yhteenveto ja pohdinta

Tässä tutkimuksessa toteutettiin kohdeorganisaation tarpeisiin graafisen käyttöliittymän tyyliopas, ohjeet komponenttikirjaston rekonstruktiolle ja malli tyylejä valvovista visuaalisista testeistä. Tutkimus vahvistaa aiempaa tietoa siitä, että käyttöliittymän graafisilla elementeillä on suuri vaikutus tuotteen laatuun ja käyttäjäkokemukseen.

Tutkimuksen viitekehyksessä käsiteltiin komponenttilähtöistä kehitystä ja komponenttien uudelleenkäyttöä, graafisia komponenttikirjastoja ja niiden merkitystä. Esiteltiin käsitteet käyttöliittymän tyyliopas, suunnittelujärjestelmä ja tyylinvalvonta. Tutkimus toteutettiin suunnittelututkimuksena sykleittäin Hevner (2007) mallia mukaillen.

Täsmällisyyssyklissä tehtiin järjestelmän ja sen keskiössä olevan komponenttikirjaston nykytilan kartoitus. Kerättiin tietoa komponenttikirjaston rakenteesta, sen käytöstä ja komponenttien keskinäisistä suhteista. Kuvattiin komponenttien vuorovaikutusta UML-diagrammein ja koottiin käyttöliittymästä sekä komponenttien lähdekoodista ulkoasuun vaikuttavia epäkohtia. Nykytilan kartoitus korosti tarvetta yhteiselle tyyliohjeistukselle.

Relevanssisyklissä määriteltiin työn tavoitteet. Artefaktin keskeisimmille osille, tyylioppaalle ja tyylinvalvojalle suoritettiin vaatimusmäärittelyt. Tyylioppaan osalta käsiteltiin yleiset asiat, kuten sijainti, rakenne ja tekninen toteutus. Määriteltiin tyyleihin ja malleihin liittyvät vaatimukset, kuten typografiaan, väreihin, välistyksiin ja sivunäkymiin liittyvät asiat. Keskeisessä osassa olivat graafisten komponenttien vaatimukset. Tyylinvalvojan osalta määriteltiin sijaintiin, toimintaan ja itse työkaluun liittyviä vaatimuksia. Kartoitettiin markkinoilla olevien testaustyökalujen kirjo. Koska suunnittelututkimus on ratkaisukeskeistä, vaatimusmäärittelyn tarkoituksena oli luoda hyvä kuva siitä, millaiselle työkalulle kohdeorganisaatiolla on tarvetta.

Työn kannalta keskeisintä, suunnittelusykliä, iteroitiin neljä kertaa. Ensimmäisessä vaiheessa toteutettiin käyttöliittymän tyyliopas osaksi komponenttikirjaston Playground-ympäristöä. Suunniteltiin työkalun rakenne ja toiminta. Luotiin organisaatiospesifit komponentit, niiden dokumentaatio ja käyttöliittymän esitysmuoto sisältäen omat näkymät myös tyylielementeille ja sivumalleille. Toteutus evaluoitiin ja sen perusteella suoritettiin toinen iteraatio, jossa jatkokehitystä vaativia työkalun osia kehitettiin edelleen. Toisen evaluoinnin perusteella todettiin tyylioppaan olevan hyväksyttävissä.

Suunnittelusyklin kolmannessa vaiheessa käsiteltiin komponenttikirjastoon tarvittavia muutok-

sia, jotta elävä tyyliopas voitaisiin ottaa käyttöön. Alkuperäisestä suunnitelmasta poiketen ja komponenttikirjaston laajuus huomioiden, päätettiin jättää varsinainen rekonstruktio tekemättä ja sen sijaan luoda ohjeet ja esimerkit kirjaston muutoksille kohti tyylioppaan käyttöönottoa. Kirjaston komponenttiesiintymät listattiin, jotta nähtäisiin työn laajuus. Käsiteltiin tyylikomponentteja yksi kerrallaan ja tehtiin kullekin esimerkinomainen muutos, jonka tulokset raportoitiin. Tyylikomponentteihin tarvittavat muutokset ja ehdotukset tuotiin raportissa myös esille.

Suunnittelusyklin viimeinen vaihe keskittyi tyylinvalvojan suunnitteluun ja kehitykseen. Valittiin testaustyökalujen joukosta kolme snapshot-tekniikalla toimivaa avoimen lähdekoodin ohjelmistoa ja suoritettiin näille pienimuotoiset esitestit tarkoitusta varten luodulla käyttöliittymällä. Vertailtiin ohjelmien toimivuutta tyylivalvojan ominaisuudessa, tarvittavan konfiguroinnin määrää, testien lähdekoodia ja testeistä muodostuvia tuloksia. Tyylinvalvojaksi valittiin Vitest image snapshots, joka konfiguroitiin komponenttikirjastoon siellä jo olevan toiminnallisen Vitest:in rinnalle. Työkalua testattiin tyylikomponenteista kootulla käyttöliittymällä, jolle suoritettiin useita erilaisia snapshot-testejä. Lopuksi analysoitiin testin tulokset ja arvioitiin työkalun soveltuvuutta tyylinvalvojaksi.

Tutkimuskysymyksiä oli 2 ja niiden lisäksi kolme apututkimuskysymystä, joita käytettiin osatavoitteina. Ensimmäinen tutkimuskysymys käsitteli tyyliopasta: **TK1: Mitä asioita tyylioppaan täytyy sisältää, jotta varmistetaan käyttöliittymän yhtenäinen ilme ja parannetaan käyttäjäkokemusta?** Tutkimuksen perusteella voidaan aivan ensin todeta, että niin kohdeorganisaation kuin muidenkin vastaavalla tavalla kaupallista tuotekehitystä tekevien yritysten tapauksessa, yhteisellä tyyliohjeistuksella on suuri merkitys. Käyttöliittymän ulkoasuun vaikuttaa merkittävästi näkymien yhtenäinen asettelu, komponenttien intuitiivinen ja keskenään samanlainen ulkoasu, tekstien ja otsakkeiden johdonmukainen käyttö sekä väriharmonia. Näin tyylioppaassa toteutetut elementit, keskeiset komponentit, typografia, värit, ikonit ja mallit ovat juuri ne tekijät, joita tyylioppaan täytyykin sisältää, jotta ilme pysyy yhtenäisenä ja harmonisena.

Toinen varsinainen tutkimuskysymys käsitteli tyylinvalvontaa: **TK2: Mitä vaihtoehtoja on olemassa valvoa komponenteista koostetun käyttöliittymän visuaalista ilmettä ja vastaavuutta tyylioppaaseen?** Käyttöliittymän visuaalisilla regressiotesteillä voidaan verkkosivuja tai yksittäisiä komponentteja vertailla keskenään. Valtavasta määrästä ohjelmistovaihtoehtoja valittiin kolme snapshot-tekniikkaan perustuvaa työkalua ja esiteltiin lisäksi yksi tekoälyyn perustuva malli. Vaihtoehtoja on tietysti muitakin, esimerkiksi CSS-koodiin perustuvat testit,

jotka toimivat paitsi itsekseen myös hyvänä lisänä snapshot-testaukselle. Tekoälyyn perustuva testaus vähentää merkittävästi manuaalista työtä, jota snapshot-testeihin ainakin aluksi liittyy. Kun opitaan ymmärtämään syyt pikselivertailun aiheuttamille väärille virheille ja määrittelemään oikeanlaiset toleranssit, voi snapshot-testauksestakin olla valtava hyöty.

Tämän tutkimuksen yhtenä haasteena oli komponenttikirjaston laajuus ja monimutkaiset keskenään vuorovaikuttavat komponentit. Tutkimussuunnitelmasta jouduttiin tämän vuoksi paikoitellen poikkeamaan. Haasteena oli myös jatkuvasti muuttuva ympäristö kirjaston ollessa käytössä ja kehittyessä koko ajan. Pelkästään nykytilan kartoituksen ja kirjaston rekonstruktion välillä oli uusia komponentteja ilmestynyt runsaasti.

Tyylien osalta pohdintaa aiheutti se tosiasia, että Playground-ympäristö on vain kehittäjille tarkoitettu työkalu. Varsinaiset asiakastoteutukset ovat lähes poikkeuksetta räätälöityjä asiakkaan oman tyylin mukaan. Tämä aiheutti hämmennystä etenkin värimaailman suunnittelussa. Tärkeämpää oli luoda värikooditus ja tekninen toteutus sille, että räätälöinti olisi jatkossa helppoa ja tehokasta. Voisi kuitenkin olla järkevää pohtia Playground:in hyödyntämistä demoympäristönä myyntitilanteissa. Tällöin kohdeorganisaation omaan brändiin perustuvalla yhtenäisellä ulkoasulla voisi olla kaupallista merkitystä.

Tyylioppaan käyttöönotto vaatii kirjaston rakenteen merkittävää uudelleenjärjestelyä ja yhden universaalin mallin yhteensovittaminen kaikkiin olemassa oleviin käyttötarkoituksiin tuntuu lähes mahdottomalta. Työ kannattaakin aloittaa uusista komponenteista ja pikkuhiljaa siirtyä rekonstruoimaan muita komponentteja. Tässä tutkimuksessa toteutettiin aihio, jota seuraamalla tavoitetilaan on mahdollista päästä.

Pohdittavaa jäi siis paljonkin ja jatkotutkimuksellekin on sijaa. Komponenttikirjaston vahva Chakra UI -riippuvuus on olemassa. Vaikuttaako hetki sitten julkaistu uusi versio 3.0 kirjaston komponenttien tai tyylioppaan esityksen toimintaan? Tutkimuksen alussa puhuttiin mahdollisuudesta rajata tämä riippuvuus organisaatiospesifeihin komponentteihin, jolloin tarvittaessa riippuvuus olisi helppo purkaa. Tämä kuitenkin vaatisi huomattavasti laajemman määrän spesifejä komponentteja, joiden rakentamiseen ei tämän tutkimuksen puitteissa ollut mahdollisuutta.

Tyylinvalvojan osalta olisi hyödyllistä tutkia tekoälypohjaisia ratkaisuja tarkemmin. Löytyisikö sieltä malli, joka suodattaisi riittävän määrän vääriä virheitä ja toimisi tehokkaasti CI/CD-putkessa? Tutkimuksessa esitellyn mallin osalta huolta aiheuttaa testien luotettavuus ja tarvittavan manuaalisen työn määrä. Työkalu antaa kuitenkin kohdeorganisaatiolle mallin, jota edel-

leen kehittämällä voidaan päästä haluttuun lopputulokseen.

Tutkimus lisäsi runsaasti uutta informaatiota siitä, kuinka olemassa olevia epäkohtia korjaamalla päästään tavoitetilaan. Se tarjoaa hyvän pohjan käyttöliittymäkehityksen yksinkertaistamiselle ja tehostamiselle kohti yhtenäistä, laadukasta ja harmonista tuoteperhettä.

Lähteet

Adebayo, Segun. 2024. “Chakra UI Documentation”. Viitattu 19. marraskuuta 2024. https://chakra-ui.com.

Applitools. 2024. “Applitools Documentation”. Viitattu 16. marraskuuta 2024. https://applitools.com/.

Battat, Michael. 2024. “What is Visual Testing?” Viitattu 16. helmikuuta 2024. https://applitools.com/blog/visual-testing/.

Coleman, Tom. 2017. “Component-Driven Development: Build UIs in a better way: from the component up”. Viitattu 12. helmikuuta 2024. https://medium.com/@tmeasday/component-driven-development-ce1109d56c8e.

ComponentDriven. n.d. “Component Driven User Interfaces”. Viitattu 12. helmikuuta 2024. https://www.componentdriven.org.

Cypress.io. 2024. “Cypress Documentation”. Viitattu 15. marraskuuta 2024. https://docs.cypress.io.

Delong, Sara. 2020. “Why You Need to Create a Data Visualization Style Guide to Tell Great Stories (Part 1)”. Viitattu 14. helmikuuta 2024. https://depictdatastudio.com/why-you-need-to-create-a-data-visualization-style-guide-to-tell-great-stories/.

Feather, Ian. 2014. “A Maintainable Style Guide”. Viitattu 15. helmikuuta 2024. https://engineering.lonelyplanet.com/2014/05/18/a-maintainable-styleguide.html.

Fessenden, Therese. 2021. “Design Systems 101”. Viitattu 13. helmikuuta 2024. https://www.nngroup.com/articles/design-systems-101/.

Foundation, GraphQL. 2024. “GraphQL Documentation”. Viitattu 19. marraskuuta 2024. https://graphql.org.

Friedman, Vitaly. 2016. “Taking The Pattern Library To The Next Level”. Viitattu 13. helmikuuta 2024. https://www.smashingmagazine.com/2016/10/taking-pattern-libraries-next-level/.

Frost, Brad. 2016. *Atomic Design.* Pittsburgh, PA. Viitattu 14. helmikuuta 2024. https://atomicdesign.bradfrost.com/table-of-contents/.

HEAVY.AI. 2024. “Graphical User Interface (GUI)”. Viitattu 18. marraskuuta 2024. https://www.heavy.ai/technical-glossary/graphical-user-interface.

Hevner, Alan R. 2007. “A Three Cycle View of Design Science Research”. *Scandinavian Journal of Information Systems* 19 (2): 87–92. Viitattu 7. helmikuuta 2024. http://aisel.aisnet.org/sjis/vol19/iss2/4.

Hevner, Alan R., Salvatore T. March, Jinsoo Park ja Sudha Ram. 2004. “Design Science in Information Systems Research”. *MIS Quarterly* 28 (1): 75–105. https://doi.org/10.2307/25148625.

HubSpot. 2024. “What is a GUI?” Viitattu 18. marraskuuta 2024. https://blog.hubspot.com/website/what-is-gui.

Kalia, Anshul ja Sumesh Sood. 2017. “Concerns in Maintaining Reusable Software Components and the Possible Solutions”. *Indian Journal of Science and Technology* 10 (23): 1–10. https://doi.org/10.17485/ijst/2017/v10i23/108703.

Kholmatova, Alla. 2017. *Design Systems.* Freiburg, Germany: Smashing Media AG. Viitattu 15. helmikuuta 2024. https://www.smashingmagazine.com/provide/eBooks/design-systems.pdf.

Lynch, Patrick J. ja Sarah Horton. 2016. *Web Style Guide, 4th Edition: Foundations of User Experience Design.* 4. painos. London, UK: Yale University Press.

March, Salvatore T. ja Gerald F. Smith. 1995. “Design and natural science research on information technology”. *Decision Support Systems* 15 (4): 251–266. https://doi.org/10.1016/0167-9236(94)00041-2.

Meta. 2024. “React Documentation”. Viitattu 19. marraskuuta 2024. https:/react.dev.

Microsoft. 2024a. “Playwright Documentation”. Viitattu 15. marraskuuta 2024. https://playwright.dev.

———. 2024b. “TypeScript Documentation”. Viitattu 19. marraskuuta 2024. https://www.typescriptlang.org.

Neville, Kat. 2010. “How To Design Style Guides For Brands And Websites”. Viitattu 14. helmikuuta 2024. https://www.smashingmagazine.com/2010/07/designing-style-guidelines-for-brands-and-websites/.

OpenJSFoundation. 2024. “Node.js Documentation”. Viitattu 19. marraskuuta 2024. https://nodejs.org.

Osmani, Addy. n.d. “Components Should Be Focused, Independent, Reusable, Small and Testable (FIRST)”. Viitattu 12. helmikuuta 2024. https://addyosmani.com/first/.

Pandey, Sakshi. 2022. “How AI in Visual Testing is transforming the Testing Landscape”. Viitattu 16. helmikuuta 2024. https://www.browserstack.com/guide/how-ai-in-visual-testing-is-evolving.

Peffers, Ken, Tuure Tuunanen, Marcus A. Rothenberger ja Samir Chatterjee. 2007. “A Design Science Research Methodology for Information Systems Research”. *Journal of Management Information Systems* 24 (3): 45–77. https://doi.org/10.2753/MIS0742-1222240302.

Qureshi, M.R.J. ja S.A. Hussain. 2008. “A reusable software component-based development process model”. *Advances in Engineering Software* 39 (2): 88–94. Viitattu 12. helmikuuta 2024. https://www.sciencedirect.com/science/article/abs/pii/S0965997807000294.

Saring, Jonathan. 2019. “A Guide to Component Driven Development (CDD): Let components drive the development of your applications.” Viitattu 12. helmikuuta 2024. https://itnext.io/a-guide-to-component-driven-development-cdd-1516f65d8b55.

Software Testing Help. 2023. “Guide To Visual Regression Testing With Visual Testing Tools”. Viitattu 15. helmikuuta 2024. https://www.softwaretestinghelp.com/visual-validation-testing/.

Storybook. 2024. “Storybook Documentation”. Viitattu 19. marraskuuta 2024. https://storybook.js.org.

UPXin. 2023. “What is a Component Library, and Why Should You Use One for UI Development?” Viitattu 13. helmikuuta 2024. https://www.uxpin.com/studio/blog/ui-component-library/.

Vale, Tassio, Ivica Crnkovic, Eduardo Santana de Almeida, Paulo Anselmo da Mota Silveira Neto, Yguaratã Cerqueira Cavalcanti ja Silvio Romero de Lemos Meira. 2016. “Twenty-eight years of component-based software engineering”. *Journal of Systems and Software* 111:128–148. https://doi.org/https://doi.org/10.1016/j.jss.2015.09.019.

Volk, Torsten. 2021. “September 2021 EMA Research Report: Disrupting the Economics of Software Testing Through AI”. Viitattu 15. helmikuuta 2024. https://ewig5qf9cgn.exactdn.com/wp-content/uploads/2021/09/EMA-Applitools-DisruptingTestingEconomics-092121.pdf.

Xu, David. 2022. “Everything you need to know about Visual Regression Testing in 2022”. Viitattu 15. helmikuuta 2024. https://david-x.medium.com/the-state-of-visual-regression-testing-in-2022-5de10ffe8f6f.

You, Evan. 2024a. “Vite Documentation”. Viitattu 19. marraskuuta 2024. https://vitejs.dev.

———. 2024b. “Vitest Documentation”. Viitattu 15. marraskuuta 2024. https://vitest.dev.