Haaga-Helia

# Suunnittelumallien ja -periaatteiden edut ohjelmointiyhteisöille käyttöliittymäkehityksessä React.js ohjelmistokirjastolla


Ilpo Jokinen

Haaga-Helia ammattikorkeakoulu



Amk-opinnäytetyö

2021

IT-Tradenomi tutkinto

## Tiivistelmä

**Tekijä(t)**
Ilpo Jokinen

**Tutkinto**
IT-Tradenomi

**Raportin/Opinnäytetyön nimi**
Suunnittelumallien ja -käytänteiden edut ohjelmointiyhteisöille käyttöliittymäkehityksessä React.js ohjelmistokirjastolla

**Sivu- ja liitesivumäärä**
34 + 6

Opinnäytetyön tarkoituksena on selvittää, mitä hyötyä suunnittelumallien ja -periaatteiden käytöstä on React.js ohjelmistokirjastolla toteutetuissa ohjelmistoissa sekä missä tilanteissa niiden implementointi aiheuttaa ohjelmointiyhteisöille tehottomia prosesseja.

Työ on laadullinen kirjallisuustutkielma, mikä sisältää kokeellisuutta konkreettisten koodiesimerkkien kautta. Lähteinä on käytetty ohjelmointikirjallisuutta, tieteellisiä julkaisuita, ohjelmistokirjastoiden virallisia dokumentaatioita sekä ohjelmoijien julkaisemia verkkoartikkeleita. Työ on laadittu keväällä 2021. Työ on rajattu käsittelemään React.js ohjelmistokirjaston suunnittelumalleja, jotka pyrkivät ratkaisemaan tilan hallintaan, informaation välitykseen, komponenttien rakenteeseen sekä ohjelmistoarkkitehtuuriin liittyviä haasteita.

Suunnittelumallit ovat dokumentoituja ratkaisuita johonkin ohjelmoinnissa useasti ilmentyvään ongelmaan. Niiden avulla voidaan taata ohjelmiston koodipohjan arkkitehtuurinen yhtenäisyys, vahvistaa koodin luettavuutta sekä optimoida sovelluksen suoritustehoa. On hyvä ymmärtää, että suunnittelumallit eivät automaattisesti estä kelvottoman arkkitehtuurin luontia.

React-sovelluksissa käytetään yleisesti Redux-kirjastoa tilan hallinnallisiin haasteisiin. Komposition avulla komponentit ovat luettavampia ja ohjelmistoarkkitehtuuri on ymmärrettävämpi sekä loogisempi. React Hooks teknologian avulla voidaan eristää tilallista logiikkaa yhtenäisiksi rajapinnoiksi, mikä vähentää tarpeetonta koodin kirjoittamista.

Ohjelmointiyhteisöiden tulisi harkita tukevatko suunnittelumallit kehitystyötä vai onko niiden implementoimisessa vaarana suunnata ohjelmiston arkkitehtuuria väärään suuntaan. Mallien hyödyntäminen pitäisi olla yhteisöiden yhteinen linjaus. Ne eivät sovi jokaiseen tilanteeseen ja voivat aiheuttaa enemmän hämmennystä kuin yhtenäisyyttä ohjelmoijien työskentelyssä.

**Asiasanat**
Suunnittelumalli, Ohjelmistoarkkitehtuuri, React.js, Komponenttihierarkia

## Sisällys


1 Johdanto ..... 1
2 Tutkimusmenetelmät ja tavoitteet ..... 3
3 Käsitteet ..... 4
4 Tietoperusta ..... 7
4.1 Mikä on suunnittelumalli ..... 7
4.2 Suunnittelumallin dokumentointi ..... 7
4.3 Antisuunnittelumallit ..... 8
4.4 SOLID-arkkitehtuuriperiaatteet ..... 9
4.4.1 S – Single Responsibility Principle ..... 9
4.4.2 O – Open-Closed Principle ..... 10
4.4.3 L – Liskov Substitution Principle ..... 10
4.4.4 I – Interface Segregation Principle ..... 11
4.4.5 D – Dependency Inversion Principle ..... 11
4.5 MVC ..... 12
4.6 Redux ..... 13
4.7 React.js ..... 15
4.7.1 Virtuaalinen DOM ja renderöintipuu ..... 16
4.7.2 Deklaratiivinen ohjelmointi ..... 16
4.7.3 Verkkosivu teknologioiden yhdistäminen ..... 17
4.7.4 JSX ..... 17
4.7.5 React Hooks ..... 19
5 Suunnittelumallien ja -periaatteiden käyttö React sovelluksessa ..... 20
5.1 Reactin suunnittelumallit ja parhaat käytänteet ..... 20
5.1.1 Tilanhallinnan ulkoistaminen Redux kirjastolle ..... 20
5.1.2 Komponentti kompositio ..... 21
5.1.3 Kustomoidut hookit ..... 23
5.1.4 Komponenttien vastuualueet ..... 25
5.2 Reactin antisuunnittelumallit ..... 26
5.2.1 Paikallisen tilan alustaminen propsien kautta ..... 26
5.2.2 Indeksin käyttäminen key-propsin arvona lista iteraatioissa ..... 27
5.2.3 Propsien levitys ..... 27
5.3 Suunnittelumallien varjopuoli ..... 28
6 Päätelmät ..... 30
Lähteet ..... 32
Liitteet ..... 35
Liite 1. Verkkosivu teknologioiden hyödyntäminen React komponentissa ..... 35

Liite 2. Propsien poraaminen ........ 36
Liite 3. Redux – Tilan hallinta ........ 37
Liite 4. JSX kompositio ja children propsi ........ 38
Liite 5. Kompositio – Komponenttien kustomointi ........ 39
Liite 6. Paikallisen tilan alustaminen propsien kautta ........ 40

# 1 Johdanto

Ohjelmoijien työ on pääpiirteisesti erilaisten ongelmien ratkomista. Laajojen ohjelmistojen rakentamisessa tulisi pyrkiä optimoimaan sovelluksen suorituskykyä ja vähentää julkaistun version latausaikoja. Asioiden uudelleen suunnittelua ja turhaa ohjelmointityötä tulisi välttää viimeiseen asti. (Gamma, Vlissides, Helm & Johnson 1994, 11–14.) Yksi tärkeimmistä ohjelmointiin liittyvistä säännöistä on estää valmiin ratkaisun uudelleen rakentaminen (MacDonald 2019).

Ihmiset tarvitsevat turvaa ja säännöllisyyttä. Ihmiset tarvitsevat käyttäytymis- ja toimintamalleja, jotka ovat todetusti toimivia ja tuottavat ratkaisun johonkin ongelmaan. He turvautuvat ongelmatilanteissa asiantuntijoihin; ihmisiin, joilla on enemmän tietotaitoa kyseisestä aihealueesta. Asiantuntijuus syntyy kokemuksesta hyödyntää tehokkaita ja toimivia ratkaisuita erilaisiin ongelmiin. (MacDonald 2019.)

Hyvänä esimerkkinä ihmisten halusta pysytellä tutuissa ja turvallisissa kaavoissa on talon rakennuttaminen. Lähestyessään putkimiehiä tai sähköasentajia ei ihmiset oletettavasti halua, että kyseiset ammattilaiset kokeilisivat jotain uusia itse kehiteltyjä metodeita putkien tai sähköjen asennuksessa. Taloja on rakennettu iät ja ajat toimivien ratkaisuiden pohjalta ja tuntuu tarpeettomalta lähteä niitä muuttamaan. (MacDonald 2019.)

Kuten talon rakentamisessa, ohjelmointityössä tulisi myös pyrkiä yhdenmukaisuuteen ja yhtenäisiin toimintamalleihin ohjelmoijien kesken. Yksi ohjelmistokehityksen haastavimmista asioista on toisen ihmisen kirjoittaman koodin ymmärtäminen ja sen jatkokehittäminen. Korjatakseen tämän ongelman ohjelmoijat ovat kehittäneet suunnittelumalleja ja -periaatteita, joiden avulla voidaan taata ohjelmiston koodipohjan arkkitehtuurinen yhtenäisyys ja vahvistaa sen luettavuutta projektin sidosryhmien sisällä. (Gamma ym. 1994, 11–14.)

Suunnittelumallit eivät kuitenkaan ole oikotie onneen ja niiden hyödyntämistä ohjelmistoprojekteissa tulisi harkita. On äärimmäisen tärkeää, että ohjelmoijat ymmärtävät sekä suunnittelumallin pääasiallisen tarkoituksen että kontekstin mihin he aikovat sitä käyttää. Modernissa ohjelmistokehityksessä mielipiteet ja käytänteet muuttuvat nopeasti ja voi olla, että suunnittelumallit ja -periaatteet voivat vanhentua tai muovautua uusiksi päivitetyiksi versioiksi. (MacDonald 2019.)

Opinnäytetyössä pyritään hahmottamaan, löytyykö React.js käyttöliittymäkehityksessä laajasti suosiota keränneitä sekä ohjelmointiyhteisöissä arvostettuja suunnittelumalleja.

Työn tarkoituksena on selvittää, millä tavalla em. kriteerien täyttävät mallit voivat tehostaa ohjelmoijien ryhmätyöskentelyä ja parantaa ohjelmiston arkkitehtuurillista rakennetta.

## 2 Tutkimusmenetelmät ja tavoitteet

Opinnäytetyössä on kaksi pääosuutta. Tietoperustan tarkoitus on antaa lukijalle yleispätevä kuva työssä esiintyvistä teemoista. Empiirisen osuuden tarkoituksena on vastata työn esittämiin tutkimuskysymyksiin:

- Millä tavoin ohjelmointiyhteisöt hyötyvät suunnittelumallien ja -periaatteiden käyttämisestä?
- Missä tilanteissa mallien käyttäminen aiheuttaa tehottomia prosesseja?
- Minkälaisilta ohjelmointirutiineilta Reactilla toteutetussa käyttöliittymäohjelmoinnissa tulisi välttyä?

Tutkimuskysymyksiin vastataan perehtymällä React ohjelmistokirjaston tilan hallintaa sekä ohjelmistoarkkitehtuuria ratkoviin suunnittelumalleihin ja -periaatteisiin sekä analysoimalla niiden hyviä sekä huonoja puolia ohjelmistokehityksessä. Opinnäytetyössä ei syvennytä JavaScriptin kieliopilliseen optimointiin tai kieleen ylipäänsä. Ajatuksena on kvalitatiivisesti tutkia luovatko ohjelmistoarkkitehtuurin yhtenäisyys ja standardisoidut käytänteet sovelluskehitystyöhön tehokkaita prosesseja, parantaako suunnittelumallien käyttäminen ohjelmointiyhteisöiden tuottavuutta pidemmällä tähtäimellä sekä miten suunnittelumallien käyttö voi positiivisesti vaikuttaa ohjelmiston suoritustehoon.

Opinnäytetyön lähdeteoksiksi on valittu muutamia suunnittelumalleihin keskittyvä ohjelmointiteoksia, ohjelmistokirjastoiden virallisia dokumentaatioita sekä lukuisia ohjelmoijien kirjoittamia verkkoartikkeleita. Opinnäytetyössä esiintyvät koodiesimerkit eivät ole lähdeperäisiä vaan kirjoittajan itse laatimia. Koodiesimerkit luovat työhön kokeellisen aspektin ja niiden tarkoitus on tukea tekstissä käsiteltyjä aiheita.

Henkilökohtaiset tavoitteet työn valmistumisen jälkeen on React.js ohjelmistokirjaston syvempi ymmärrys sekä suunnittelumallien hyötyjen ja haittojen ymmärtäminen. Tavoitteena on myös kasvattaa ammatillista kapasiteettia ja laajentaa ammattimaista termistöä, jotta viestinnälliset kyvyt kehittyisivät ja työnteko työpaikalla helpottuisi. Tarkoituksena on myös kohentaa ohjelmistojen arkkitehtuurin analysointikykyä ja helpottaa niiden heikkouksien sekä hyvien puolien tunnistamista.

## 3 Käsitteet

Opinnäytetyössä tullaan viittaamaan useisiin ohjelmistokehityksessä esiintyviin termeihin sekä lyhenteisiin, joiden merkityksen tulisi olla pääpiirteisesti selkeitä ymmärtääkseen paremmin niiden käyttöä sekä tarkoitusta kyseisessä kontekstissa. Työssä käsiteltyjen teemojen tulisi olla ymmärrettävissä myös henkilöille, joiden tietoteknillinen osaaminen on rajallisempaa kuin alan koulutuksen omaavilla.

Käsitteet ovat avattu tekijän oman kokemustaustan pohjalta eivätkä ole lainauksia ulkopuolisista lähteistä.

**Ohjelmointiyhteisö:**
Tällä termillä viitataan opinnäytetyössä ryhmään ihmisiä, jotka toimivat yhdessä ohjelmistokehityksen parissa. Ohjelmointiyhteisö voi olla minkä kokoinen tahansa ja voi toimia yrityksen sisällä tai itsenäisenä joukkona. Se voi siis olla esimerkiksi ryhmä vapaan lähdekoodin kehittäjiä tai yrityksen projektissa toimiva ohjelmointitiimi.

**Paradigma:**
Opinnäytetyön kontekstissa paradigmalla tarkoitetaan jotain tietoteknillistä toimintamallia tai -periaatetta. Paradigma on jokin yleisesti tunnettu sekä käytetty tapa toimia.

**Ohjelmistoarkkitehtuuri:**
Ohjelmiston rakenne. Ohjelmistoarkkitehtuuri käsittää ohjelmiston osien riippuvuudet, toiminnallisuudet sekä niiden tarkoitusperän. Ohjelmistojen arkkitehtuurimallit on yleensä visuaalisesti kuvattu erityyppisten kaavioiden avulla. Arkkitehtuuri toimii ohjelmistokehittäjien ohjenuorana sekä jakaa järjestelmän rakenteen loogisiin palasiin.

**Ohjelmistokirjasto:**
Tiettyyn tarkoitukseen kehitetty itsenäinen ohjelmisto. Ohjelmistokirjastot voivat olla eri kokoisia ja niitä voidaan käyttää spesifisiin tai laajoihin tarkoituksiin. Opinnäytetyön aiheena esiintyvä React.js on hyvä esimerkki laajasta ohjelmistokirjastosta, jonka avulla voidaan rakentaa kokonaisia sovelluksia.

**AJAX:**
Tulee sanoista Asynchronous JavaScript And XML. Viittaa perinteisesti tapaan missä lähetetään dataa http-protokollan avulla selaimen ja palvelimen välillä päivittämättä verkkosivua. Teknologian tarkoitus on lisätä verkkosivujen käytettävyyttä ja nopeuttaa sovelluksen toimintaa. Opinnäytetyössä käsitellään modernia ohjelmointia, missä XML-tietomuoto on korvattu uudemmalla JSON-tietomuodolla.

**Funktio:**

Ohjelmiston suoritettava yksikkö. Funktion avulla voidaan luoda ohjelmistoälyä mm. matemaattisia laskuja varten. Yleisesti funktion tarkoitus on muokata sille lähettämää dataa.

**Puhdas funktio:**

Funktionaalisen ohjelmoinnin periaatteiden mukaan luotu funktio. Puhdas funktio käsittelee sille lähetettyjä arvoja muuttumattomina, jolloin sen suorittaminen ei muuta annettuja arvoja mitenkään. Yleinen periaate on tehdä funktion sisällä kopio sille lähetetystä muuttujasta, jota voidaan manipuloida ilman, että alkuperäinen muuttuja muuttuu.

**Funktionaalinen ohjelmointi:**

Ohjelmointiparadigma. Funktionaalisessa ohjelmoinnissa pyritään tuottamaan haluttu ratkaisu rakentamalla monikerroksisia loogisia yksiköitä. Loogiset yksiköt tulee olla puhtaita funktioita, jotka eivät saa muokata globaalilla tasolla luotuja muuttujia.
Funktionaalinen ohjelmointi perustuu ajattelutapaan ratkaista arkkitehtuurillinen tavoite miettimällä mitä tulisi tehdä tavoitteen ratkaisemiseksi mieluummin kuin miten se tulisi ratkaista.

**Olio:**

Abstrakti tietotyyppi ohjelmoinnissa. Olio voi pitää sisällään yhden tai useita eri avainarvo pareja. Avain on jokin sen arvoa kuvastava nimi, minkä voi itse määrittää. Arvoksi voidaan asettaa esimerkiksi totuusarvoja, merkkijonoja, kokonaislukuja, funktioita tai toisia olioita. Oliot voivat kuvastaa ohjelmistossa mitä tahansa tietorakenteita.

**Olio-orientoitunut ohjelmointi**:

Ohjelmointiparadigma. Olio-orientoituneessa ohjelmoinnissa pyritään toteuttamaan ohjelmistoarkkitehtuuri luokkaolioiden pohjalta. Perimis- sekä riippuvuushierarkiat ovat tämän paradigman keskiössä. Tässä ohjelmointimallissa käytetään vahvempaa olio dokumentaatiota tyyppirajapintojen avulla.

**DOM:**

Dokumenttioliomalli. DOM on sovelluksen arkkitehtuurinen mallinnus ja toimii rajapintana ohjelmiston ja selaimen välillä. Sen avulla selain pystyy rakentamaan verkkosivun haluttuun muotoon.

**Komponentti:**

Osa ohjelmistoa, mikä tarjoaa sille jotain arvoa oman käyttöliittymänsä kautta. Komponentit ovat usein riippuvaisia toisistaan muodostaen yhdessä laajan riippuvuuksien verkoston ohjelmiston sisälle.

**Propsi:**

Toimii React-komponenttien informaation välityksen rajapintana. Propsi on avainarvo pari, minkä avain on jokin sana ja arvo on sanaan liitetty informaatio.

**Reducer:**

Redux-kirjastossa esiintyvä tilan hallintaan liittyvä funktio. Reducer-funktio saa tietoonsa ohjelmiston tilan tai sen osan sekä tapahtuman, minkä perusteella se manipuloi tilaa. Funktio palauttaa uuden tilan ohjelmistoon.

# 4 Tietoperusta

## 4.1 Mikä on suunnittelumalli

Erich Gamma, John Vlissides, Richard Helm ja Ralph Johnson, jotka ovat yhdessä tunnettu myös nimellä Gang of Four kirjoittivat vuonna 1994 ilmestyneen teoksen Design Patterns: Elements of Reusable Object-Oriented Software, mitä pidetään suunnittelumallien dokumentoinnin uranuurtajana. Opinnäytetyön edetessä viittaan näihin henkilöihin nimellä Gang of Four.

Suunnittelumallit ovat em. teoksen mukaisesti ohjelmistokehityksessä erinomaisiksi todettuja toimintamalleja sekä arkkitehtuurisia ratkaisuita, joiden tarkoituksena on ratkaista jokin tietyssä kontekstissa useasti toistuva ongelma (Gamma ym. 1994, 12). Opinnäytetyö tulee jatkossa tukemaan tätä määritystä suunnittelumalleista puhuessa.

Suunnittelumallien dokumentointi on tärkeää, sillä se mahdollistaa tehokkaan tavan jakaa hyviä käytänteitä eri ohjelmointiyhteisöiden välillä (Gamma ym. 1994, 388).

Suunnittelumalli voidaan ajatella olevan todistettu ratkaisu johonkin ohjelmoinnin ongelmaan, minkä voi helposti ottaa käyttöön useissa projekteissa saman haasteen ratkaisemista varten. Nämä mallit eivät kuitenkaan ole suoria ratkaisuja eivätkä korvaa sovellusarkkitehtien tai ohjelmistokehittäjien työtä. (Osmani 2012, 3.)

## 4.2 Suunnittelumallin dokumentointi

Suunnittelumallien dokumentointia pyritään toteuttamaan yhtenäisen skeeman mukaisesti. Suunnittelumallilla on oltava nimi tai määritys sen pääasiallisesta tarkoituksesta. Mallissa tulee selkeästi dokumentoida sen konteksti sekä ongelma, jonka se pyrkii ratkaisemaan. On yleistä, että mallin dokumentaatiossa ilmenee myös jokin konkreettinen skenaario, missä kyseistä mallia on käytetty. Dokumentoinnissa tulisi myös esittää, millä tavoin se voidaan implementoida ohjelmistoon ja voi sisältää siitä esimerkkejä koodin muodossa. (Gamma ym. 1994, 17–18.)

Suunnittelumallin dokumentaatiossa esitetään sen käyttöyhteydessä esiintyvät ohjelmistokomponentit sekä niiden vastuualueet. Komponenttien esittelyssä tulisi selkeästi ilmaista, millä tavoin ne toteuttavat suunnittelumallin pääasiallisen tarkoituksen. Lisäksi dokumentaatiossa tulisi tulla ilmi mallin käytön seuraukset, mitä järjestelmän osa-alueita se käyttää sekä onko se kielispesifinen eli onko malli tarkoitettu käytettäväksi jollain tietyllä ohjelmointikielellä. (Gamma ym. 1994, 17–18.)

Suunnittelumalli kertoo siis käyttäjälleen missä kontekstissa mallia tulisi käyttää, minkälaisia haasteita kontekstissa on sekä kokoonpanon, jonka tarkoituksena on ratkaista ko. ongelmat. On yleistä, että mallin arkkitehtuurinen ratkaisu on visuaalisesti esitetty diagrammeilla tai muilla kuvaajilla. Mallissa tulee esittää mahdolliset riippuvuudet, jos malli tarvitsee toimiakseen jotain toista olemassa olevaa suunnittelumallia sekä relaatiot, jos muita samantyyppisiä suunnittelumalleja on jo olemassa. Lisäksi on yleistä nähdä keskustelua mallin hyödyistä joko ohjelmointiyhteisöiden tai suunnittelumallin luojan taholta. (Osmani 2012, 9.)

### 4.3 Antisuunnittelumallit

Jotkut huonot tavat toistuvat useasti ohjelmoijien kehitystyössä ja niitä on alettu dokumentoimaan. Samalla tavalla, kun suunnittelumallien tarkoitus on dokumentoida todistetusti toimivia arkkitehtuurisia ratkaisuita, on epäsuotuisia prosesseja alettu kirjaamaan ylös, jotta ohjelmointiyhteisöt välttäisivät niitä kokonaisuudessaan. Antisuunnittelumalli koostuu yleensä johonkin ongelmatilanteeseen johtavasta viallisen käytänteen kuvauksesta sekä sen ratkaisuun tarvittavista toimenpiteistä. (Osmani 2012, 13–14.)

Ei ole yllättävää ajatella skenaariota, missä ohjelmointitiimin rosterissa tapahtuu äkillisiä muutoksia. On täysin tavallista, että IT-yrityksissä ohjelmoijat vaihtavat projekteja sidosryhmien painostuksen, resurssimuutoksien tai jopa omien henkilökohtaisten mieltymyksiensä takia. Näin ollen ei voida taata saman tiimin työskentelevän saman projektin parissa hamaan tulevaisuuteen asti. Uudet tiimin jäsenet voivat tarkoituksetta luoda ohjelmistoon "huonoa koodia". Koodin huonous mitattaisiin tässä tilanteessa sen yhteensopivuudella muuhun järjestelmään. Uusi työntekijä ei välttämättä saa ohjeistusta suunnittelumallien käytöstä vaan voi työnteollaan aiheuttaa muuten erittäin synergiseen ja yhtenäiseen järjestelmään eroavia tapoja ratkaista ongelmia, jotka voivat pitkässä juoksussa aiheuttaa haastavia ja vaikeasti jäljiteltäviä ohjelmistovirheitä. (Osmani 2012, 13–14.)

Hyväksi todettu suunnittelumallikin voi kuitenkin muuttua antisuunnittelumalliksi väärissä käsissä, väärään kontekstiin implementoidessa tai oikeaan asiayhteyteen liitettäessä, jos perustelut sen käytölle ovat vääriä. Suunnittelumallien abstrakti luonne ei aina anna järkevää lähtökohtaa ongelman ratkaisulle, jos ongelman luonne itsessään on yksinkertainen. Yli suunnitellessaan ohjelmoija voi luoda muuten tehokkaasta ratkaisumallista huonon käytänteen omaan työhönsä. Ohjelmointi työssä on hyvin tärkeää optimoida omaa ajankäyttöään, joten jokaista työvaihetta tulisi harkita mahdollisimman aikaisessa vaiheessa. (Burtch 2017.)

### 4.4 SOLID-arkkitehtuuriperiaatteet

SOLID-arkkitehtuuriperiaatteet ovat kuuluisan ohjelmoijan sekä kirjailijan Robert C. Martinin 80-luvulla kehittämä joukko suunnittelumalleja ohjelmoijille. Nämä suunnittelumallit selventävät, millä tavalla moduulitason ohjelmistoarkkitehtuuria tulisi suunnitella. Kyseiset mallit ovat pääsääntöisesti helposti ymmärrettäviä ja niitä on mahdollista toteuttaa joustavasti erilaisissa ohjelmistoissa sekä eri ohjelmointikielillä toteutetuissa järjestelmissä. SOLID-periaatteet toimivat niin olio-orientoidussa kuin funktionaalisessa ohjelmoinnissa, sillä niiden määrittelemät fundamentaaliset ohjeistukset eivät rajaudu tiettyyn ohjelmointiparadigmaan. Ne tarjoavat geneerisiä ohjenuoria kaikkiin teknologioihin, joiden rakenne koostuu toisistaan riippuvaisista loogisista moduuleista, jotka tarjoavat dataa sekä funktioita toisilleen. (Martin 2017, 72.) SOLID pitää sisällään viisi erilaista arkkitehtuuriperiaatetta moduulien rakenteelle. Nämä viisi periaatetta ovat:

- The Single Responsibility Principle
- The Open-Closed Principle
- The Liskov Substitution Principle
- The Interface Segregation Principle
- The Dependency Inversion Principle

Yhdistämällä jokaisen periaatteen ensimmäisen kirjaimen saamme sanan SOLID. Kuten aiemmin mainittu, SOLID ei ole rajattu tiettyyn ohjelmistokehityksen teknologiaan. Opinnäytetyössä tarkastellaan periaatteita nimenomaisesti React.js ohjelmointikirjaston kautta. Moduuleista voidaan Reactin kontekstissa puhua sanalla komponentti, joten työssä tullaan käyttämään tätä ilmaisua, kun periaatteita käydään läpi.

#### 4.4.1 S – Single Responsibility Principle

SOLID periaatteiden ensimmäisen suunnittelumallin määritys on, että yksittäinen komponentti tulisi ottaa vastuun vain yhdestä tehtävästä. Esimerkiksi komponentti, joka suorittaa AJAX-kutsuja, päivittää tilaa, hoitaa jotain väliohjelmiston tehtävää ja renderöi dataa käyttäjälle toimii vastoin tätä periaatetta. Erottaessa nämä tehtävät omiin komponentteihin lisäämme koodin luettavuutta sekä skaalautuvuutta huomattavasti, sillä yksittäisten komponenttien koko pienenee ja ohjelmoija pystyy uutta toiminnallisuutta rakentaessaan helposti paikantamaan sille optimaalisen komponentin ja lisäämän sille kyseisen toiminnallisuuden. Tätä voi pitää yhtenä suurista fundamenteista käyttöliittymäohjelmistojen optimoinnissa. (Mitchell 2020.)

Single Responsibility periaatteen implementointi React-sovellukseen helpottaa myös yksittäisten komponenttien testaamista, sillä niiden yksilöity tarkoitus on selkeästi määritelty ja siten loogisesti testattavissa. Komponenttien pilkkominen esityksellisiin sekä tilaa ja logiikkaa säilöviin komponentteihin ehostaa periaatteen asettamia määreitä. (Ezeokoye 2019.)

### 4.4.2 O – Open-Closed Principle

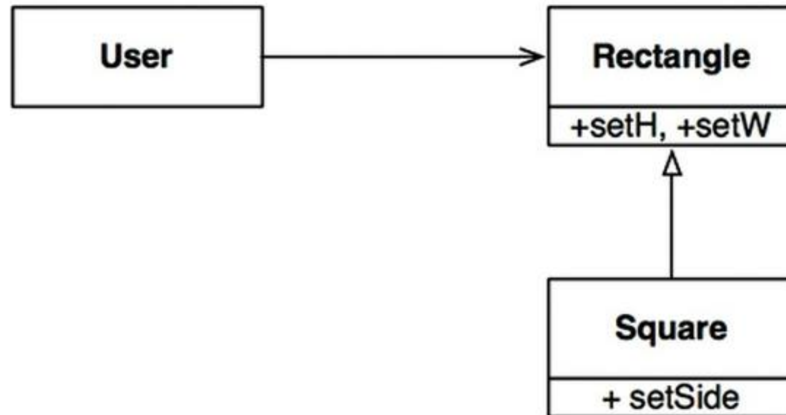
Tämän arkkitehtuuriperiaatteen on tarkoitus minimoida mahdollisten muutosten määrää ohjelmistossa silloin, kun tarve muutokseen on välttämätön. Periaatteen mukaan ohjelmiston komponenttiarkkitehtuuri tulisi suunnitella niin, että komponentit, joiden vastuulla on ohjelmiston bisneslogiikan kannalta toisarvoisempia tehtäviä, erotellaan komponenteista, joiden vastuulla on jokin järjestelmän avaintekijä. Toisin sanottuna komponentit, joiden vastuu ohjelmiston bisneslogiikan ilmaisussa on matalampi, tulisi olla riippuvaisia suuremmista tekijöistä eikä toisin päin. Näin ollen, jos järjestelmään tulee tehdä muutoksia ohjelmoijan ei tarvitse koskea millään lailla näihin erittäin oleellisiin komponentteihin, vaan tekee muutoksia riippuvuushierarkiassa alemmalla tasolla toimiviin ohjelmiston osiin. (Martin 2017, 82–87.)

Komponentit tulee siis olla riippuvaisia vain niille kriittisesti tärkeistä asioista. Toimimalla em. tavalla ohjelmoijien on helpompaa ryhtyä isompiinkin muutoksiin järjestelmän koodipohjassa ja laajentaa sovellusta turvallisemmin. (Martin 2017, 82–87.)

### 4.4.3 L – Liskov Substitution Principle

Periaatteen tarkoitus on estää alemman tason komponenttien tietotyyppien muokkaamisen siten, että tapahtuva muokkaaminen rikkoisi sen perimän yläkomponentin. Tällainen tilanne estetään oikeaoppisesti luomalla jonkunlainen sopimuspohja lapsikomponenteille lähetetyn datan tyypeistä. TypeScript on hyvä esimerkki missä periaatteen arvot nousevat vahvasti esille, sillä TypeScriptin sääntöjen mukaan joka ikinen komponentille lähetettävä propsi ja sille alustettu tyypitys tulisi olla dokumentoituna ennen käyttöä.

Hyvä periaatetta rikkova esimerkki on Martinin kirjassaan esittelemä suorakulmio- ja neliöluokan ongelma arkkitehtuurisessa suunnittelussa (Martin 2017, 90):




Kuva 1. Liskov Substitution periaatetta rikkova komponenttihierarkia (Martin 2017, 90)

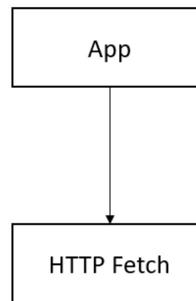
Suorakulmion ulottuvuudet ovat itsenäisesti muokattavissa, mutta neliön tapauksessa sekä korkeuden että leveyden tulisi muuttua saman verran. Neliökomponentti ei ole suunnittelumallin sääntöjen mukaan oikeaoppisesti perinnöllinen suorakulmiokomponentista, sillä se ei voi suoraan toimia sen perinnöllisen vanhemman toimintojen perusteella. Käyttäjän tulisi siis luoda jonkinlainen tarkistus siitä, onko kyseinen suorakulmio neliö vai ei. Tämän komponenttihierarkiassa ilmestyvä perinnöllinen epävarmuus on juuri se, mitä Liskov Substitution periaatteella yritetään ratkaista. (Martin 2017, 90.)

### 4.4.4 I – Interface Segregation Principle

Interface Segregation periaatteen mukaan komponenttien riippuvuussuhteet tulisi suunnitella tavalla, jossa jokaiselle komponentille tarjotaan vain ja ainoastaan niille kriittisiä propseja. Kun propseja muutetaan ylemmän tason komponentissa voi sen lapsi komponentit joutua turhaan uudelleen kääntämisen sekä uudelleen renderöinnin kohteeksi, jos niillä on käytettävissään toiminnallisuuksia, joita ne eivät tarvitse. (Martin 2017, 94–97.)

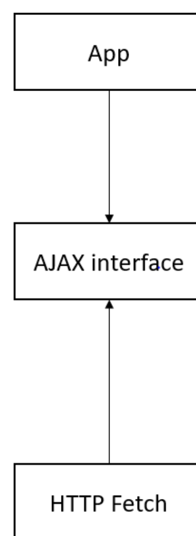
### 4.4.5 D – Dependency Inversion Principle

Viimeisen SOLID-suunnittelumallin määrityksen mukaan korkeamman tason komponentti ei tulisi olla riippuvainen matalamman tason komponenttien logiikasta. Siitä syystä riippuvuushierarkia tulisi rakentaa käänteisesti, jolloin näiden komponenttien väliin rakennetaan jokin tiettyä tarkoitusta palveleva rajapinta.




Kuva 2. Dependency Inversion periaatetta rikkova komponenttirakenne (mukaillen Gold 2018)

Kuvassa 2 esitetty komponenttirakenne rikkoo suunnittelumallia, sillä korkeamman tason App-komponentti ei tulisi olla riippuvainen AJAX-kutsuista huolehtivasta lapsi komponentistaan.




Kuva 3. Dependency Inversion periaatteen mukainen riippuvuushierarkia (mukaillen Gold 2018)

Kuvassa 3 komponenttien väliin on rakennettu AJAX-kutsuista huolehtivan komponentin tarjoamista palveluista kasattu rajapinta. Näin ollen riippuvuudet on käännetty osoittamaan yhteiseen rajapintaan ja mahdolliset muutokset verkkopyynnöistä huolehtivaan komponenttiin ei suoraan vaikuta App-komponentin käytökseen. (Gold 2018.)

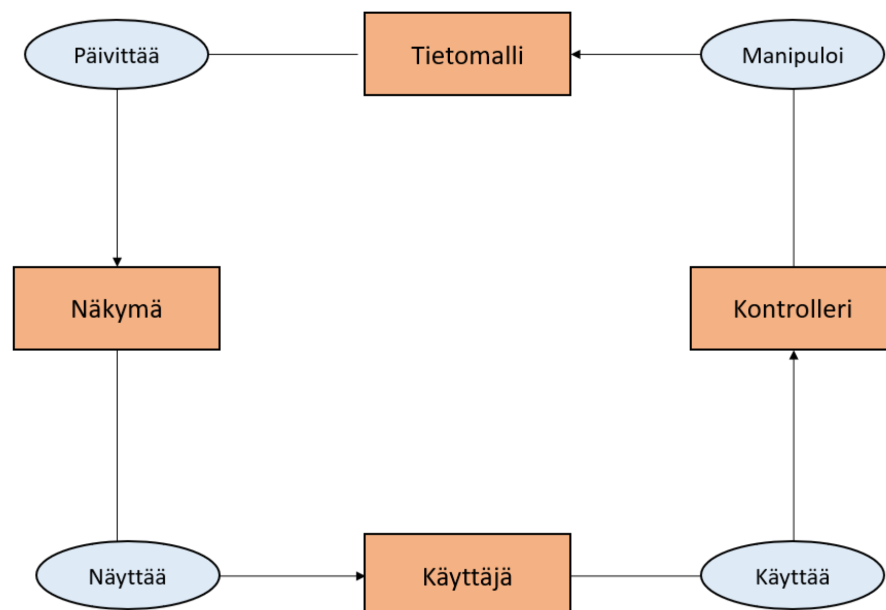
### 4.5 MVC

MVC on arkkitehtuurillinen suunnittelumalli, joka jakaa ohjelmiston kolmeen loogiseen ryhmään: tietomalli (Model), näkymä (View) sekä kontrolleri (Controller). (Tutorialspoint 2021.)

Tietomalliryhmän tarkoitus on kuvata ohjelmiston tietorakenteellista tilaa ja on vastuussa kaikesta tiedonsiirtoon liittyvästä logiikasta esimerkiksi sovelluksen tietokanta instanssin välisestä tiedon manipulaatiosta. Näkymäryhmä vastaa DOM:n rinnalla

käyttöliittymämallinnuksesta ja toimii rajapintana loppukäyttäjän ja sovelluslogiikan välillä. Ryhmän tarkoitus on esittää loppukäyttäjälle ohjelmiston tilaa käyttäen tietomalliryhmän määrittelemää tietorakennetta. Kontrolleri toimii kahden muun loogisen ryhmän kommunikointi kappaleena. Sen avulla voidaan keskustella selain- sekä palvelinlogiikan välillä. Kontrolleri kuuntelee näkymäryhmän laukaisemia tapahtumia ja välittää tiedon oikealle tietomalliryhmän edustajalle. (Tutorialspoint 2021.)

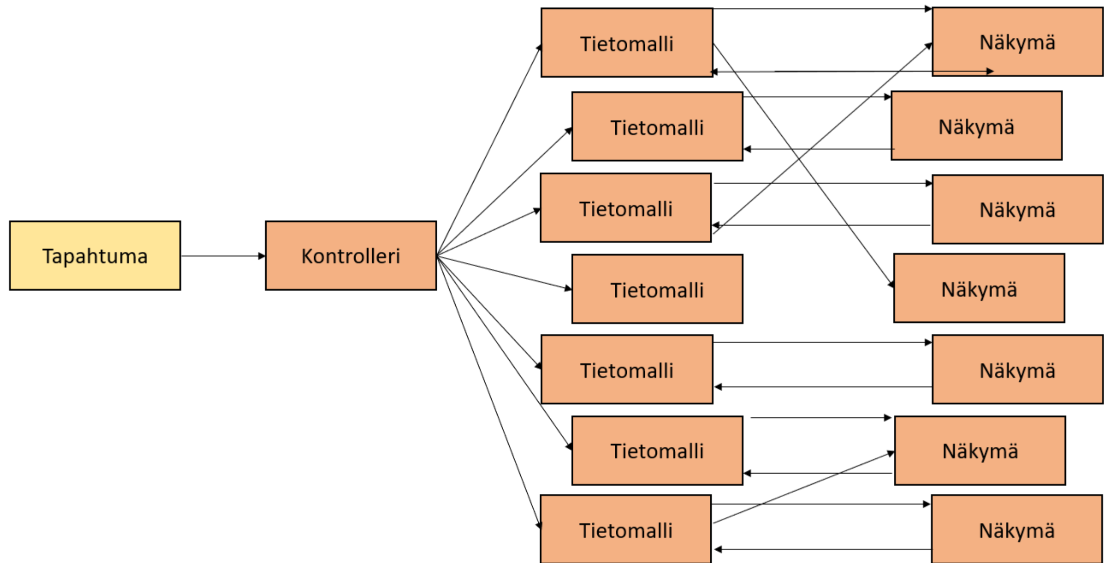
MVC tapahtumaketjussa kontrolleri saa näkymäryhmältä tietosyötteen ja lähettää tiedon eteenpäin tietomalliryhmälle. Tietomalliryhmä päivittää tietokannan ja kontrolleri lähettää päivitetyn tiedon takaisin näkymäryhmälle, mikä päivittää lopuksi DOM:n avulla käyttöliittymän loppukäyttäjälle. (Tutorialspoint 2021.)




Kuva 4. MVC-mallin tapahtumaketju (mukaillen Kumar 2018)

### 4.6 Redux

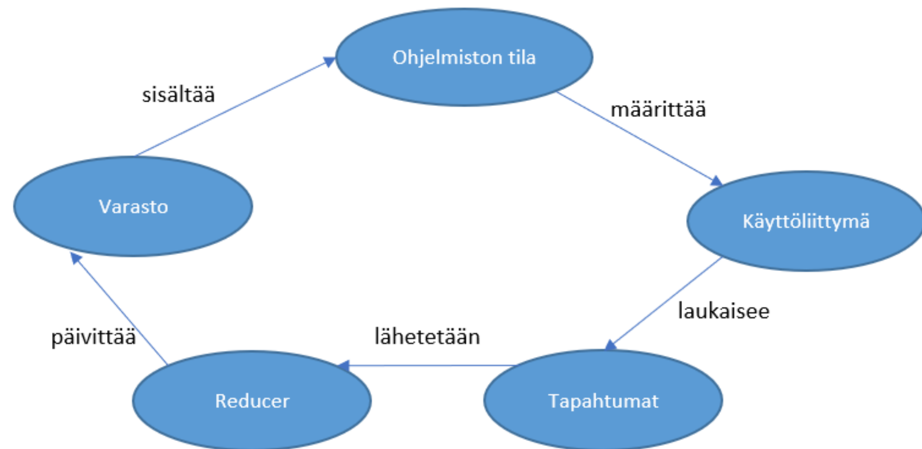
MVC mallin loogiset ryhmät toimivat kahdensuuntaisen tietovirran avulla. MVC mallin tapa lähettää tietoa edestakaisin eri loogisten ryhmien välillä alkaa tuntua huonolta ratkaisulta, kun rakennettava ohjelmisto kasvaa tiettyyn pisteeseen. Yksi käyttäjän aiheuttama toiminto ohjelmistolle voi johtaa hyvin monen tietomalli- sekä näkymäryhmän päivittämiseen ketjuttaen päivitysmääräyksiä ja aiheuttaen huonoimmassa skenaariossa loputtoman päivitysketjun. Tämä johtaa usein vaikeasti havaittaviin ohjelmistovirheisiin järjestelmässä, jos jokin menee pieleen. (Abhishek & Akshat 2015, 76–79)




Kuva 5. Tiedon siirto MVC-arkkitehtuurissa sovelluksen kasvaessa (mukaillen Abhishek & Akshat 2015, 77)

Facebook taisteli juuri näiden ongelmien parissa omissa järjestelmissään ja lähti kehittämään niihin ratkaisua. Ratkaisuksi kehittyi aivan uusi tapa välittää tietoa ohjelmiston sisällä. Flux-arkkitehtuurimalli oli syntynyt. Fluxin pääperiaatteena on eristää tilanhallinta kokonaan sovelluksen komponenttien ulkopuolelle omalle vastuulleen. Flux perustuu yksisuuntaiseen tiedonsiirtoon, mikä estää kuvassa 5 esiintyvän ongelman kehittymistä. Fluxin avulla jokainen tilan muutos ja sitä seuraava tapahtumien ketju on helposti ennustettavaa ja jäljitettävissä. Tilan hallinnassa ilmestyvät virheet ovat myös tehokkaammin paikannettavissa yksisuuntaisen tiedonsiirron ansiosta. (Abhishek & Akshat 2015, 76–79)

Vuonna 2015 kehittäjät Dan Abramov ja Andrew Clark kehittivät Flux arkkitehtuurimallia soveltavan ohjelmistokirjaston nimeltä Redux. Reduxin ideana on toimia Flux arkkitehtuurimallin mukaisesti ulkoistamalla tilan hallinnan komponenteilta erilliseen varastoon. Varasto huolehtii nimensä mukaisesti sovelluksen tilan varastoimisesta. Sovelluksen käyttäjä voi manipuloida ohjelmiston tilaa varastossa tietyin avainsanoin, mitä kutsutaan tapahtumiksi. Tapahtuma on olio, mikä pitää sisällään tiedon asiasta, mitä halutaan muokata ja millä tavalla se tulisi päivittää. Reduxissa kirjoitetaan funktioita, joiden tarkoitus on ottaa vastaan tapahtumia. Näitä funktioita kutsutaan reducereiksi. (Mbanugo 2018.) Reducer-funktiot ovat puhtaita funktioita ja helposti testattavia (Nwamba 2019). Reducer-funktioilla on tieto varaston tilan nykytilasta ja tulevasta muutoksesta. Varasto päivitetään funktion sisällä sille lähetetyn tapahtuman perusteella. Tila itsessään määrittelee käyttöliittymälle lähetetyn datan ulkoasun. (Mbanugo 2018.)

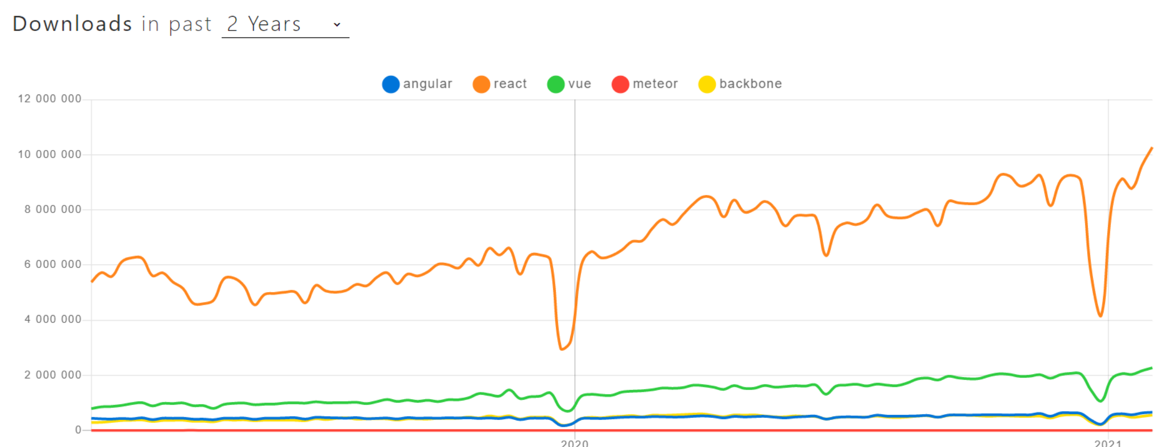



Kuva 6. Tiedon siirto Redux-arkkitehtuurissa (mukaillen Levkovsky 2017)

## 4.7 React.js

Facebookin vuonna 2013 lanseeraama ohjelmistokirjasto React.js on ollut osana viime vuosikymmenen modernien ohjelmistokehyksien renessanssissa, jonka Googlen kehittämä Angular.js aloitti vuonna 2010. Tutkielmassa viitataan React.js ohjelmistokirjastoon lyhenteellä React.

Vuosien varrella React on mullistanut JavaScriptillä toteutettua käyttöliittymäohjelmointia. Se on saavuttanut suuren suosion kehitysyhteisöissä ja sitä on Npm Trends sivuston mukaan ladattu viimeisen parin vuoden aikana enemmän kuin muita suosituimpia käyttöliittymäkirjastoja yhteensä (Npm Trends 2021). Reactia ylläpitää Facebookin kehitystiimi sekä lukemattomat henkilöt JavaScript-yhteisöissä (Bertoli 2017, 20).




Kuva 7. Ladatuimmat NPM käyttöliittymäkirjastomoduulit 2019–2021 (Npm Trends 2021)

React on komponenttipohjainen käyttöliittymäohjelmointiin suunnattu ohjelmistokirjasto, joka noudattaa deklaratiivisen ohjelmoinnin sääntöjä. Se ohjaa ohjelmoijia ajattelemaan

koodin ylläpidettävyyttä sekä komponenttien uudelleenkäytettävyyttä. (Mukhiya & Hung 2018 ,7.)

Reactin avulla voidaan luoda helposti skaalautuvia ohjelmia, jotka ovat yleisesti ottaen nopeampia kuin Angularilla toteutetut vastineet. React on myös tunnettu sen suorituskyvystään ja kompetenssista toteuttaa korkean kapasiteetin sovelluksia. Reactilla toteutetuissa ohjelmistoissa voidaan tarvita enemmän ulkoisia kirjastoita kuin esimerkiksi Angularilla, sillä se käyttää MVC-arkkitehtuurimallista hyväkseen lähinnä näkymäryhmän periaatteita. (Borrelli 2021.)

### 4.7.1 Virtuaalinen DOM

DOM:n päivitys sujuu nopeasti sen puumaisen rakenteen vuoksi, mutta sovelluksen ja komponenttien määrän kasvaessa käyttöliittymän uudelleen renderöiminen voi ruveta hidastumaan. React käyttääkin apunaan virtuaalista versiota DOM:sta. React laskee virtuaalisen DOM:n kautta optimaalisen renderöinnin virran aina, kun komponentteja luodaan tai manipuloidaan. Optimaalisen renderöintitavan löydettyään React päivittää oikean DOM:n sen perusteella. (Hamedani 2018.)

React ei kuitenkaan päivitä aitoa DOM:a kokonaan, vaan tarkistaa ennen renderöintiä nykyisen virtuaalisen DOM:n tilan ja vertaa sitä päivitystä edeltävään tilaan. Tämä tapahtuu Reactin diff-algoritmin avulla. Kun muutos on paikallistettu React osaa arvioida, mitkä komponentit on pakko virkistää ja päivittää ainoastaan ne oikeaan DOM:iin. Päivittäminen tapahtuu Reactiin sisäänrakennetun render-funktion avulla. Tämä on yksi suuri tekijä minkä vuoksi React on tehokas näkymäryhmän ohjelmistokirjasto. (Hamedani 2018.)

### 4.7.2 Deklaratiivinen ohjelmointi

React kehottaa ohjelmoijia kirjoittamaan logiikkaa deklaratiivisesti sekä kompressoidusti imperatiivisen paradigman sijaan. Deklaratiivisessa paradigmassa ajatuksena on luoda abstraktimpia, luettavampia sekä tiiviimpiä funktioita, minkä tarkoituksena on lisätä laajojen ohjelmistojen huollettavuutta. Deklaratiivisessa ohjelmoinnissa siis keskitytään siihen mitä halutaan saavuttaa, kun taas imperatiivinen ajattelu pyrkii kuvaamaan mitä askeleita haluttuun lopputulokseen vaaditaan. Deklaratiivinen ohjelmointi peittää laskennallisen komputaation ohjelmoijalta ja keskittyy luettavuuden parantamiseen. Deklaratiivista lähestymistapaa hyödyntävää ohjelmistoa on helpompi ylläpitää ja se yleisesti ottaen sisältää vähemmän ohjelmistovirheitä. (Bertoli 2017, 8–10.)

React suorittaa kaikki käyttöliittymän renderöintiin vaadittavat tapahtumat deklaratiivisesti, eli kirjastolle ei tarvitse selittää, miten se suorittaa virtuaalisen DOM:n rakennuksen. Tästä

syystä ohjelmoijan ei tarvitse tehdä muuta kuin määrittää mitä hän haluaa käyttöliittymässä nähdä. (Bertoli 2017, 8–10.) Deklaratiivinen paradigma on yksi funktionaalisen ohjelmoinnin fundamenteista (Bertoli 2017, 45).

Esimerkkinä täysin saman logiikan implementointi funktioon ensin imperatiivisesti ja sitten deklaratiivisesti:

```
const findGiraffe = animals => {
  for (let i = 0; i < animals.length; i++) {
    if (animals[i] === 'Giraffe') {
      return console.log('Giraffe was found')
    }
  }
  return
}

const findGiraffe = animals => animals.forEach(animal => animal === 'Gi-
raffe' && console.log('Giraffe was found'))
```

#### 4.7.3 Verkkosivu teknologioiden yhdistäminen

Verkkosivujen rakenne koostuu kolmesta teknologiaryhmästä. Sivujen interaktiivisuus ja looginen osuus hoidetaan JavaScriptin avulla. Rakenne luodaan HTML-kielellä ja sivun ulkoasu CSS-kielellä. Ennen vanhaan nämä kolme ryhmää refaktoroitiin erillisiin tiedostoihinsa, mutta React haluaa kovasti eroon tästä tavasta. Kun toisistaan riippuvainen informaatio on aseteltu samaan tiedostoon, on epätodennäköisempää, että vaikeasti havaittavia ohjelmistovirheitä ilmenisi, kun muuttaa jotain komponentin ulkoasuun tai toiminnallisuuteen liittyvää koodia. (Bertoli 2017, 11–14.)

Opinnäytetyön liitteen 1 Togglable-komponentilla on nappi, joka huolehtii onClick-kuuntelijan avulla komponentin lapsien renderöintiin liittyvästä logiikasta. Nappia painamalla toinen nappi muutetaan dynaamisesti näkymättömäksi manipuloimalla sen tyyliattribuuttia. Myös komponentin rakenteellinen muoto on ilmaistu sen palauttamassa HTML-koodissa. Näin ollen komponentti käyttää samassa tiedostossa hyödykseen jokaista teknologiaryhmän osaa. Komponentin koodi on siitä huolimatta selkeä ja kestävä.

#### 4.7.4 JSX

React renderöi komponenttejaan JSX-teknologian avulla. JSX on JavaScriptin syntaksinen laajennus (Introducing JSX s.a.). Selaimet eivät kuitenkaan ymmärrä JSX-syntaksia tai miten esittää sitä sivulla, joten Reactia käyttävän ohjelmoijan tulee hyödyntää Babel-kääntäjää, joka kääntää JSX-koodin selaimia tukevaksi ECMAScript 5:ksi. ECMAScript 5 on JavaScriptin vanhempi versio mitä selaimet tukevat (Bertoli 2017, 21).

Esimerkiksi seuraava JSX:llä kirjoitettu React-komponentti:

```
<Button color="blue" shadowSize={2}>
  Click Me
</Button>
```

käännetään Babelin avulla ES5-muotoon:

```
React.createElement(
  Button,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

Tämä on hyvä ymmärtää, kun kehitetään verkkosivuja Reactilla. Huomion arvoista on myös, että esimerkissä esiintyvä nappi komponentti ei ole vastike HTML-elementille <button>, vaan isolla alkukirjaimella viitataan aina React-komponentteihin. (JSX in Depth s.a.)

JSX käyttää hyödykseen koko JavaScript-kielen voimaa luoden ohjelmoijille mahdollisuuden käyttää sen täysimittaista kapasiteettia renderöitävien komponenttien sisällä. JavaScript-logiikka tulee sijoittaa JSX-syntaksin mukaan aaltosulkeiden sisälle. Aaltosulkeita voidaan käyttää hyväksi esimerkiksi, jos halutaan renderöidä muuttujia JSX:n avulla verkkosivulle tai suorittaa laskennallisia toimintoja suoraan komponentin paluuarvossa. (JSX in Depth s.a.)

```
const SwimlaneList = ({ tasksInOrder, showAll, boardId }) => {
    const classes = swimlaneStyles()

    return (
        <Grid container direction="column" classes={{ root: classes.swimlaneL-
istComponent }}>
            {tasksInOrder.map((task, index) => <Grid item key={task.id} in-
dex={index}><Swimlane tasksInOrder={tasksInOrder} task={task} index={in-
dex} showAll={showAll} boardId={boardId} /></Grid>)}
        </Grid>
    )
}
export default SwimlaneList
```

Esimerkissä nähdään, että komponentti suorittaa JavaScriptin map-listametodin suoraan JSX-kaaviossa. JSX:n avulla voidaan myös renderöidä ehdollisesti jotain komponentin osia tai kokonaisia komponentteja, kunhan ehdon varmistava logiikka on asetettu kaarisulkujen sisään. JSX:n avulla ohjelmoijat voivat luoda joustavasti monimutkaistakin logiikkaa sisältäviä komponentteja selaimen käyttöön. (JSX in Depth s.a.)

### 4.7.5 React Hooks

Reactin versio 16.8 esitteli ohjelmointiyhteisöille uuden teknologian hyödyntää Reactin sisäisiä ominaisuuksia. Tämän teknologian nimi on React Hooks. Hookkien tarkoituksena on palvella React-sovelluksia eri asiayhteyksiin tarkoitetuilla rajapinnoilla sekä vahvistaa funktionaalisen ohjelmointiparadigman periaatteita sovelluskehityksessä, sillä perinteisiä luokkakomponentteja ei niiden avulla enää tarvita. Suurin osa React 16.8 oletuksena kehitetyistä hookeista pyrkivät ratkaisemaan tilan hallintaan liittyviä haasteita. (Introducing Hooks s.a.)

Hookien käyttämiseen liittyy kuitenkin runsaasti sääntöjä. Niitä ei esimerkiksi voida käyttää ehdollisissa argumenteissa tai React-komponenttien ulkopuolella, mikä voi johtaa erinäisiin ongelmiin sovelluksen tiedostorakenteen suunnittelussa. Sääntöjen opettelu voi itsessään vaikuttaa negatiivisesti yhteisöiden migraatioon luokkakomponenteista modernien hookien käyttäjiksi. (Introducing Hooks s.a.)

# 5 Suunnittelumallien ja -periaatteiden käyttö React sovelluksessa

Tutkielman empiirisessä osiossa aiheena on selvittää millä tavoin ohjelmointiyhteisöt hyötyvät suunnittelumallien ja -periaatteiden käyttämisestä. Empiriassa käsitellyt suunnittelumallit ovat tunnettuja ja pitkään ohjelmointiyhteisöiden käytössä olleita. On olemassa myös asiayhteyksiä, joissa mallien hyödyntämisestä voi koitua enemmän haittaa kuin hyötyä, joten tässä osuudessa esitetään myös konteksteja missä niiden käyttö voi aiheuttaa tehottomia prosesseja.

Yleisimmät Reactin suunnittelumallit yrittävät ratkaista renderöintiin sekä informaation välitykseen liittyviä haasteita. Reactin ympärille luodut suunnittelumallit käsittävät lisäksi menetelmiä luoda komponentteja rakenteellisesti sekä hierarkkisesti optimaalisella tavalla.

Empirian tavoitteena on kiteyttää millä tavalla voidaan toteuttaa tukevia sekä skaalautuvia React-sovelluksia, joiden komponenttiarkkitehtuuri sekä tilan hallinta on kuvattu ja toteutettu helppolukuisesti sekä ymmärrettävästi. Osuudessa käydään myös läpi, minkä tyyppisistä ohjelmointirutiineista Reactilla toteutetussa käyttöliittymäohjelmoinnissa tulisi välttyä.

## 5.1 Reactin suunnittelumallit ja parhaat käytänteet

React-pohjaiset ohjelmistot ovat selainpohjaisia käyttöliittymäsovelluksia. Ohjelmistot käännetään ajoympäristössä kompressoituun muotoon ja niiden pakettikoko sekä koodilogiikan kompleksisuus vaikuttavat huomattavasti sovelluksien latausaikoihin, suoritustehoon sekä loppukäyttäjien kokemukseen. Näin ollen React-pohjaisessa kehityksessä tulisi kiinnittää huomiota ohjelmiston optimointiin, jotta em. asiat paranisivat.

### 5.1.1 Tilanhallinnan ulkoistaminen Redux kirjastolle

React-sovellukset toimivat komponenttihierarkkisessa arkkitehtuurissa, missä tieto liikkuu ylhäältä alaspäin. Korkeamman tason komponentit lähettävät hierarkiassa alemman tason komponenteille niille kriittisiä toiminnallisuuksia ja dataa. Useasti ohjelmistojen kehityksessä törmätään ongelmaan missä alemman tason komponenttien tulisi kommunikoida suoraan toisilleen tai arvoasteikolla ylemmällä tasolla oleville sovelluksen osille. Ilman Reduxia ja sen arkkitehtuurista suunnittelumallia tällaiset tiedonsiirto väylät olisivat vaikeasti toteutettavissa ja saattaisivat johtaa koodin luettavuuden heikentymiseen ja työläästi paikannettaviin ohjelmistovirheisiin. (Why Use React Redux? s.a.)

Reduxin tarkoituksena on estää ns. propsien poraaminen (Liite 2), missä tietty propsi joudutaan turhaan välittämään porrasmaisesti usean hierarkiatason kautta lopulliseen

päämääräänsä. Propsin muuttuessa uusi tila päivittyy kaikissa komponenteissa, joiden läpi se on lähetetty aiheuttaen turhaan usean komponentin uudelleen renderöimisen. Turhat uudelleen renderöinnit laskevat sovelluksen suoritustehoa. Kyseinen tapa suunnitella tiedonsiirtoa rikkoo mm. tutkielmassa esitettyä Interface Segregation suunnittelumallia, sillä komponenteilla on turhaan riippuvuuksia ohjelmiston tilan osiin, mistä niillä ei ole minkäänlaista vastuuta. (Redux FAQ: React Redux s.a.)

Reduxin tarjoama suunnittelumalli tiedonsiirrolle tarjoaa ohjelmointiyhteisöille järkevän ja yhtenäisen tavan mikä tottelee SOLID-periaatteita. Reduxissa ohjelmiston korkeimman tason komponentti kääritään Provider-komponenttiin, mikä tarjoaa propsina tilan säilöntää vastaavan varaston. Tällöin mitä tahansa tilan osaa voidaan manipuloida missä tahansa hierarkkisessa tasossa välttäen tarpeettoman tiedonsiirron ohjelmiston sisällä sekä estäen turhien riippuvuussuhteiden syntymistä. (Provider s.a.) Yksi tapa tarttua haluttuun tilaan on kutsua react-redux-kirjaston funktiota useSelector, minkä avulla voidaan asettaa tavoiteltu tila muuttujaan, jotta sitä voidaan hyödyntää komponentin sisällä (Liite 3).

Redux-kirjastolla on kuitenkin huonojakin puolia. Redux-varaston elinkaari ei vastaa komponenttien elinkaarta. Ohjelmoijien täytyy siis manuaalisesti pitää huolta siitä, että varaston tila vastaa sovelluksen tilaa puhdistamalla sieltä tiettyjä avainarvo pareja sitä mukaa, kun loppukäyttäjä siirtyy näkymistä toisiin tai siirtyy muuten käyttäjäpolussaan eteenpäin. (Yosef 2020.)

Redux-kirjastolla on suhteellisen haastava oppimiskäyrä. Ohjelmoijat, jotka perehtyvät kirjastoon joutuvat tuntemattomille vesille, jossa on laaja joukko uusia termejä. Reduxin opettelu vie oman aikansa, mikä on hyvä tiedostaa, jos kirjastoa halutaan alkaa käyttämään. Lisäksi Reduxin arkkitehtuurinen malli sekä sen dokumentaation mukainen tiedostorakenne voi tuntua kokemattomalle kompleksilta ja vaikeasti lähestyttävältä. (Yosef 2020.)

Yosefin (2020) mukaan ohjelmoijien tulisi suunnata kohti Reactin sisäänrakennettuja tilanhallintaratkaisua nimeltä Context. Context on kuin Redux, mutta se käyttää Reactin hookkeja tiedonsiirtoa varten, mikä tuntuu intuitiiviselta vaihtoehdolta, sillä silloin ylimääräisiä kirjastoita ei tarvitse ladata tai opiskella.

### 5.1.2 Komponentti kompositio

Ohjelmointi on pohjimmiltaan prosessi, missä yritetään ratkaista jokin ongelma rikkomalla se pienempiin ongelmiin ja kehittämällä niihin ratkaisut. Yksittäisen ongelman ratkaistuaan toteutettu tulos lisätään osaksi isompaa kokonaisuutta. Tämä toistuu niin kauan, kunnes

alkuperäinen ongelma on täysin ratkaistu. React-kehityksessä toteutuneet tulokset voivat olla React-komponentteja, jotka yhdessä koostavat sovelluksen käyttöliittymän. Kompositiolla tarkoitetaan ohjelmiston kokoamista pienistä palasista itsenäiseksi kokonaisuudeksi. (Lobera 2019.)

Yksi arkkitehtuurillinen lähestymistapa tilanhallinnan toteuttamiselle React-sovelluksissa on käyttää hyödyksi komponentti kompositiota (Composition vs Inheritance s.a.). Liitteessä 4 komponentti Dashboard saa JSX-tagin sisään sille rakennetut lapsikomponentit näin ollen toimien natiivin HTML-kielen tavoin. Komponentti pääsee kyseisiin lapsiin käsiksi erityisellä propsilla nimeltä children. Kyseisessä asiayhteydessä vältytään user-propsin poraamiselta komponenttien läpi WelcomeMessage-komponentille, mikä vastaa käyttäjän tervehdyksen renderöinnistä. Children-propsi luo deklaratiivisuutta ja helppo luettavuutta komponentin koodipohjaan (Lobera 2019).

On kuitenkin tärkeää huomioida, että laajemmissa sovelluksissa ei ole järkevää käyttää JSX-kompositiota suoraan App-tasolla eli sovelluksen arvojärjestyksen ylimmällä portaalla ainakaan jokaisen komponentin osalta, sillä silloin App-komponentti paisuisi todella isoksi. (Composition vs Inheritance s.a.)

Kompositiolla on myös muita käyttötarkoituksia kuin tilanhallintaan liittyvät ratkaisut. Geneeristen komponenttien lapsia ei välttämättä tiedetä etukäteen. Komponentit kuten Button tai Heading ovat hyviä esimerkkejä children-propsin käyttäjiksi, sillä niiden sisältämä teksti voi vaihdella kontekstista riippuen. (Composition vs Inheritance s.a.)

Liitteessä 5 on kuvattu yleistä Button-komponentin käyttöä komposition avulla. Sen pohjalta on luotu spesifimpi DeleteButton-komponentti, missä määritetään Button-komponentin väri sekä sen sisältämä teksti. Tällä tavalla ohjelmoijat pystyvät luomaan geneeristen komponenttien pohjalta tarkempiin tarkoituksiin soveltuvia kustomoituja komponentteja. Tämän tyyppinen komponenttien uudelleenkäytettävyys on React-sovelluskehityksen tärkeimpiä arkkitehtuurillisia pilareita. (Bhargav 2017.)

Komponentti kompositiota käyttämällä rakennetut ohjelmistot ovat helpommin laajennettavissa ja niiden rakenne on selkeämpi. Yksittäisten komponenttien testaaminen sekä ohjelmistovirheiden paikantaminen on myös sujuvampaa. Kompositio muistuttaa olio-orientoidussa ohjelmoinnissa esiintyvää perimishierarkiaan pohjautuvaa logiikkaa. Voisi ajatella, että liitteen 5 DeleteButton-komponentti perii geneerisemmän Button-komponentin toiminnot. Reactin dokumentaatiossa kehotetaan kuitenkin unohtamaan perinnälliset suhteet ja käyttämään kompositiota suunnitellessaan ohjelmiston arkkitehtuuria. (Composition vs Inheritance s.a.)

### 5.1.3 Kustomoidut hookit

Reactin oletusarvoiset hookit ovat yleishyödyllisiä, mutta Reactin dokumentaatio kehottaa ohjelmoijia luomaan omia kustomoituja hook-funktioita spesifimpiin tarkoituksiin. Kustomoitu hook on toistuvaa tapahtumaa varten kirjoitettu funktio, mikä käyttää Reactin omia hook-funktiota tai muita kustomoituja hook-funktioita hyödykseen. Lisäksi kustomoitu hook-funktio ulkoistaa useasti käytettyä logiikkaa komponenttien ulkopuolelle. Oma tekoisten hookkien nimet tulisi alkaa sanalla use, jotta muut sovellusta kehittävät ohjelmoijat ymmärtävät, että kyseessä on jokin hookkeja hyödyntävä rajapinta mikä noudattaa React hook sääntöjä. (Building Your Own Hooks. s.a.)

Eristetty logiikka lisää skaalautuvuutta ohjelmistossa, sillä hookki on helppo ottaa käyttöön missä tahansa komponentissa importoimalla se komponentin alustuksen yhteydessä. Näin ollen ohjelmoijat pystyvät uudelleenkäyttämään tilallista logiikkaa eri puolilla ohjelmistoa muuttamatta sen arkkitehtuurista mallia. Sovelluksen rakenne on myös lähestyttävämpi, kun kustomoidut hookit on refaktoroitu omiin tiedostoihinsa. Hook-funktioissa tapahtuvaa logiikkaa voidaan käyttää joustavasti sovelluksen eri osissa vähentäen sen koodirivien kokonaismäärää. (Building Your Own Hooks. s.a.)

Kustomoitujen hookkien implementointi ohjelmistoihin tukee vahvasti SOLID-periaatteita. Ne piilottavat kompleksista logiikkaa yksinkertaisen käyttöliittymän taakse niin kuin Interface Segregation periaatteessa on määritelty. Myös Dependency Inversion periaate toteutuu, sillä hookit luo hierarkiatasojen väliin yhteisen rajapinnan, josta sekä matalamman että korkeamman tason komponentit ovat riippuvaisia. Lisäksi hookkien käyttö noudattaa Single Responsibility periaatetta, koska toistuvaa logiikkaa ei pääse syntymään komponenttien sisälle. Hookit ovat vastuussa vain ja ainoastaan niille asetetuista tehtävistä.

```
const useAddTask = () => {
    const returnValue = useMutation(ADD_TASK, {
        update: async (cache, response) => {
            addNewTask(response.data.addTaskForColumn)
        },
    })
    return returnValue
}
```

Esimerkin kustomoitu hook-funktio useAddTask päivittää apollo-client-kirjaston hookilla uuden tehtävän tietokantaan sekä päivittää selaimen välimuistin. Kyseinen kustomoitu hook-funktio voidaan importoida mihin tahansa sitä tarvitsevaan komponenttiin, jos sovelluksessa halutaan luoda uusi tehtävä useissa eri tilanteissa.

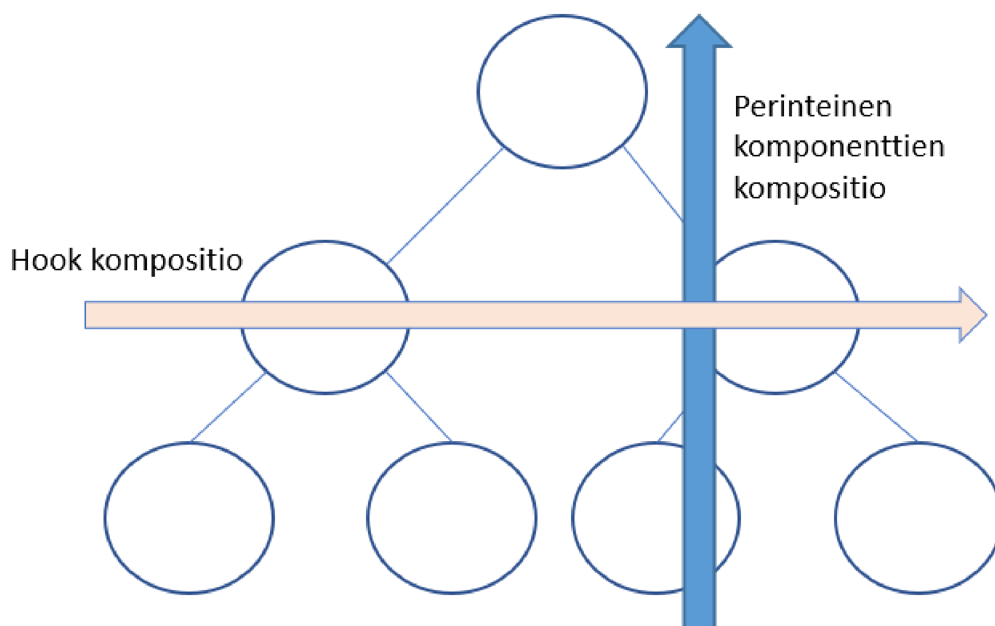
Hookkien hyödyt verrattuna muihin kompositiomalleihin, kuten korkeamman tason komponentteihin tai renderöinti propseihin (Lobera 2019):

- Ne eivät luo uusia komponentteja Reactin renderöintipuuhun parantaen renderöintiprosessia ja ohjelmistoarkkitehtuurin luettavuutta
- Niiden avulla komponentti voidaan rikkoa pienemmiksi funktioiksi mitkä ovat jaeltavissa ympäri sovellusta
- Niiden käyttö vähentää sovelluskehittäjien resursseja pitkällä tähtäimellä

Perinteisesti komponenttien kompositio tapahtuu alatasolta ylöspäin. Alemmassa esimerkissä korkeamman tason komponentin luomisessa kompositio tapahtuu alhaalta ylöspäin niin, että komponentille tarjoillaan Reactin reititinominaisuus (Lobera 2019):

```
export default withRouter(Component)
```

Hookkien avulla komponenttilogiikkaa voidaan jakaa Reactin renderöintipuun näkökulmasta sivuttain komponenteilta toisille (Kuva 8). Komponenttilogiikalla tarkoitetaan logiikkaa, mikä käsittelee komponenttien elinkaareen tai tilaan liittyviä toimintoja. Tämän kompositiomallin avulla Reactin komponentit voivat jakaa komponenttilogiikkaa helposti yhtenäisen rajapinnan kautta vähentäen tarpeettomia riippuvuussuhteita ohjelmistossa kuten SOLID-periaatteissa on määritelty. (Lobera 2019.)




Kuva 8. React sovelluskehityksessä esiintyvät kompositiomallit (mukaillen Lobera 2019)

Hookit ovat modernein tapa toteuttaa komponentti kompositiota ja mm. Reduxin kehittäjän Dan Abramovin mukaan paras tapa toteuttaa sitä. Ainoana miinuksena hookeille voisi todeta, että ohjelmoijat, jotka ovat tottuneet luokkakomponentteihin ja toteuttaneet vuosia React-sovelluksia niiden pohjalta joutuvat hetken aikaa opiskella hook-teknologiaa ennen kuin voivat käyttää sitä tehokkaasti.

### 5.1.4 Komponenttien vastuualueet

Reduxin kehittäjä Dan Abramov on luonut tunnetun komponentteja komponenttilogiikkaa säilöviin ja ohjelmiston tilaa esittäviin vastuualueisiin jaottelevan suunnittelumallin. Malliin on dokumentoituna nyrkkisäännöt komponenttien roolien tunnistamiselle. Tiivistettynä säiliökomponentti on tilaa ja tilallista logiikkaa hallitseva, kun taas esityksellinen komponentti vastaa nimensä mukaisesti ainoastaan sovelluksen tilan esittämisestä loppukäyttäjälle. (Abramov 2019.)

Käyttäjä toki muokkaa sovelluksen tilaa esityksellisten komponenttien kautta, mutta todellisuudessa käyttäjän aiheuttamat toiminnot aktivoivat esitykselliseen komponenttiin propsina tarjotun tapahtumankäsittelijän. Tapahtumankäsittelijä suoritetaan ylemmällä tasolla logiikkaa säilövässä komponentissa ja tapahtumasta seuraava tilan muutos käsitellään ja mahdollisesti varastoidaan ennen kuin uusi tila tarjotaan jälleen esitykselliselle komponentille, mikä lopulta esitetään sovelluksen käyttäjälle. (Abramov 2019.)

Abramovin mukaan nykypäivänä Reactin hookit ratkaisevat komponenttien vastuualueiden roolitukseen liittyvät haasteet. Koska hook-funktiot ovat itsenäisiä ja komponenteista riippumattomia ohjelmiston osia voi dataa visualisoivat komponentit kutsua tilaa hallinnoivia hookkeja ja silti pysyä esityksellisen komponentin määreissä. (Abramov 2019.)

Abramovin suunnittelumallissa keskitytään komponentin vastuuseen tilanhallinnan ja toimintojen näkökulmasta, kun määritellään sen roolia ohjelmistossa. Komponentteja jaotellaan mallin mukaan pikemminkin vastaamalla kysymykseen, mitä komponentti tekee, kun mitä se visuaalisesti esittää. Tämä ei ole Abramovin itsensä mielestäkään aina optimaalisin tarkastelumalli. (Malerba 2018.)

Käytänteet, joilla ei ole spesifistä määrettä tai dokumentaatiota voi aiheuttaa eriäviä mielipiteitä ohjelmointiyhteisöiden kesken. Esimerkiksi komponenttien vastuualueiden määrittämiselle ei ole luotu universaalia toteutustapaa, mikä voi johtaa erilaisiin toteutuksiin ohjelmiston sisällä. Ohjelmistokehityksessä tärkeä aspekti on koodin yhtenäisyys, joten tällaisten käytänteiden soveltaminen sovelluksessa vaatii sen kehitystiimiltä yhtenäisen linjauksen. Jos näkemys komponenttien roolituksesta eroaa ohjelmoijien kesken, voidaan taata

epäselkeä ja vaikeasti ymmärrettävä ohjelmistoarkkitehtuuri, jonka jatkokehittäminen on hankalaa. Käänteentekeväksi asiaksi nousee se, onko suunnittelumallien käyttö yhtenäistä ohjelmistoa rakentavien ohjelmoijien kesken. (Abramov 2019.)

Abramovin epäluulot kehittelemäänsä suunnittelumallia kohtaan modernissa React-kehityksessä on saanut ohjelmointiyhteisöt miettimään uusia lähestymistapoja komponenttien vastuualueiden määrittämiselle. Yksi lähestymistapa on Austin Malerban kehittelemä suunnittelumalli missä keskitytään komponenttien tuottamiin käyttöliittymiin ja mietitään minkä tasoista räätälöintiä komponentin sisällä sallitaan. Malerban mukaan komponentit voidaan jaotella universaaleihin-, globaaleihin- sekä näkymäkomponentteihin. (Malerba 2018.)

Universaalit komponentit ovat äärimmäisen abstrakteja sekä ohjelmiston logiikasta tietämättömiä itsenäisiä komponenttipohjia, joiden päälle voidaan hyvin joustavasti rakentaa globaaleja komponentteja sovelluksen käytettäväksi. Universaalit komponentit tulisi olla tarpeeksi geneerisiä, jotta niitä voitaisiin hyödyntää useissa ohjelmistoissa riippumatta niiden käyttötarkoituksista. Globaalit komponentit ovat riippuvaisia universaaleista vastikkeistaan ja niiden tarkoitus on rajata universaalien komponenttien joustavuutta. Globaalit komponentit hyödyntävät ohjelmistologiikkaa, mutta ovat tarpeeksi abstrakteja, että ne soveltuvat käytettäviksi eri puolilla ohjelmistoa. Näkymäkomponenttien tehtävänä on palvella loppuasiakasta, ja ne ovat yhdistelmä Abramovin suunnittelumallin tilan säilömiseen sekä esitykseen suunnitelluista komponenteista. Nämä komponentit vastaavat siis sovelluksen interaktiivisia osia. Näkymäkomponentit ovat globaalien komponenttien spesifimpiä vastikkeita, joiden suunnittelussa ei tarvitse miettiä niiden mahdollista uudelleenkäytettävyyttä. (Malerba 2018.)

Abramov kuitenkin painottaa, että komponenttien vastuualueiden liiallinen miettiminen voi helposti johtaa tarpeettomiin työvaiheisiin ja heikentää ohjelmiston luettavuutta. Kaikissa skenaarioissa ei ole järkevää toteuttaa komponenttien roolitusta vaan pitää sekä looginen että visuaalinen koodi samassa komponentissa. (Abramov 2019.)

## 5.2 Reactin antisuunnittelumallit

### 5.2.1 Paikallisen tilan alustaminen propsien kautta

Komponentin paikallista tilaa ei tulisi koskaan alustaa propsien kautta. Liitteen 6 esimerkissä UserNameInput-komponentti vastaa syötteeseen kirjoitetun käyttäjän nimen tallentamisesta. Käyttäjän nimi tallennetaan komponentin paikalliseen tilaan nimeltä name. Funktio setName päivittää onnistuneesti muuttuneen tilan, mutta komponentin uudelleen renderöinnin jälkeen tila on jälleen alustettu InputForm-komponentin pohjustamaan muotoon.

Alustamalla tilan alkuarvon propsin muodossa komponentille, jonka tarkoitus on muuttaa sitä, aiheutetaan turhaa epäselkeyttä komponentin tilaan sekä heikennetään koodin luettavuutta. Jos tila jostain syystä muuttuisi komponentin elinkaaren aikana eli ilman uudelleen renderöimistä ei komponentti osaisi käyttää päivitettyä tilaa, vaan se käsittelisi tilaa InputForm-komponentissa alustetussa muodossa. (Bertoli 2017, 270–272.)

### 5.2.2 Indeksin käyttäminen key-propsin arvona lista iteraatioissa

Kun React renderöi listoja se tarkastelee niiden yhdenvertaisuutta jokaisen renderöinnin aikana. Jos listaan lisätään uusi elementti ei React osaa itsessään käsitellä muuttunutta arvoa yksittäisenä muutoksen kohteena, vaan olettaa koko listan muuttuneen näin ollen päivittäen jokaisen elementin uudestaan DOM:iin. Listojen kasvaessa tapa aiheuttaa komponentin renderöintinopeuden hidastumista ja vaikuttaa tehottomalta lähestymistavalta. React tarjoaakin kehittäjilleen kyseiseen ongelmaan ratkaisuksi elementeille annettavan key-propsin, minkä tarkoituksena on yksilöidä jokainen elementti listan sisällä DOM:n renderöintiä varten. Key-propsin avulla React osaa paikantaa listasta elementin mihin kyseinen muutos on kohdistunut ja päivittää vain sen. Tämä vähentää renderöintioperaatioiden kokonaismäärää, avustaa Reactia hahmottamaan optimaalisimman renderöintipolun ja lisää komponentin suorituskykyä sekä vakautta. (Bertoli 2017, 204–205.)

Intuitiivinen ratkaisu olisi asettaa iteroitavan listan indeksi key-propsin arvoksi, sillä indeksi on juokseva kokonaisluku ja tällöin uniikki jokaisen elementin kohdalla. Ongelmana on, että uuden elementin luonti vaiheessa sen indeksinumeroksi asetetaan aina 0 riippumatta siitä, mihin kohtaan listaa kyseinen elementti halutaan asettaa. React siis olettaa, että jokaisen elementin indeksi muuttuu ja päivittää turhaan koko listan uudelleen mitätöiden key-propsin merkityksen kokonaisuudessaan. (Bertoli 2017, 276–279.)

Key-propsin arvoksi ei siis tulisi asettaa elementin indeksilukua, vaan jokin todellisesti yksilöllinen tieto. Uniikit tunnisteet ovat hyvä vaihtoehto key-propsin arvoiksi, sillä niillä ei ole osaa listan rakenteeseen ja eivät tule käytännössä koskaan muuttumaan. (Bertoli 2017, 279.)

### 5.2.3 Propsien levitys

ECMAScript 6 eli modernin JavaScript syntaksimalli on tuonut useita uusia kieliopillisia laajennuksia ohjelmoijien työn tueksi. Yksi uusista laajennuksista on niin sanottu levitysoperaattori. Sen avulla voidaan käsitellä iteroitavia elementtejä, kuten olioita sekä listoja (Spread syntax (…) s.a). Yksi käyttötarkoitus levitysoperaattorille on elementtien lisääminen React-komponentin propseihin (Wray 2019).

React-projekteissa nähdään usein tapa lähettää propsit lapsi komponenteille geneerisesti levitysoperaattorin avulla. Tämä tapa voi tuntua aluksi järkevältä, jos jokainen iteroitavan elementin attribuutti halutaan lapsi komponentin käyttöön. Levitys operaattori saa koodin näyttämään kauniilta ja luettavalta. Ohjelmistokehitys on kuitenkin jatkuvaa eikä sen iteroitavien elementtien tila tule luultavasti pysymään samanlaisena ikuisesti. Näin ollen vastaan tulee aika, kun lapsi komponentille lähetetään jotain arvoja mistä se ei ole kiinnostunut, mikä rikkoo SOLID-periaatteita. (Wray 2019.)

Levitysoperaattori estää myös propsien nimellisten arvojen asettamisen vähentäen ohjelmoijien luovia nimeämiskäytänteitä (Wray 2019). Propsien nimien piilottaminen toimii vasten ohjelmointiperiaatteita, sillä kyseinen koodipätkä ei anna ohjelmoijille mitään informaatiota. Propsien levityksellä voidaan myös täysin rikkoa ohjelmisto, jos HTML-elementeille asetetaan niille kelpaamattomia attribuutteja. (Bertoli 2017, 280–281.)

### 5.3 Suunnittelumallien varjopuoli

Suunnittelumallit ja niiden olemassaolon tiedostaminen on ehdottomasti hyvä asia ohjelmointiyhteisöille. Usein ohjelmoijat eivät kuitenkaan harrasta kriittistä ajattelua suunnittelumalleja implementoidessaan. Esimerkki yleisestä ohjelmoinnissa esiintyvästä antisuunnittelumallista on tarpeeton koodin optimointi tilanteissa, missä kyseinen työskentely ei tuo minkäänlaista lisäarvoa lopputulokseen. Ohjelmoijilla on paha tapa miettiä liikaa ohjelmistojen optimointiratkaisuita varmuuden vuoksi, vaikka optimoinnin kohteena oleva ohjelmiston osa ei sitä kaipaisi. Amerikkalainen tietojenkäsittelytieteen tutkija ja matemaatikko Donald Knuth ilmaisi asian seuraavasti: (Christopher 2019.)

"*The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.*"

Toinen todellinen ongelma on tarpeeton tiedostorakenteiden refaktorointi. Ongelma ilmenee yleisesti kokemattomilla ohjelmoijilla. On hienoa, että ohjelmiston tiedostorakenteen optimointi kiinnostaa, mutta ammattilaisprojektien kansiohierarkiaa ei kannata kopioida suoraan jokaiseen projektiin. Välillä optimaalisin tapa esittää React-komponentti on säilyttää kaikki siihen liittyvä koodi samassa tiedostossa. Luettavuutta tai ohjelmointityön tehokkuutta ei ainakaan lisää se, että komponentin osia pitää etsiä tunti kaupalla ohjelmiston eri kansioista. (Wray 2019.) Stoverin (2019) mukaan Reduxin kehittäjä Dan Abramov on kehottanut ohjelmointiyhteisöitä järjestelemään sovelluksen kansiorakenteen yksinkertaisesti

siten, että se palvelee kehittäjäänsä mahdollisimman hyvin. Tarpeetonta optimointia tulisi välttää myös tässä kontekstissa.

Useat suunnittelumallit ovat valitettavasti vanhanaikaisia. Niiden edustama fundamentaalinen ongelma on yhä olemassa, mutta dokumentoidut malliratkaisut eivät enää ole valideja modernissa ohjelmistokehityksessä ja niiden implementointi sovelluksiin on ohjelmoijan soveltavien taitojen varassa. Nykypäivän funktionaalinen paradigma, mitä Reactilla toteutettu käyttöliittymäohjelmointikin palvelee, on hyvin erilainen Gang of Four:n ajoilta. Gang of Four:n suunnittelemat mallit ovatkin suunnattu olio-orientoituun ohjelmointiin ja palvelevat vanhanaikaisia perimishierarkioita sekä luokkakomponentteja. Suunnittelumallien käyttö virheellisissä asiayhteyksissä tai ilman asianmukaista perustelua voi muodostaa kehitystyöhön tehottomia prosesseja, heikentää koodin luettavuutta ja aiheuttaa turhaa päänvaivaa ohjelmoija kollegoille. (Cook 2018.)

## 6 Päätelmät

Useat suunnittelumallit ovat laajalti tunnettuja ja niitä käytetään hyvin erilaisissa projekteissa. Niiden avulla ratkotaan kuitenkin samoja fundamentaalisia haasteita. Perinteiset suunnittelumallit ovat olleet olemassa jo pitkään, minkä vuoksi ohjelmoija voi olla lähes varma siitä, että kyseiset mallit ovat hyvin pitkälle testattuja ja optimoituja. Kokeneiden ohjelmoijien suunnittelemat mallit voivat tuntua täysin triviaaleilta vaihtoehdoilta kokemattomille ohjelmoijille. On kuitenkin hyvä tiedostaa, että suunnittelumallit eivät automaattisesti estä kelvottoman arkkitehtuurin luontia. Voi olla, että juurisyy ohjelmiston huonolle suunnittelulle on suunnittelumallin implementointi ohjelmistoon.

SOLID-suunnittelumallit ovat geneerisiä eli niitä ei ole rajattu toimimaan tietyn ympäristön tai edes saman ohjelmointikielen kanssa, mikä lisää niiden uudelleenkäytettävyyttä. Ne vähentävät ohjelmoijien työmäärää puhtaasti ohjelmiston luettavuutta ja rakennetta ehostavissa tehtävissä.

Mitä useampi ohjelmointitiimi soveltaa näitä malleja sitä helpompaa ohjelmistokehittäminen on yhteisöiden välillä ja yritysten sisällä. Mallien opettelu vahvistaa yhteisöiden kesken tapahtuvaa viestintää, sillä niiden tunteminen edesauttaa mahdollisien ratkaisuiden löytymistä ja kompleksisia metodeita ei tarvitse selittää, kun kyseiset termit ovat kollegoille tuttuja. Syvä yhteisöllisyys edesauttaa myös mallien jatkokehitystä ja luo mahdollisuuksia kehittää uusia suunnittelumalleja. Suunnittelumallien käyttö voi vähentää projektin kokonaista koodirivien määrää yhdistämällä useasti suoritettavaa logiikkaa yhteen korkeamman tason yksikköön.

Ohjelmiston rakentaminen tulisi olla mahdollisimman yksinkertaista mahdollisimman pitkään. Jos suunnittelumallien implementoiminen ohjelmistoon lisää sen komponenttien määrää tai kompleksisuutta huomattavasti olisi malli syytä unohtaa ja miettiä yksinkertaisempi tapa ratkaista ongelma.

Toisaalta suunnittelumallien tiedostaminen tekee ohjelmoijista ammattitaitoisempia ja saa heidät ajattelemaan sovelluskehitystä uusin silmin. Tunnistaessaan perinteiset mallit ohjelmoija pystyy tehokkaammin analysoimaan kehittämänsä järjestelmän tilaa. Mallien avulla voidaan kääntää analyyttiset mallinnukset toteutettaviksi ratkaisuiksi. Lisäksi niiden soveltaminen voi vähentää tarvittavan refaktoroimisen määrää olettaen, että jokainen ohjelmointiyhteisön jäsen toimii samojen periaatteiden mukaan. Mitä vankempi ohjelmiston peruskivi on, sitä vakaampaa ja luontevampaa sen jatkokehittäminen on.

Ohjelmointiyhteisöiden tulisi harkita tukevatko suunnittelumallit kehitystyötä vai onko niiden implementoimisessa vaarana suunnata ohjelmiston arkkitehtuuria väärään suuntaan. Mallien hyödyntäminen pitäisi olla yhteisöiden yhteinen linjaus ja sovitussa suunnittelumallissa tulisi pysyä koko ohjelmiston kehityskaaren ajan. Ohjelmistokehitys on luovaa työtä ja kriittinen ajattelu olisi aina paikallaan. Mitään suunnittelumalleja ei tulisi mieltää säännöiksi tai ainoiksi tavoiksi ratkaista ongelmia.

Suunnittelumallien tarkoitus on mahdollistaa ohjelmoijille syvempi ja abstraktimpi ajattelutapa kehitystyössään. Niitä ei voi käyttää tekosyynä ohjelmointi fundamenttien unohtamiselle tai tarpeettoman koodin kirjoittamiselle. Useat suunnittelumallit voivat ohjata ohjelmoijia kehittämään ohjelmistoa liian abstraktiin suuntaan ilman, että he antaisivat tilaa kriittiselle ajattelulle. Suunnittelumallit eivät sovi jokaiseen tilanteeseen ja voivat aiheuttaa enemmän hämmennystä kuin yhtenäisyyttä ohjelmoijien kesken.

## Lähteet

Abhishek, N., Akshat, P. 2015. React Native for iOS Development. Apress. Kalifornia.

Abramov, D. 2019. Presentational and Container Components. Luettavissa: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0. Luettu: 23.4.2021.

Bertoli, M. 2017. React Design Patterns and Best Practices. Packt Publishing Ltd. Birmingham.

Bhargav. 2017. Enhancing React Components: Inheritance & Composition. Luettavissa: https://www.codementor.io/@imbhargav5/extending-react-components-inheritance-composition-qo59dqili. Luettu: 26.4.2021.

Building Your Own Hooks. s.a. Luettavissa: https://reactjs.org/docs/hooks-custom.html. Luettu: 22.4.2021.

Christopher, T. 2019. 5 Anti-Patterns in JavaScript to Avoid When Working With Collections. Luettavissa: https://jsmanifest.com/5-anti-patterns-in-javascript-to-avoid-when-working-with-collections/. Luettu: 12.4.2021.

Composition vs Inheritance. s.a. Luettavissa: https://reactjs.org/docs/composition-vs-inheritance.html. Luettu: 21.4.2021.

Cook, D. 2018. Design Patterns should be considered harmful. Luettavissa: https://medium.com/comparethemarket/design-patterns-should-be-considered-harmful-45ac06cd15ce. Luettu: 27.4.2021.

Ezeokoye, M. 2019. S.O.L.I.D principles with React. Luettavissa: https://medium.com/@jaymykels69/s-o-l-i-d-principles-with-react-cd43fc93b1be. Luettu: 14.4.2021.

Gamma, E., Vlissides, J., Helm, R. & Johnson, R. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional.

Gold, T. 2018. How to apply SOLID principles in React applications. Luettavissa: https://blog.usejournal.com/how-to-apply-solid-principles-in-react-applications-6c964091a982. Luettu: 16.4.2021.

Hamedani, M. 2018. React Virtual DOM Explained in Simple English. Luettavissa: https://programmingwithmosh.com/react/react-virtual-dom-explained/. Luettu 13.4.2021.

Introducing Hooks. s.a. Luettavissa: https://reactjs.org/docs/hooks-intro.html. Luettu: 22.4.2021.

Introducing JSX. s.a. Luettavissa: https://reactjs.org/docs/introducing-jsx.html. Luettu: 13.4.2021.

Kumar, S. 2018. MVC Design Pattern. Luettavissa: https://www.geeksforgeeks.org/mvc-design-pattern/. Luettu: 16.4.2021.

Levkovsky, M. 2017. Luettavissa: https://hackernoon.com/thinking-in-redux-when-all-youve-known-is-mvc-c78a74d35133. Luettu: 12.4.2021.

Lobera, A. 2019. React is all about composition. Luettavissa: https://medium.com/leanjs/react-is-all-about-composition-f9f49dec183c. Luettu: 4.5.2021.

MacDonald, M. 2019. Is It Time to Get Over Design Patterns? Luettavissa: https://medium.com/young-coder/is-it-time-to-get-over-design-patterns-8851864a6834. Luettu: 3.5.2021.

Malerba, A. 2018. A New Approach to React Component Design. Luettavissa: https://www.freecodecamp.org/news/a-new-approach-to-react-component-design-2bf76a87add1/. Luettu: 23.4.2021.

Martin, R. 2017. Clean Architecture. Pearson Education Inc, US. Boston.

Mbanugo, P. 2018. Exploring the what and the why of Redux. Luettavissa: Exploring the what and the why of Redux (freecodecamp.org). Luettu: 13.4.2021.

Mitchell, J. 2020. Practical examples for applying SOLID principles in your React applications. Luettavissa: https://dev.to/jmitchell38488/practical-examples-for-applying-solid-principles-in-your-react-applications-19ab. Luettu: 14.4.2021.

Mukhiya, S. & Hung, H. 2018. An Architectural Style for Single Page Scalable Modern Web Application.

Nwamba, C. 2019. When (and when not) to use Redux. Luettavissa: https://blog.logrocket.com/when-and-when-not-to-use-redux-41807f29a7fb/. Luettu: 7.5.2021.

Osmani, A. 2012. Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide. O'Reilly Media, Inc, USA.

Provider. s.a. Luettavissa: https://react-redux.js.org/api/provider. Luettu: 18.4.2021.

Redux FAQ: React Redux. s.a. Luettavissa: https://redux.js.org/faq/react-redux. Luettu: 18.4.2021.

Spread syntax (…). s.a. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax. Luettu: 27.4.2021.

Stover, C. 2019. Optimal file structure for React applications. Luettavissa: https://charlesstover.medium.com/optimal-file-structure-for-react-applications-f3e35ad0a145. Luettu: 27.4.2021.

Tutorialspoint 2021. MVC Framework – Introduction. Luettavissa: https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm. Luettu: 17.2.2021.

Why Use React Redux? s.a. Luettavissa: https://react-redux.js.org/introduction/why-use-react-redux. Luettu: 18.4.2021.

Wray, T. 2019. React Anti-Patterns. Luettavissa: https://tylerwray.me/blog/react-anti-patterns/. Luettu: 27.4.2021.

Yosef, N. The Ugly Side Of Redux. Luettavissa: https://codeburst.io/the-ugly-side-of-redux-6591fde68200. Luettu: 7.5.2021.

## Liitteet

### Liite 1. Verkkosivu teknologioiden hyödyntäminen React komponentissa

```
const Togglable = React.forwardRef((props, ref) => {
  const [visible, setVisible] = useState(false);

  const hideWhenVisible = { display: visible ? 'none' : '' };
  const showWhenVisible = { display: visible ? '' : 'none' };

  const toggleVisibility = () => {
    setVisible(!visible);
  };

  useImperativeHandle(ref, () => {
    return {
      toggleVisibility
    };
  });

  return (
    <div>
      <div style={hideWhenVisible}>
        <button className="button" onClick={toggleVisibility}>
          {props.buttonLabel}
        </button>
      </div>
      <div style={showWhenVisible}>
        {props.children}
        <button className="button" onClick={toggleVisibility}>
          Cancel
        </button>
      </div>
    </div>
  );
});
```

### Liite 2. Propsien poraaminen

```
const App = () => {
  const [user, setUser] = React.useState('Bob')
  return (
    <div>
      <Dashboard user={user} />
    </div>
  )
}

const Dashboard = ({ user }) => {
  return (
    <div>
      <DashboardNav />
      <DashboardContent user={user} />
    </div>
  )
}

const DashboardContent = ({ user }) => {
  return (
    <div>
      <h3>Dashboard Content</h3>
      <WelcomeMessage user={user} />
    </div>
  )
}

const WelcomeMessage = ({ user }) => {
  return (
    <div>
      <p>Welcome {user}!</p>
    </div>
  )
}
```

**Liite 3. Redux – Tilan hallinta**

```
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)

const App = () => {
  return (
    <div>
      <Dashboard />
    </div>
  )
}

const Dashboard = () => {
  return (
    <div>
      <DashboardNav />
      <DashboardContent />
    </div>
  )
}

const DashboardContent = () => {
  return (
    <div>
      <h3>Dashboard Content</h3>
      <WelcomeMessage />
    </div>
  )
}

const WelcomeMessage = () => {
  const user = useSelector(state => state.user)
  return (
    <div>
      <p>Welcome {user}!</p>
    </div>
  )
}
```

**Liite 4. JSX kompositio ja children propsi**

```
const App = () => {
  const [user, setUser] = React.useState('Bob')
  return (
    <div>
      <Dashboard>
        <DashboardNav />
        <DashboardContent>
          <h3>Dashboard Content</h3>
          <WelcomeMessage user={user} />
        </DashboardContent>
      </Dashboard>
    </div>
  )
}

const Dashboard = ({ children }) => {
  return (
    <div>
      { children }
    </div>
  )
}

const DashboardContent = ({ children }) => {
  return (
    <div>
      { children }
    </div>
  )
}

const WelcomeMessage = ({ user }) => {
  return (
    <div>
      <p>Welcome {user}!</p>
    </div>
  )
}
```

**Liite 5. Kompositio – Komponenttien kustomointi**

```
const Button = (props) => {
  const buttonColor = props.color
  return (
    <div>
      <button style={{ backgroundColor: buttonColor }}>
        {props.children}
      </button>
    </div>
  )
}

const DeleteButton = () => {
  return (
    <div>
      <Button color="red" onClick={deleteFunction}>
        Delete
      </Button>
    </div>
  )
}
```

**Liite 6. Paikallisen tilan alustaminen propsien kautta**

```
const UserNameInput = (props) => {
  const [name, setName] = useState(props.name)

  const handleSubmit = () => {
    // send name to the server and database
  }
  const handleNameChange = (event) => {
    event.preventDefault()
    setName(event.target.value)
  }
  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input type="text" name="name" value={name} onChange={handle
NameChange}/>
        <button type="submit">Save User</button>
      </form>
    </div>
  )
}

const InputForm = () => {
  return (
    <UserNameInput name='Guest' />
  )
}
```