TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

VESA LAHDENPERÄ

YLEISTETTÄVÄN WEB-NÄKYMÄARKKITEHTUURIN SUUNNITTELU JA TOTEUTUS MALLIPOHJAISEEN YMPÄRISTÖÖN

Diplomityö

Tarkastaja: Prof. Kari Systä
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkö tiedekuntaneuvoston
kokouksessa 03.12.2014

TIIVISTELMÄ

VESA LAHDENPERÄ: Yleistettävän web-näkymäarkkitehtuurin suunnittelu ja toteutus mallipohjaiseen ympäristöön
Tampereen teknillinen yliopisto
Diplomityö, 76 sivua, 1 liitesivua
Toukokuu 2015
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Kari Systä
Avainsanat: Mallipohjainen ymäristö, Web-järjestelmä, HTML5, Näkymäarkkitehtuuri

Web-järjestelmissä on usein ratkaisuja, joissa ei hyödynnetä uusien tekniikoiden mahdollisuuksia. Varsinkin HTML5-tekniikan kehittyminen ja yleinen tietokoneiden suorituskyvyn kehitys mahdollistavat yhä monipuolisempien sovellusten toteuttamisen käyttäjän selaimessa.

Tässä työssä suunnitellaan ja toteutetaan mallipohjaiseen web-järjestelmään uudenlainen arkkitehtuuri, jossa pyritään siirtämään järjestelmässä ajettavia sovelluksia ajettavaksi käyttäjän selaimelle. Ongelman tekee haastavaksi mallipohjainen ympäristö, jonka perustana toimii mallinnuksen periaatteita noudattava tietokanta.

Työssä esitellään muutama sovellus, jonka kaltaisia uudella arkkitehtuurilla olisi tarkoitus toteuttaa. Lisäksi määritellään arkkitehtuurille ja sen ominaisuuksille liuta eitoiminnallisia ominaisuuksia. Sovellusten esittelyn pohjalta arvioidaan samankaltaisuuksia sovellusten välillä. Käyttämällä hyväksi arvioinnin havaintoja sekä arkkitehtuurin vaatimuksia voidaan suunnitella arkkitehtuurin konkreettisia toimintatapoja.

Arkkitehtuurin suunnittelussa keskityttiin muodostamaan yleistettävä arkkitehtuuri, jossa muodostetaan palvelimelta saatavasta mallidatasta näkymiä, joissa esitetään malliin kuuluvia mallielementtejä korttipohjaisissa muodoissa. Näihin kortteihin on mahdollista liittää toiminnallisuutta erilaisilla uudelleenkäytettävillä käytöskomponenteilla.

Työn tuloksena dokumentoitiin arkkitehtuurin keskeisimmät ratkaisut sekä toteutettiin selainsovellusarkkitehtuuri luomaan ja ajamaan korttipohjaisia sovelluksia. Työn tulosten perusteella voidaan todeta toteutettujen ratkaisujen olevan sekä teoriassa että käytännössä toimivia.

ABSTRACT

VESA LAHDENPERÄ: Web-view architecture design and implementation in model based environment
Tampere University of Technology
Diplomityö, 76 pages, 1 Appendix pages
May 2015
Master's Degree Programme in Information Technology
Major: Software engineering
Examiner: Prof. Kari Systä
Keywords: Model based environment, Web-system, HTML5, View Architecture

Web-systems usually contain solutions where new opportunities of new technologies are not utilized. Development in HTML5 technologies and the general increase in computing power give way to implement more versatile applications in browser.

In this thesis an architecture is designed and implemented for model based web-system where applications in the system are relocated to run in browsers. The challenge comes from the model based environment and its database that uses principles of modelling.

The thesis presents a few example applications that serve as guidelines for future applications that need to be implemented using the architecture. In addition some non-functional requirements are laid down for the architecture and its properties. Using the example applications we can compare the similarities between them and in combination with requirements we can guide the architectural design process.

The architecture was designed with generalization in mind and it is used to create views out of model data that a server provides. The views consist of cards that represent the models model-elements. These cards can be enhanced by applying functionality to them using reusable behaviour components.

The results are documented core solutions of the architecture and software architecture in the browser that creates and runs card-based applications. Used solutions can be declared valid in both theory and practice.

ALKUSANAT

Tavallisesti tässä kohdassa listataan nimiä ja kiitetään heitä. En halua nostaa ketään yksittäistä henkilöä jalustalle. Oletan lukijan tietävän, mikäli hän on elämääni vaikuttanut.

Kiitos siitä.

Tampere, 17.5.2015

Vesa Lahdenperä

SISÄLLYS

KUVALUETTELO

TAULUKKOLUETTELO

LYHENTEET JA MERKINNÄT

UML	Unified Modeling Language
OMG	Object Management Group
MOF	Meta object Facility
Realm	Cometa Solutions Oy:n mallipohjainen ympäristö
HTTP	Hypertext Transfer protocol
TCP	Transmission Control protocol
DOM	Document Object Model
CSS	Cascading Style Sheets
MVC	Malli-Näkymä-Ohjain -arkkitehtuurimalli
ECS	Entiteetti-Komponentti-Järjestelmä -arkkitehtuurimalli
SPA	Single-Page Application

1. JOHDANTO

Ohjelmistojen toteutustekniikat kehittyvät jatkuvasti. Tästä syystä Web-järjestelmissä usein havaitaan ratkaisuja, joissa nykyajan mahdollisuuksia ei käytetä hyödyksi tehokkaalla tavalla.

Web-järjestelmät ovat hyvä esimerkki, sillä niiden kehitystyössä joudutaan suunnittelemaan sekä palvelinpään ohjelmistoa että mahdollista asiakaspään sovellusta. Vaatimusten kasvaessa ja käyttöliittymien monipuolistuessa sovellusten toteuttaminen vaatii yhä enemmän painoa asiakaspään sovelluksiin, jotka hyödyntävät palvelinpään ohjelmistoja.

Toisaalta Web-järjestelmien muuntaminen uusille tekniikoille sopiviksi ei välttämättä ole triviaali työ. Usein kiire ja yksilötyö tuottavat hyvin paljon dokumentoimattomia ratkaisuja ja tietynlaiseen paradigmaan kaartuvia sovelluksia.

Tässä työssä on tavoitteena on suunnitella ja toteuttaa Realm-nimiseen järjestelmään yleinen arkkitehtuuri, jossa sovelluksien toimintoja voidaan helposti uudelleenkäyttää eri sovelluksissa. Tämän lisäksi arkkitehtuurin tavoite on vahvistaa sovellusten integroimista osaksi Realm-järjestelmää ja ottaa käyttöön mallipohjaisen ympäristön tarjoamat työkalut uusien näkymien rakentamiseen.

Ratkaisuksi esitetään uutta arkkitehtuuria, jossa näkymät koostetaan niin sanotuista Korteista. Kortin tarkoitus on olla yleinen mallipohjaisen ympäristön rakenne, jolla voidaan sekä esittää tietoa järjestelmän malleista että sitoa toiminnallisuutta, jolla voidaan muokata sitä edustavan mallin tietoja.

Luvussa 2 käsitellään työn ymmärtämisen kannalta keskeiset taustakäsitteet ja taustateknologiat. Luvussa käydään myös läpi suunniteltavan arkkitehtuurin ymmärtämisen kannalta keskeiset arkkitehtuuri- ja suunnittelumallit. Luvussa 3 esitellään muutama sovellus, joiden kaltaisia arkkitehtuurilla tulee kyetä toteuttamaan. Luvussa 4 esitellään toteutettavan arkkitehtuurin vaatimukset ja luvussa 5 käsitellään vaatimuksien pohjalta suunniteltu arkkitehtuuri. Luvussa 6 esitellään muunnellulla arkkitehtuurilla toteutetut sovellukset, joiden toteutuksia arvioidaan luvussa 7. Luku 8 sisältää yhteenvedon.

2. TAUSTAKÄSITTEET JA -TEKNOLOGIAT

Tässä luvussa käsitellään työn ymmärtämisen kannalta tärkeimmät taustatiedot, taustateknologiat ja teoria. Luvussa käsiteltäviä asioita ovat mallipohjaisuus, mallinnus, olemassaoleva alusta, web-ohjelmistojen toteutustekniikat sekä työn kannalta tärkeimmät arkkitehtuuri- ja suunnittelumallit.

2.1 Ohjelmistojen mallinnus

Ohjelmistojen suunnittelussa yleinen työkalu on mallinnus. Mallinnuksessa suunniteltavasta ohjelmistosta luodaan ohjelmistoa monesta näkökulmasta kuvaava malli. Ohjelmistoa kuvataan mallissa useilta eri abstraktiotasoilta, joista kuvataan ainoastaan merkitykselliset tiedot.

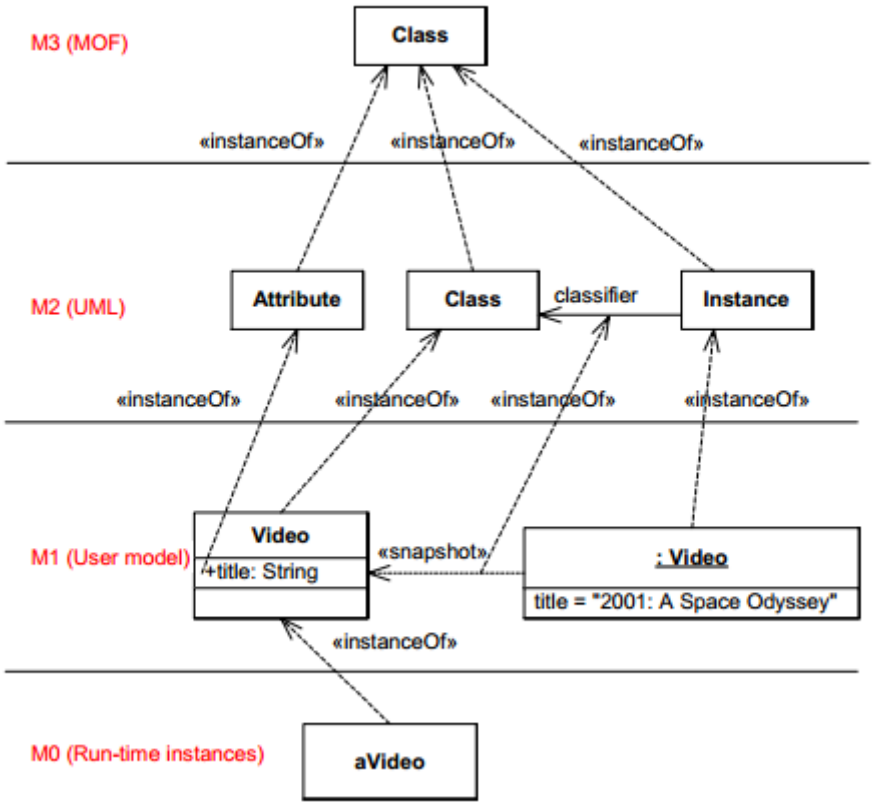
Mallien pääkäyttötarkoitus ohjelmistotyössä on ohjelmiston toteutukseen liittyvien ongelmien analysointi. Malleja myös voidaan käyttää sovelluksen kehityksessä mukana olevien ihmisten väliseen kommunikaation selkeyttäjänä. Malli luo myös pohjan yhtenäistä kieltä käyttävien dokumenttien kirjoittamiseen. [12]

Mallieja kuvataan mallinnuskielien määrittelemillä tavoilla. Työn kannalta tärkein mallinnuskieli on Unified Modeling Language (**UML**). UML on Object Management Groupin (**OMG**) luoma ja ylläpitämä graafinen mallinnuskieli, jonka uusin standardi 2.0 julkaistiin vuonna 2004.

Mallinnuskielet perustuvat mallissa käytettävien mallinnuskäsitteiden määritelmiin. Mallinnuskäsitteiden määrittelyiden luomaa tasoa kutsutaan metamalliksi. Koska metamalli itsessään on malli, voi metamalli perustua johonkin abstraktimpaan malliin, jota kutsutaan metametamalliksi. Teoriassa meta-tasojen lisäämistä malliin voitaisiin jatkaa loputtomiin, mutta useimmissa tapauksissa metametamalli suunnitellaan siten, että se voi kuvata itse itsensä. UML perustuu metametamalliin, jota kutsutaan Meta Object Facilityksi (**MOF**).

Kuvassa 2.1 [16] on kuvattuna UML:n metamallihierarkkia, joka on jaettu neljään tasoon, joiden abstarktiotaso laskee alaspäin. Tasolla M3 on MOF. Taso kuvaa sitä,

miten mallinnuskielet mallinnetaan. Taso on suunniteltu myös siten, että se määrittelee itse itsensä. Tasolla M2 sijaitsee UML-kielen määrittelevä taso. Taso määrittelee M1-tason tuottamiseen vaaditun mallinnuskielen. Tasolla M1 on varsinainen käyttäjän UML-kielellä toteuttama malli. Taso kuvaa malliin perustuvan sovelluksen ajonaikaisten olioiden rakennetta.



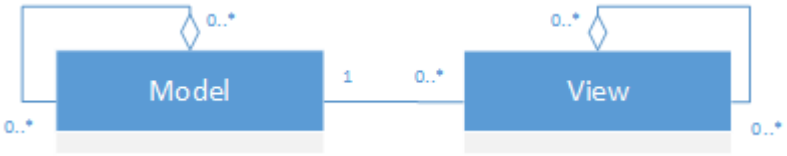
Kuva 2.1 *UML-kielen abstraktion tasot.*

2.2 Mallipohjaisuus

Tässä työssä mallipohjaisuus tarkoittaa sitä, että kaikki sovelluksien esittämä data luetaan sovellukseen mallinnus-metodologiaa toteuttavasta tietokannasta. Tietokanta koostuu metametamalleista, metamalleista ja varsinaisista malleista. Varsinaista sovelluksen tietosisältöä varten mallinnetaan sovelluksesta malli, jossa määritellään, mitä mallielementtejä mallissa voi olla.

Yksittäisissä malleissa erotellaan malli- ja näkymädata toisistaan malli- ja näkymäelementeillä. Yksittäinen mallielementti kuvaa malliin liitettyjä ominaisuuksia, muttei sisällä tietoa kuinka kyseinen elementti esitetään käyttäjälle. Näkymäelementit

ovat mallielementtien graafisia esitysmuotoja, joita voidaan luoda tarvittaessa mallielementeille useita eri käyttökohteita varten. Mallielementin ja näkymäelementin suhde toisiinsa kuvataan kuvassa 2.2. [10]



Kuva 2.2 *Malli- ja näkymäelementtien suhde.*

Mallit ovat dynaamisia tietokantaskeemoja, joiden avulla voidaan generoida abstraktiotasoissa alempien mallien tietokantaskeemat. Metametamallilla voidaan siis määritellä metamallin tietokantaskeema, ja metamallin tietokantaskeemalla määritellään mallin tietokantaskeema. Muutokset metamalliin vaikuttaa dynaamisesti siitä johdettujen mallien tietokantaskeemoihin. Tämä aiheuttaa mallipohjaiseen järjestelmään rakennettavien sovellusten suunnittelulle omat haasteensa. Ohjelmisto tulisikin suunnitella siten, että muutokset metamalliin eivät vaikuta ohjelmiston toimintaan. Tämä on mahdollista käyttämällä elementtipohjia.

Elementtipohjat ovat yleisiä tapoja esittää näkymäelementeissä sen mallielementin metaominaisuudet. Ominaisuuksiin luetaan, mihin mallielementtityyppeihin näkymäelementit liittyvät ja minkälaisia lapsielementtejä niille voi asettaa. Pohjat eivät varsinaisest määrittele näkymäelementtien ulkoasua, vaan toimivat ainoastaan määritelmänä siitä, mitä tietoja näkymäelementissä näytetään. Käyttämällä elementtipohjia voidaan mallielementille tuottaa samankaltaisia näkymäelementtejä eri sovelluksiin.

Koska sovellukset voivat päätellä hyvin paljon sovelluksen rakenteesta mallielementtien metamallien avulla, voidaankin sovelluksten käyttöliittymät ja niiden perustoiminnallisuudet koostaa siihen liittyvien mallielementtien elementtipohjien mukaan. Näin esimerkiksi yhdelle mallielementille voidaan muodostaa näkymäelementti, jossa elementtipohjaa käyttämällä määritellään näkymäelementissä näytettävät mallielementin ominaisuudet. Koska elementtipohjat ovat yleisiä, voidaan näkymiä toteuttaa työkalukohtaisesti eri sovelluksille. Tämän johdosta mallipohjaisessa järjestelmässä on mahdollisuus toteuttaa tehokkaasti mallielementtien muokkauksen ja katselmoinnin tuki eri tekniikoilla toteutetuille sovelluksille. Samaa dataa voitaisiin täten muokata sekä web-ympäristössä että perinteisissä työpöytäsovelluksissa.

2.3 Realm-järjestelmä

Realm-järjestelmä on Cometa Solutions Oy:n mallipohjaisen tietokannan päälle rakennettu integrointi- ja sovellusalusta web-pohjaisille työkaluille. Järjestelmään pyritään toteuttamaan arkkitehtuurimuutos siten, että sovellusten toteutuksen paino siirretään palvelimelta selaimelle. Nykyinen järjestelmä toimii täten toteutettavan arkkitehtuurin pääasiallisena palvelimena ja tuottaa arkkitehtuurin avulla kehitettyjen sovellusten pohjasivut.

Kuva 2.3 *Realm-järjestelmän yleisarkkitehtuuri*

Kuten kuva 2.3 osoittaa, Realm-järjestelmän arkkitehtuuri jakautuu kolmeen osaan: mallipohjainen tietokanta, palvelin sekä asiakaspään selainsovellukset.

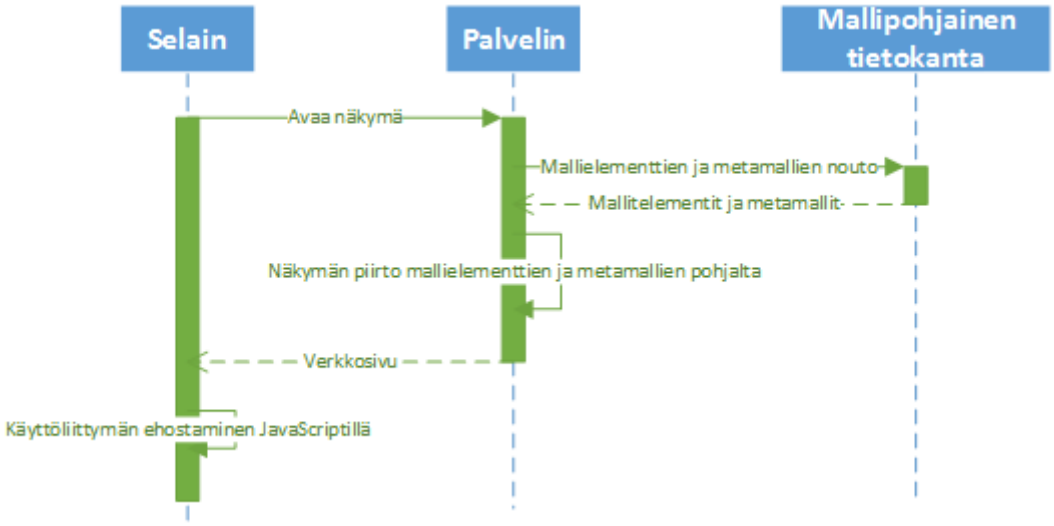
Mallipohjainen tietokanta on tietokanta, joka noudattaa mallinnuksen periaatteita. Realm-järjestelmässä kaikki käsiteltävä tieto tallennetaan tietokannan mallinnustavan mukaisiin malleihin. Tietokannan vastuulla on täten tiedon säilytys, tiedon tarjoaminen palvelimen käyttöön ja mallirakenteiden eheyden ylläpito.

Palvelimen vastuualueita ovat näkymien luominen sekä tiedon välittäminen tietokannan ja selaimen välillä. Palvelimen vastuulla on myös tiedon muokkausta koskevien pyyntöjen käsittely ja ja muokkauksien päivitys tietokantaan. Tämän lisäksi palvelimen hallitsee käyttöoikeuksia.

Selaimen tehtävänä on esittää käyttäjälle näkymiä mallidatasta HTML-muodossa. Selaimen tehtävänä on myös toimia mallidatan muokkaustyökaluna. Nykyisessä järjestelmässä selaimen sovellusten toteutus on rajoittunut erillisiin verkkosivulla ajettaviin scripteihin, eikä selaimessa täten ole varsinaista hyvin määriteltyä sovellusarkkitehtuuria. Selaimen ja palvelimen välissä ei noudateta mitään yleistä web-ohjelmoinnin rajapintaa, vaan rajapinta on tarkoitukseen räätälöity. Tietoa välitetään palvelimen ja selaimen välillä useimmiten HTML- tai JSON-muodossa.

Realm-järjestelmän näkymän rakentamisen vaiheet esitetään kuvassa 2.4. Palvelimelle tulee kutsu käyttäjän selaimelta, jossa pyydetään tietyn näkymän avaamista.

Palvelin vastaa kutsuun hakemalla näkymään liittyvät mallielementit sekä metamallit tietokannasta. Tämän jälkeen palvelin muodostaa näkymän HTML-merkkauksen ja palauttaa sen selaimelle. Tämän jälkeen selain vastaanottaa näkymän ja ajaa käyttöliittymää ehostavia scriptejä.



Kuva 2.4 *Näkymän rakentaminen Realm-järjestelmässä*

Näkymä muodostetaan mallielementtien ja niiden metamallien pohjalta, joten mallipohjaisen ympäristön elementtipohjia ei käytetä hyödyksi. Myöskään ympäristön tarjoamia näkymäelementtejä ei käytetä tehokkaasti hyödyksi. HTML-merkkaus muodostetaan palvelinohjelmiston kirjastossa, joka muodostaa HTML-muotoiset elementit mallielementtien esitystä varten. Mallipohjaisen ympäristön tehokkaiden työkalujen käyttämättä jättäminen on yksi suurimmista syistä uuden arkkitehtuurin luomiselle.

2.4 Web-ohjelmistot ja niiden toteutustekniikat

Web-ohjelmistot ovat ohjelmistoja, jotka käyttävät ajoympäristöinään verkkoselaimia. Suurimmat erot natiivien työpöytäsovellusten ja web-sovellusten välillä ovat käyttöjärjestelmä- ja käyttölaiteriippumattomuus. Yksi web-ohjelmistototeutus teoriassa toimii kaikilla käyttöjärjestelmillä ja -laitteilla, jotka tarjoavat verkkoselaimen käyttäjälle. [5]

Natiiveista työpöytäsovelluksista poiketen web-sovelluksien käyttö useimmiten vaatii verkkoyhteyden. Tämä johtuu siitä, että web-sovellusta käytettäessä sovellus on ladattava käyttäjän selaimeen. Web-sovelluksia voi myös toteuttaa siten, että

offline-käyttö on sovelluksen latauksen jälkeen mahdollista HTML5 application cache -rajapinnan avulla. [22] Siinä missä natiivit työpöytäsovellukset vaativat erilliset asennus ja päivitysohjelmansa, web-ohjelmiston asennus ja päivitys tapahtuu yksinkertaisesti verkkoselaimessa lataamalla ohjelmisto palvelimelta normaalin web-pyynnön yhteydessä. Koska verkkoselaimelle ladattu ohjelma päivittyy automaattisesti, palvelimen saamat pyynnöt web-ohjelmistoilta ovat aina teoriassa saman ohjelmistoversion pyyntöjä. [3]

Web-ohjelmistot, kuten web yleensäkin, perustuvat asiakas-palvelin -arkkitehtuurimalliin. Malli on hajautettujen järjestelmien rakentamiseen tarkoitettu malli, jossa työ jaetaan palveluntarjoajan, eli palvelimen, ja palvelun käyttäjän, eli asiakkaan, välillä. Tämän luvun aliluvuissa käydään läpi tämän työn kannalta keskeiset web-ohjelmistojen toteutustekniikat.

2.4.1 HTTP(S)

Hypertekstin siirtoprotokolla (Hypertext transfer protocol) eli **HTTP** on protokolla, jonka päälle web-ohjelmistojen tiedonsiirto useimmiten perustuu. HTTP on tilaton protokolla, joka koostuu asiakaspään pyynnöistä ja palvelinpään vastauksista. Normaali HTTP-kommunikaatio tapahtuu siten, että asiakas lähettää palvelimelle pyynnön johonkin palvelimen tuntemaan resurssiin. Tähän palvelin vastaa siihen sopivalla vasteella, joka voi olla HTML-merkkausta tai yleistä binääridataa. Itsessään HTTP:n tiedonsiirto tapahtuu **TCP**-protokollalla. [4]

HTTP-viestit noudattavat niille määriteltyjä raameja, jotka koostuvat otsaketiedoista ja viestirungosta. Otsaketiedoissa määritellään pyynnön tiedot ja voidaan asettaa vastaukselle vaatimuksia muunmuassa siitä, mitä dataa asiakas voi vastaanottaa, ja mitä enkoodausta tekstipohjainen vastaus saa käyttää.

HTTP-pyyntöihin liittyy aina metodi. Metodin avulla palvelin tietää, mitä käyttäjä haluaa pyynnössä kohdistettuun resurssiin tehdä. HTTP-pyynnöt aina alkavat käytetyllä metodilla. Yleisimmin käytetyt metodit ovat GET- ja POST-metodit. GET metodia käytetään vain pyynnössä kohdistetun resurssin noutoon, eikä sillä pitäisi olla muuta vaikutusta palvelimeen. POST-metodia käytetään välittämään esimerkiksi verkkosivulla täytettyä lomaketietoa palvelimelle. On olemassa myös lukuisia muita HTTP-pyynnön metodeita, mutta verkkoselaimet mahdollistavat lomakkeissa useimmiten vain GET- ja POST-metodit.

HTTP-vastauksiin liittyvät vastauskoodit. Vastauskoodit ovat numeromuotoisia esityksiä siitä, miten pyyntö on käsitelty. Vastauskoodit on jaoteltu vastauskoodin

aloittavan numeron perusteella erilaisiin ryhmiin, jotka kuvataan taulukossa 2.1.

Ryhmä	Selitys
1XX	Informatiivinen viesti
2XX	Onnistunut pyynnön käsittely
3XX	Pyydetty sisältö löytyy muualta
4XX	Virhe asiakkaan toiminnassa
5XX	Virhe palvelinpään toiminnassa

Taulukko 2.1 *HTTP-vastauskoodien ryhmät*

HTTP on tilaton protokolla, joka tarkoittaa sitä, että pyynnöissä ja vastauksissa ei säilötä palvelimen tai asiakkaan tilatietoja. Mikäli järjestelmässä tarvitsee säilöä tila pyyntöjen välillä, kuten verkkokaupassa käytettävän ostoskorin sisältö, voidaan tilatiedot tallentaa istuntoon. Istunto on tapa tallentaa käyttäjän tila siten, että järjestelmä voi sen tarvittaessa lukea. Istunnot voidaan toteuttaa asiakaspäässä evästeillä, joihin voidaan tallentaa tekstipohjaista dataa. Vastaavasti palvelimella voidaan istunto toteuttaa palvelimen muistiin tai palvelimen tietokantaan luotavilla istunnoilla. Kummassakin tekniikassa on omat huonot puolensa. Asiakaspään evästeitä on helppo muokata, ja täten muokata istunnon tilaa. Palvelinpäässä taasen tilan sitominen tiettyyn asiakkaaseen on epävarmaa, sillä osoitteenmuutostekniikoiden johdosta ei asiakkaasta voida olla pelkän IP-osoitteen perusteella varmoja.

Protokollasta on olemassa myös salattu versio HTTPS, joka on HTTP yhdistettynä SSL/TLS-protokollaan. HTTPS on muuten identtinen HTTP:n kanssa, mutta siinä HTTP-pyynnöt ja vastaukset ovat salattuja SSL/TLS-protokollan avulla.

2.4.2 HTML & DOM

HTML on XML-pohjainen merkkauskieli. HTML määrittää dokumentin rakenteen ja täten osittain sen, miten selaimet dokumentin piirtävät. HTML myös tarjoaa dokumentin oliomallin **DOM**:in (Document Object Model). DOM on rajapinta, joka mahdollistaa dokumenttien sisällön, rakenteen ja tyylien dynaamisen käsittelyn ja päivityksen. Dokumentin DOM-rajapinnan tuottaminen on verkkoselaimen vastuulla. Rajapintaan liittyy myös käsite DOM-puu, joka on puumainen rakenne dokumentin sisällöstä. Puu-rakenteen johdosta dokumenttia voidaan navigoida tehokkaasti solmusta toiseen, ja näin luoda monimutkaisia dokumenttia muokkaavia sovelluksia. [18]

HTML-merkkaus on rakenteista tekstiä, joka perustuu elementteihin. Elementti koostuu elementin tunnisteesta (**Tag**), mahdollisista elementin attribuuteista ja elementin sisällöstä. Elementin tunniste kertoo verkkoselaimelle, millaisesta elementistä on kyse. Näin verkkoselain voi reagoida siihen näyttämällä käyttäjälle halutun kaltaisen elementin. Vaikka sisältö itsessään on hyvin yksinkertaista käyttäjälle esitettävää dataa, tulee monimutkaisuus erilaisista lomake-elementeistä, joissa on itsessään omaa toiminnallisuutta. Elementin attribuutit asettavat ominaisuuksia elementin sisällölle. Esimerkiksi hyperlinkit toisiin sivuihin tehdään asettamalla hyperlinkin elementtiin href-attribuutti. Elementin sisältö voi olla normaalia tekstiä tai muita elementtejä, joilla voidaan toteuttaa dokumentille syviä puumaisia hierarkkioita. Näistä hierarkkisista elementeistä muodostuu dokumentin rakenne.

2.4.3 HTML5

HTML-merkkauskielen uusin versio on HTML5. Vaikka HTML on käytännössä pelkkä merkkauskieli, on HTML5 terminä on laajentunut käsittämään paljon muutakin kuin pelkkää HTML-merkkausta, ja siten se on yleistynyt synonyymiksi moderneille web-tekniikoille. HTML5-version mukana esiteltiin paljon uusia web-sovelluksien toteuttamisessa käytettäviä ominaisuuksia. Ominaisuuksia ovat uudet HTML-elementit ja ohjelmointirajapinnat. Vanhoja ja harvoin käytettyjä HTML-elementtejä poistettiin HTML5-versiosta.

Uudet HTML-elementit jakautuvat semanttisiin-, graafisiin- ja multimedia-elementteihin. Semanttisilla elementeillä pyritään tarkentamaan dokumenttien rakennetta yleisesti käytettyjen web-rakenteiden mukaisilla tunnisteilla. Graafiset elementit mahdollistavat grafiikan näyttämisen ja tuottamisen ohjelmallisesti joko vektorigrafiikkana **SVG**-elementissä tai rasterina **canvas**-elementissä. Multimediaelementit mahdollistavat videoiden ja äänileikkeiden esittämisen käyttäjälle.

Uusia ohjelmointirajapintoja esiteltiin lukuisia. Tässä työssä käytetään Drag and Drop -rajapintaa. Drag and Drop -rajapinnalla on mahdollista toteuttaa web-sovelluksen käyttöliittymään raahaus ja tiputus interaktio. [15]

Rajapintojen määrittelyn johdosta HTML5-tekniikoilla toteutettu ohjelmisto toimii kaikissa HTML5-yhteensopivissa laitteissa. Täten sama ohjelmisto toimii esimerkiksi älypuhelimilla, tableteilla sekä perinteisillä työasemilla. [20]

2.4.4 CSS

CSS (Cascading Style Sheets) on tyylikieli, jolla voidaan määritellä tyylejä XML-tyyppisille dokumenteille. Kieli irrottaa dokumentin ulkoasun dokumentin sisällöstä siten, että samalla dokumentilla voidaan toteuttaa ulkoasuja eri käyttökonteksteihin, kuten isoille ruuduille tai mobiililaitteille. [19]

CSS-kielen mukainen tyylimääre koostuu valitsimesta ja avain-arvo-pareista, missä valitsin tarkoittaa sääntöä noudattavia elementtejä ja parit ovat varsinaisia sääntöjä. CSS-valitsimilla on mahdollista valita dokumentin elementtejä sen DOM-puusta. Koska DOM-puu on navigoitava tietorakenne, on mahdollista muodostaa puu-rakennetta hyväksikäyttäviä valitsimia. Valitsimilla voidaan valita esimerkiksi kaikki tietyt elementit jonkin elementin sisältä.

Itse säännöt toteutetaan avain-arvo-pareilla. Pareissa avaimena toimii jokin valitulle elementille määritelty ominaisuus, kuten taustaväri tai fontti. Ominaisuuden arvoa muutetaan parissa määritellyllä arvolla siten, että määritelty arvo korvaa alkuperäisen arvon. Tämä myös mahdollistaa tyylimääreiden ylikuormittamisen. Ylikuormittaminen tarkoittaa sitä, että tilanteissa, joissa yhdelle elementille on asetettu jollekin ominaisuudelle useampia eri arvoja, jää viimeksi määritelty arvo voimaan.

2.5 Backbone

Backbone on JavaScript-kirjasto [1], jonka tarkoitus on tarjota kehittälle valmiit MVC-arkkitehtuurimallin mukaiset Malli- ja Näkymä-oliot. Kirjastoa käytetään työssä toteutettujen sovellusten pohjana.

Backbone eroaa tavallisesta MVC-arkkitehtuurista siten, että siinä ei ole toteutettuna Ohjain-komponenteille valmista oliota. Useimmiten toiminnallisuus Backbone-sovelluksissa totetutetaankin osaksi näkymää.

Backbone tarjoaa hyvin määritellyn rajapinnan mallien muokkaukseen. Tämän lisäksi malleissa on toteutettuna oletustoiminnallisuus mallin muokkausten välittämiseen palvelimelle. Lisäksi mallit voivat lukea palvelimelta uudet tietosisältönsä. Backbone-kirjasto myös tarjoaa näkymien hallinnalle rajapinnan. Rajapinnan avulla näkymä voidaan asettaa kuuntelemaan käyttäjän syötteitä, jonka avulla voidaan itse näkymiin toteuttaa sovelluslogiikkaa.

Malli- ja Näkymä-olion lisäksi kirjasto tarjoaa Events-olion, joka on valmis toteutus Tarkkailija-suunnittelumallista. Tarkkailija-suunnittelumalli käsitellään tarkemmin kohdassa 2.6.7.

2.6 Suunnittelu- ja arkkitehtuurimallit

Suunnittelumalli on yleinen hyväksi todettu tapa ratkaista jokin yleisesti havaittu ongelma ohjelmistojen toteutuksessa. Suunnittelumalleja voidaan käyttää helpottamaan sekä ohjelmistojen suunnittelua että ohjelmistotyön ohessa tapahtuvaa kommunikaatiota. Kommunikaatio helpottuu, koska toimijat voivat vaihtaa runsaasti tietoa sovelluksissa käytetyistä ratkaisuista käyttämällä suunnittelumallien nimiä.

Suunnittelumallit voidaan jakaa kolmeen ryhmään: luomismallit, rakennemallit sekä käyttäytymismallit. Luomismallit ovat erilaisia tapoja tuottaa ohjelman toiminnan vaatimia olioita. Rakennemallit ovat erilaisia tapoja tuottaa olioiden välisiä yhteyksiä ja suhteita sekä niiden yhdistelyä suuremmiksi kokonaisuuksiksi. Käyttäytymismallit kuvaavat olioiden välistä kommunikaatiota.

Tässä kohdassa käydään läpi työn ymmärtämisen kannalta tärkeimmät arkkitehtuuri- ja suunnittelumallit.

2.6.1 Entiteetti-komponentti-järjestelmä -arkkitehtuurimalli

Entiteetti-komponentti-järjestelmä eli **ECS** (Entity-Component-System) on arkkitehtuurimalli, joka oliot koostetaan sijaan haluttuja toiminnallisuuksia toteuttavista komponenteista [2]. Entiteetti-komponentti-järjestelmää käytetään yleisesti peliohjelmoinnissa, mutta samaa arkkitehtuurimallia voidaan käyttää myös muissa sovelluksissa, kuten lääketieteellisissä kuvantamismenetelmissä [17].

Entiteetti-komponentti-järjestelmä koostuu taulukossa 2.2 esitellyistä osista. Työssä käytetään mukailtua entiteetti-komponentti-järjestelmä -mallia ja arkkitehtuurimallin termit sidotaan konkreettisesti toteutettavaan arkkitehtuuriin vasta arkkitehtuurin suunnittelun osuudessa.

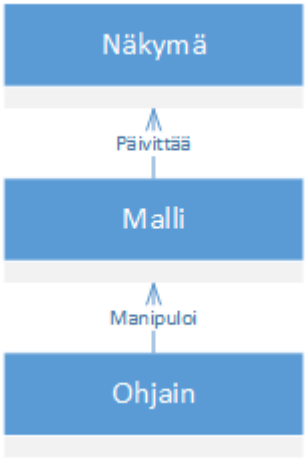
Osa	Selitys
Entiteetti	Yleinen konkreettinen olio, jolle halutaan toiminnallisuutta.
Komponentti	Entiteettiin liitettävä olion käytösaspekti, jolla määritellään olion käyttäytymistä tietyissä tapauksissa.
Järjestelmä	Järjestelmä on entiteetti-komponenttia ympäröivä osa, joka toteuttaa toiminnot komponentin omaaville entiteeteille.

Taulukko 2.2 *Entiteetti-komponentti-järjestelmä*

Konkreettinen esimerkki edellämainituista voidaan johtaa peleissä yleisesti käytetyistä ratkaisuista. Pelaaja-olio on entiteetti, jolle voidaan asettaa komponentteina näppäimistön ja hiiren avulla tapahtuvan liikkumisen mahdollistava komponentti. Järjestelmä on pelimoottori, jonka tehtävänä on välittää jokaisella peliä ajavan silmukan kierroksella pelaajan syötteet pelaajalle, joka komponentin avustuksella päivittää uuden sijaintinsa pelimaailmassa.

2.6.2 MVC-arkkitehtuurimalli

Malli-näkymä-ohjain (**MVC**, model-view-controller) arkkitehtuurimallin tarkoitus on erottaa sovelluksen sisältämä data, sen näkymä ja sovelluslogiikka erillisiksi komponenteiksi. Sovellus on siis jaettu kolmeen erityyppisiin osiin: Malli, Näkymä ja Ohjain. MVC-arkkitehtuurimalli on kuvattu kuvassa 2.5. [11]



Kuva 2.5 *MVC-arkkitehtuurimalli*

Malli edustaa sovellusdataa. Tämä voi tarkoittaa esimerkiksi tietokannassa säilytettävää riviä tai sovelluksen loogista tilaa. Mallin tehtävänä on hallita sen sisältämää dataa ja tarjota dataa manipuloivia operaatioita ohjainten käytettäviksi.

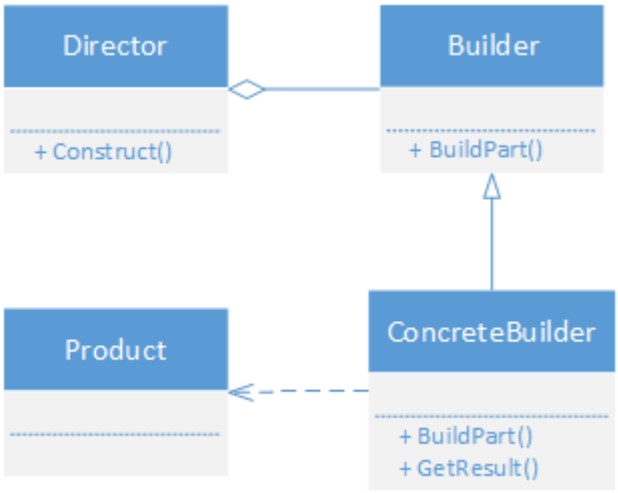
Näkymä edustaa jotain osaa käyttöliittymästä. Näkymä käytännössä on mallin sisältämän datan esitysmuoto. Näkymän vastuulla on hallita näytönpäivitystä, esittää mallin tiedot käyttäjälle ja päivittää tilaansa mallin muutoksien mukaan.

Ohjaimen tarkoitus on käsitellä käyttäjän syötteitä ja muuntaa ne loogisiksi sovellustoiminnoiksi. Ohjaimen kautta käyttäjä voi käsitellä mallidataa, ja täten muuttaa mallin tilaa.

MVC-arkkitehtuurimallista on olemassa useita mukaelmia. Tässä käytetään mukailtua MVC-arkkitehtuurimallia, jossa ohjain-komponentit toteutetaan entiteetti-komponentti-järjestelmä -arkkitehtuurimallin mukaisilla komponenteilla.

2.6.3 Rakentaja-suunnittelumalli

Rakentaja-suunnittelumalli (**Builder pattern**) on luomismalli, jossa kompleksi olio rakennetaan ulkoisen rakentaja-olion avulla. Suunnittelumalli koostuu rakentaja-rajapinnasta, sen toteuttavasta konkreettisesta rakentaja-oliosta ja olioiden rakennusta ohjaavasta ohjaaja-oliosta. Suunnittelumalli on kuvattu kuvassa 2.6.



Kuva 2.6 *Rakentaja-suunnittelumalli*

Kuvassa on esitetty suunnittelumallissa keskeisissä rooleissa olevat oliot. Ohjaaja (**Director**) on olio, joka käynnistää rakennusprosessin. Ohjaaja siten on asiakas, joka käyttää rakentajaa tuottamaan komplekseja olioita omiin tarpeisiinsa. Rakentaja (Builder) on abstrakti olio, joka määrittelee kompleksien olioiden rakennukseen käytettävän julkisen rajapinnan. Konkreettinen rakentaja (**ConcreteBuilder**) on varsinainen Rakentaja-rajapinnan toteuttava olio. Konkreettisen rakentajan tarkoitus on määritellä sen tuottamien kompleksien olioiden rakentamisessa käytettävä rakennusprosessi. Tuote (**Product**) on rakennusprosessissa tuotettu kompleksi olio.

Suunnittelumallin käytössä keskeinen käyttötarkoitus on sovelluksen kompleksien olioiden rakentamisen yksinkertaistaminen kehittäjälle. Kompleksit oliot koostuvat usein monista muista olioista, joiden rakentaminen pitää ottaa kehitystyössä huomioon. Rakentaja-suunnittelumallissa erotetaan olion luontilogiikka ohjelmiston muusta rakenteesta. Täten kompleksien olioiden luomislogiikka muodostaa oman

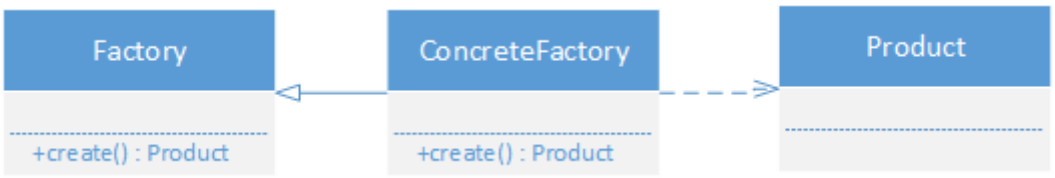
moduulinsa, jota voi uudelleenkäyttää muissa samaa oliota vaativissa ohjelmistoissa.

Rakentaja-suunnittelumalli saavuttaa tavoitteensa määrittelemällä yhteisen rajapinnan olioiden tekemiseen. Varsinainen konkreettinen rakennustyö tapahtuu rajapinnan toteuttavissa olioissa. Rakentaja myös tarjoaa rajapinnan, jota käyttämällä saadaan konkreettinen rakennettu olio.

Suunnittelumalli parantaa ohjelman modulaarisuutta kapseloimalla kompleksien olioiden rakennuksen sisäänsä. Toisin kuin muut luomismallit, rakentaja tuottaa olioita vaiheissa. Vaiheet määrittelevät tuotettavan olion lopullisen muodon. Vaiheittainen olioiden rakentaminen antaa kehittäjälle huomattavasti suuremman hallinnan olioiden rakennusprosesseista. [6]

2.6.4 Tehdasmetodi-suunnittelumalli

Tehdasmetodi-suunnittelumalli (**Factory Method pattern**), joka on esitetty kuvassa 2.7, on luomismalli, joka kätkee jonkin rajapinnan toteuttavien olioiden luontilogiikan yhteen luomisfunktioon. Tehdasmetodilla voidaan ajonaikaisesti valita luotavan olion tyyppi ja palauttaa rajapinnalla ehostettu olio. Näin tehdasmetodin kutsuja ei tiedä luodun olion tarkkaa tyyppiä, mutta voi käyttää sitä määritellyn rajapinnan läpi. Myös tilanteissa, joissa luokan rakentajaa ei ole suositeltavaa kuormittaa, voidaan käyttää tehdasmetodia määrittelemään tarkoitukseen sopeutettu olion luontimetodi. [6]



Kuva 2.7 *Tehdasmetodi-suunnittelumalli*

Kuvassa Tehdas (**Factory**) on olio, joka määrittelee konkreeteille tehdasolioille (**ConcreteFactory**) rajapinnan. Rajapintaa kutsumalla konkreetti tehdas palauttaa tuotteen (**Product**).

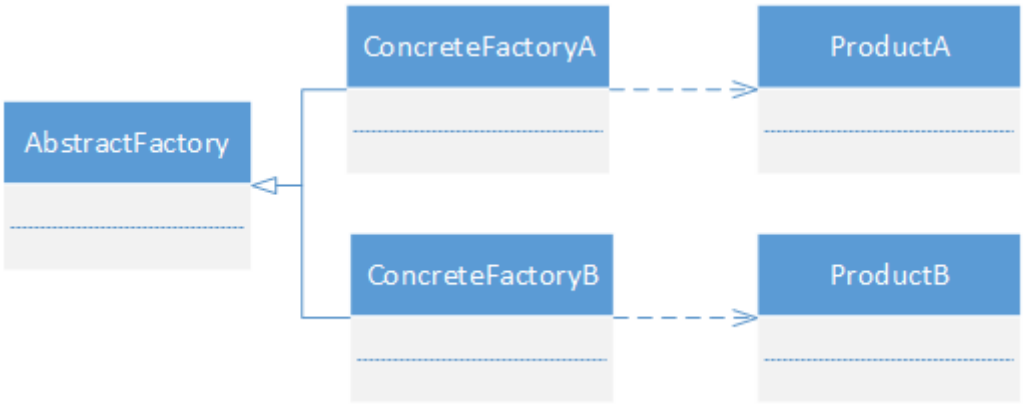
Suunnittelumallissa keskeisin ratkaistava ongelma on sovelluksen tarvitsemien olioiden luonnin irroittaminen itse olion luovasta logiikasta. Tämän ongelman suunnittelumalli ratkaisee siten, että olioiden luonnille määritellään vain rajapinta. Olioiden

luonnista vastuussa ovat rajapinnan toteuttavat oliot. Tämän lisäksi itse tuotetuille olioille suunnittelumallissa määritellään yleinen rajapinta.

Tehdasmetodi-suunnittelumallia on hyvä käyttää esimerkiksi silloin, kun oliota tarvitsevan ohjelmaosan on turha tietää konkreetin olion riippuvuuksista. Nämä riippuvuudet voidaankin kapseloida osaksi konkreetin olion luovaa tehdasta. Näin tehdas toimii täysin itsenäisenä ohjelmaosana, joka lisää ohjelmakoodin uudelleenkäyttöä muissa käyttökohteissa.

2.6.5 Abstrakti tehdas-suunnittelumalli

Abstrakti tehdas -suunnittelumalli (**Abstract Factory pattern**) on luomismalli, jossa konkreettisten olioiden luominen tapahtuu yhden yhteisen rajapinnan läpi. Abstrakti tehdas kapseloi itseenä joukon konkreettisia tehdasolioita, joiden vastuulla on konkreettisten olioiden luominen. Abstraktin tehtaan vastuulla on päätellä käyttäjän syötteestä, mitä tehdasta kunkin olion luomiseen käytetään. Suunnittelumalli on kuvattu kuvassa 2.8.



Kuva 2.8 *Abstrakti tehdas-suunnittelumalli*

Kuvassa suunnittelumallin keskeisissä rooleissa on abstrakti tehdas, joka määrittelee rajapinnan tuotettavien olioiden tuottamiseen. Abstrakti tehdas ajonaikaisesti liitetään konkreettiin tehtaaseen. Konkreetti tehdas on vastuussa tietyn tuotteen tuottamisesta.

Suunnittelumalli toimii periaatteiltaan hyvin samankaltaisesti Tehdasmetodi-suunnittelumallin kanssa. Abstrakti tehdas usein toteutetaankin siten, että sen sisältämät konkreetit tehtaat toteuttavat Tehdasmetodi-suunnittelumallin. Abstrakti tehdas ratkaisee käytännössä samoja ongelmia kuin Tehdasmetodi, mutta soveltuu huomattavasti paremmin tilanteisiin, missä tuotetaan samankaltaisia, mutta erilaisia

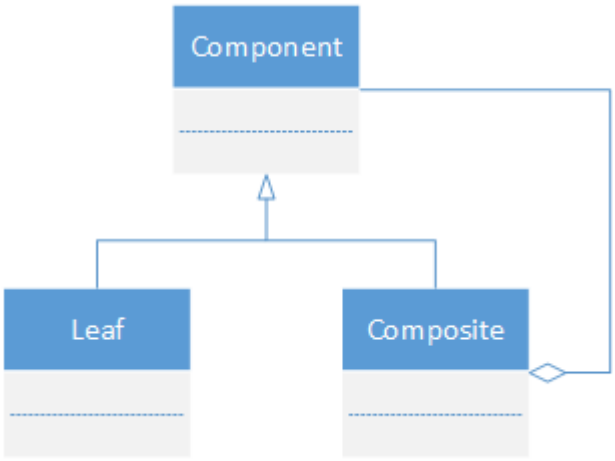
tehtaita vaativia olioita. Näitä tuotettuja olioita voidaankin kutsua tuoteperheeksi. [6]

Koska suunnittelumallissa usein konkreettiset tehtaat toteutetaan Tehdasmetodi-suunnittelumallin mukaisesti, suunnittelumalli mahdollistaa samankaltaisen ohjelmakoodin uudelleenkäytön kuin Tehdasmetodi-suunnittelumallissa koskien tuoteperheitä.

2.6.6 Kooste-suunnittelumalli

Kooste-suunnittelumalli (**Composite pattern**) on rakennemalli, jossa olio voi koostua saman rajapinnan jakavista olioista. Oliot voivat siten koostua hyvinkin syvistä rekursiivisista rakenteista. Suunnittelumalli on kuvattu kuvassa 2.9.

Koostetta käytetään hyvin yleisesti graafisten käyttöliittymien ohjelmoinnissa, jossa käyttöliittymäelementit usein toteutetaan koosteolioina. Esimerkiksi tavallinen dialogi käyttäjälle voidaan käsittää koosteoliona, jossa juurena toimii dialogin ikkuna, ja loput käyttöliittymäelementit koostetaan osaksi ikkunaa. [6]



Kuva 2.9 *Kooste-suunnittelumalli*

Kuvan keskeisissä rooleissa ovat komponentti (**Component**), joka määrittelee koosteessa käytettävien olioiden rajapinnan. Lehdet (**Leaf**) ovat olioita, joilla ei ole lapsiolioita. Lehdet ovat siten suunnittelumallin hierarkkiatasoissa alimpina. Komposiitit (**Composite**) ovat olioita, jotka voivat koostua useammista komponenteista. Koostamalla komposiitteja komponenteista oliohierarkkiasta saadaan muodostettua puumainen rakenne.

Keskeinen ongelma, jonka suunnittelumalli ratkaisee, on erillisten rajapintojen määrittelyn puumaisessa rakenteessa oleville olioille. Suunnittelumallin ansiosta sekä lapsellisia olioita että lapsettomia olioita voidaan käsitellä saman rajapinnan läpi ottamatta kantaa siihen, sijaitseeko käsiteltävän olion hierarkkiassa lapsia.

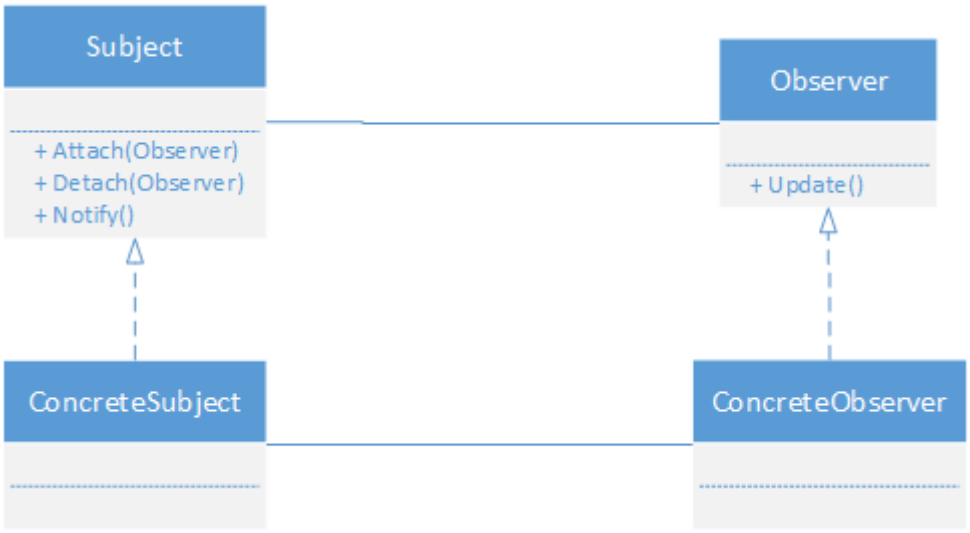
Esimerkkinä Kooste-suunnittelumallia käytetään usein graafisien käyttöliittymien ohjelmoinnissa, joissa käyttöliittymät koostetaan usein osaksi jotain isompaa elementtiä. Esimerkiksi ohjelman ikkuna koostuu käyttäjälle näytettävästä datasta sekä vierityspalkeista, ja käyttäjälle näytettävä data koostuu taulukosta ja toimintoja suorittavista painikkeista.

2.6.7 Tarkkailija-suunnittelumalli

Tarkkailija-suunnittelumalli (**Observer pattern**) on käyttäytymismalli, jossa ohjelmistojen komponenttien välinen kommunikaatio tapahtuu löyhästi sidotun tarkkailijan läpi. Suunnittelumalli on kuvattu kuvassa 2.10.

Suunnittelumallissa tarkkailija (**Observer**) kuuntelee sille rekisteröityneiden kohteiden (**Subject**) muutoksia. Yhden kohteen muutos aiheuttaa sen, että tarkkailija ilmoittaa muutoksista kaikille muille kohteille, joille sama tarkkailija on rekisteröity. [6]

Suunnittelumallin keskeisissä rooleissa ovat tarkkailija, joka määrittelee rajapinnan, jota käyttämällä tarkkailijalle voidaan rekisteröidä tai poistaa tarkkailtavia kohteita. Rajapinnan toteuttava olio on konkreetti tarkkailija, jonka tehtävänä on toteuttaa konkreettinen tarkkailija-olio. Kohde on rajapinta, joka määritellään konkreeteille kohteille. Kommunikaatio tarkkailijan ja kohteiden välillä tapahtuu määritellyn rajapinnan kautta.



Kuva 2.10 *Tarkkailija-suunnittelumalli*

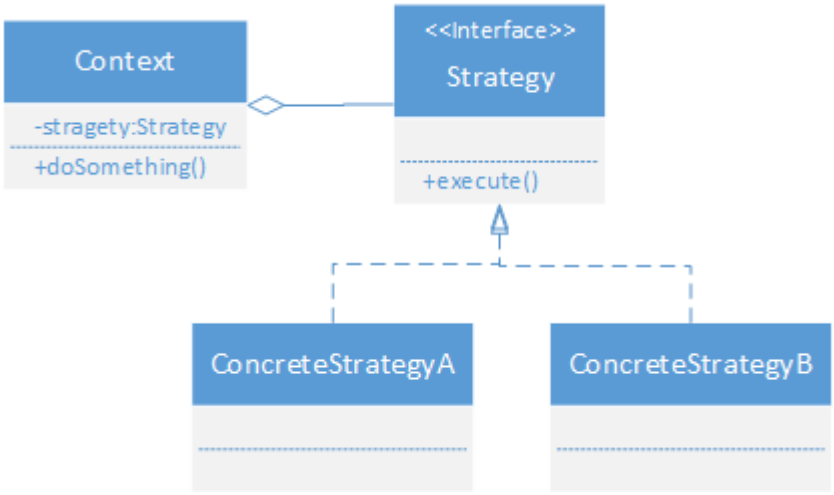
Keskeisin ongelma, jonka suunnittelumalli ratkaisee, on halu välittää tietoa olioiden tilanmuutoksista toisille olioille. Suunnittelumallilla on mahdollista irrottaa oliot toisistaan hoitamalla kommunikaatio olioiden välillä erillisen tarkkailijan avulla. Tämä poistaa tarpeen sitoa olioita toisiinsa vahvasti edellämainitun tilanteen toimintojen mahdollistamiseksi. [6]

Malli ja mallin näkymän päivitys esimerkiksi voidaan toteuttaa Tarkkailija-suunnittelumallin avulla. Mallin näkymä on tarkkailija, joka tarkkailee sen esittämää mallia, joka on kohde. Mallin muuttuessa ilmoitetaan muutoksesta näkymälle, joka tietää päivittää käyttäjälle näytettävät tiedot ilmoituksen perusteella. Näin käyttäjälle voidaan esittää malliin tehdyt muutokset välittömästi niiden tekemisen jälkeen.

2.6.8 Strategia-suunnittelumalli

Strategia-suunnittelumalli (**Strategy pattern**) on käytösmalli, jossa jonkin toiminnon suorittava algoritmi valitaan ajonaikaisesti. Suunnittelumalli on kuvattu kuvassa 2.11.

Suunnittelumallissa olioissa, jotka poikkeavat toisistaan vain käyttäytymisen osalta, voidaan toteuttaa toiminnallisuuksia erillisten strategia-olioiden avulla. Strategia-olioiden tarkoitus on pitää rajapinta olioiden välillä yhtenäisenä, vaikka varsinainen toteutusalgoritmi muuttuisikin.



Kuva 2.11 *Strategia-suunnittelumalli*

Suunnittelumallin keskeisissä rooleissa ovat konteksti (**Context**), joka on strategian (**Strategy**) avulla jonkin toiminnon toteuttava olio. Strategia on rajapinta, joka määritellään konkreettisille strategioille. Konkreettiset strategiat (**ConcreteStrategy**) ovat olioita, joiden vastuulla on suorittaa vain niille määritelty algoritmi.

Suunnittelumallin keskeinen ongelma on toiminnallisuuksien eristäminen muusta ohjelmasta. Tämä ratkaistaan siten, että toiminnallisuudet kapseloidaan omiksi olioikseen tarkoituksenmukaisiin toiminnallisuusperheisiin. Kapseloinnin ja yhtenäisen rajapinnan avulla toiminnallisuus toteutetaan erillään muusta ohjelmasta siten, että toiminnallisuus voidaan helposti uudelleenkäyttää muissa vastaavaa toiminnallisuutta käyttävissä kohteissa. [6]

Suunnittelumallilla voidaan esimerkiksi toteuttaa verkon yli dataa lähettävälle oliolle eri protokollien toteutukset. Esimerkiksi UDP- ja TCP-protokollat toteuttavat datan siirtoa verkossa eri metodeilla, jotka voidaan hyvin toteuttaa suunnittelumallin mukaisesti samankaltaisen rajapinnan takana. Dataa verkon yli lähettävä olio voidaan siis Strategia-suunnittelumallin avulla erottaa varsinaisesta datan lähetyksestä vastaavasta algoritmista. Näin oliolle voidaan toteuttaa useita tiedonsiirtometodeja ilman, että olion sisäistä toteutusta tarvitsee muuttaa.

2.7 Toteutettavien sovellusten taustatiedot

Arkkitehtuurin suunnittelun lähtökohtana on muutama arkkitehtuurin avulla kehitettävä sovellus. Tässä kohdassa esitellään työn kannalta keskeiset lähtötiedot sovellusten tarkoituksen ymmärtämisen kannalta.

2.7.1 Kanban

Kanban on toiminnanajoitusjärjstelmä, jossa keskeisessä asemassa ovat Kanban-kortit. Korttien tarkoitus on toimia viestimekanismeina, jotka kertovat hyödykkeiden kulutuksesta. Kanban-kortit välittävät viestejä siitä, mitä resursseja on kulutettu, ja mitä on näin ollen on tuotettava tai tilattava lisää. Kanban on hyvin sovellettava järjestelmä, josta on sovelluksia useisiin tuotantoprosesseihin.

Työssä keskitytään Kanban-järjestelmään, jota käytetään ohjelmistotuotannon prosesseissa. Ohjelmistotuotannossa Kanban-järjestelmään lisätään korttien rinnalle termi Kanban-taulu. Taulun tarkoitus on hahmottaa korttien senhetkistä tilaa siten, että taulu on jaoteltu tiloja vastaaviin alueisiin. Laittamalla kortti tietyn alueen sisään asetetaan kortti alueen osoittamaan tilaan. Käyttämällä edellämainittua ehostettua Kanban-järjestelmää, toimii Kanban-taulu visuaalisena menetelmänä hahmottaa prosessien suoritusta.

2.7.2 Katselmus

Katselmus on tapahtuma, jonka tarkoitus on todeta projektin jonkin vaiheen päättyminen. Katselmuksia on erityyppisiä, mutta yhteinen tekijä kaikille katselmuksille on projektin arviointi jostain näkökulmasta.

Katselmuksen yksi keskeisistä tarkoituksista on laadunvarmistus. Katselmuksia yleisesti käytetäänkin siihen, että tarkastetaan projektin etenemisen täyttävän yrityksen sisäisen laatujärjestelmän rajoitteet ja vaatimukset. Katselmukset toimivat myös projektin etenemisen indikaattoreina. [8]

3. TOTEUTETTAVAT SOVELLUKSET

Ennen varsinaista arkkitehtuurin suunnittelua on hyvä ymmärtää arkkitehtuurin käyttökohteiden yleisten elementtien ja vaatimusten perimmäiset syyt. Arkkitehtuuri luodaan tukemaan periaatteeltaan samankaltaisten ohjelmistojen kehtystyötä ja vahvistamaan niiden integroimista osaksi Realm-ympäristöä. Tässä kappaleessa kuvataan kolme esimerkkitapausta sovelluksista, joiden kaltaisia pitäisi suunnitellulla arkkitehtuurilla toteuttaa.

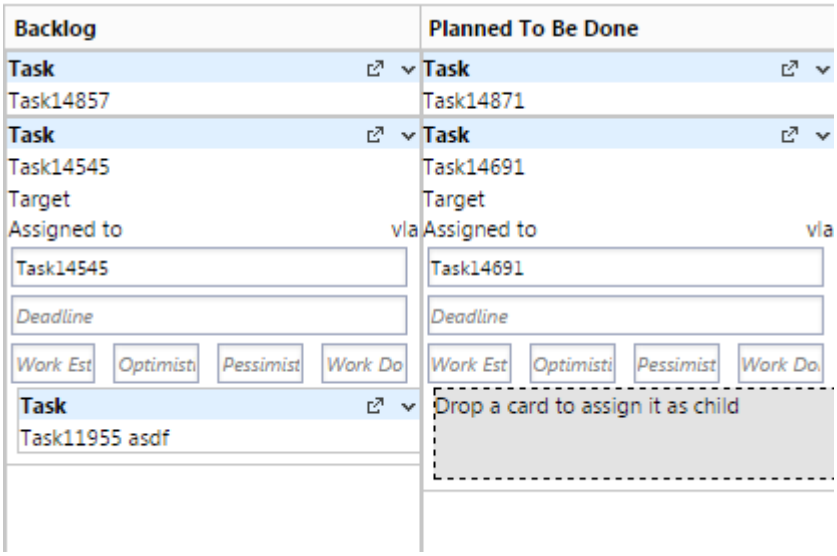
Kaikki työssä toteutetut sovellukset toteutetaan osaksi Realm-järjestelmää. Sovelluksia ei kuitenkaan haluta vahvasti sitoa osaksi Realm-järjestelmää ylläpidon ja jatkokehitystyön helpottamiseksi. Sovelluksien halutaan myös toimivan siten, että operaatiot eivät vaadi sovelluksen pohjasivun päivitystä.

3.1 Kanban-sovellus

Kanban-sovellus on ensisijaisesti tarkoitettu yrityksen sisäisen toiminnan ohjaukseen, priorisointiin ja yksittäisen työntekijän työkuorman seurantaan. Yksittäinen kortti luodussa sovelluksessa voi tarkoittaa jotain yksittäistä, yrityksen sisällä tapahtuvaa, itsenäistä työkokonaisuutta. Näitä työkokonaisuuksia voidaan koostaa isommiksi kokonaisuuksiksi, joilla voidaan mallintaa ja mitata suurempien työkokonaisuuksien edistymistä.

Sovellus koostuu kahdesta käyttäjälle näkyvästä näkymätyypistä, joista toinen on edellämainittu Kanban-kortti, ja toinen on kortin tilaa indikoiva alue Kanban-taulussa.

Sovelluksessa on myös vaatimus tuelle monen käyttäjän yhdenaikaiseen käyttöön. Tämä tarkoittaa sitä, että yhden käyttäjän muutokset kortteihin päivittyvät muille saman kortin näkeville henkilöille.



Kuva 3.1 *Ote Kanban-sovelluksen ulkoasusta*

Kuvassa 3.1 on hahmoteltu osa toteutettavasta Kanban-sovelluksen taulusta. Kuvassa on esitetty kaksi tilaa (Backlog ja Planned To Be Done) ja useita Kanban-kortteja. Kaikki tilat koostuvat Kanban-korteista, joita voidaan raahata tilojen välillä. Kortteja voidaan avata, jolloin niistä näytetään enemmän tietoa. Avatussa tilassa olevan kortin sisältöä voidaan muokata.

3.1.1 Kanban-kortin tiedot

Yksittäiseen Kanban-korttiin luodussa sovelluksessa halutaan määrittää tietoja, jotka toimivat työn ohjaamisen välineinä. Kortissa halutaan aina näyttää vähintään kortilla esitetyn tiedon tyyppi, sen tärkeysaste ja nimi. Myös linkki kortin edustamaan varsinaiseen työkokonaisuuteen tulee olla aina mukana. Näiden tietojen tulee aina näkyä korteista ilman käyttäjän tekemiä toimintoja.

Korttien loput keskeisiksi katsottavat tiedot tulee löytää kortista itsestään, mutta oletuksena kaikki tiedot on piilotettu. Piilotetut tiedot tulevat siis omaan kortin alielementtiinsä, jota ei oletuksena käyttäjälle näytetä. Näitä tietoja ovat muun muassa:

- Kortin deadline
- Kortin työkokonaisuuden suorittava henkilö
- Mihin työaiheeseen kortti liittyy

- Arviot työkokonaisuuden toteuttamiseen vaaditusta ajasta
- Työkokonaisuuden toteuttamiseen kulunut aika

Tiedon kategoriat ovat tämän työn toteutuksessa näytettävät tiedot. Näytettäviä tietokategorioita tulee pystyä lisäämään ja poistamaan vapaasti.

Yksittäisen kortin toteuttamat toiminnallisuudet ovat kortin tietojen muutos, tietojen piilottaminen, alikorttien luominen ja alikortin irrottaminen. Korttien tiedot tulee voida muuttaa itse korttiin sen ominaisuuksille määriteltyjen syötekenttien avulla. Alikorttien luominen luo uuden kortin kohdistetun kortin lapseksi. Luonnollisesti uuden kortin luominen luo samalla kortin edustaman työkokonaisuuden. Alikortti halutaan kyetä irroittamaan isäntäkortista siten, että kortti muunnetaan itsenäiseksi, toisista korteista riippumattomaksi kortiksi.

3.1.2 Tilat

Koska Kanban-sovelluksen toteutus on tarkoitettu sisäiseen työkokonaisuuksien ohjeistamiseen ja ajastamiseen, korteille on toteutettu tilat. Tila tarkoittaa työn vaihetta, jossa yksittäisen kortin edustama työkokonaisuus sillä hetkellä on. Koska Kanban-korttien vaatimuksena on korttien hierarkkisuus, voidaan sama rakenne yleistää myös tiloiksi. Täten tilatkin voidaan esittää korttipohjaisina elementteinä.

Tilat ovat Kanban-taulun alueita, jotka koostuvat tilan tiedoista sekä samassa tilassa olevista Kanban-korteista. Kanban-korttien tilasiirtymät toteutetaan raahaamalla kortti yhdestä tilakortista toiseen tilaan.

Tiloissa olevat kortit halutaan järjestää työkokonaisuuden prioriteetin perusteella. Tämän johdosta Kanban-kortteihin asetetaan metatietona ominaisuus, jota käytetään korttien keskinäisen järjestyksen selvittämiseen. Varsinainen konkreettinen järjestämisvastuu on kuitenkin tilakorteilla. Korttien järjestykseen tarvitaan vain yksi attribuutti, sillä kortti voi olla kerrallaan vain yhdessä tilassa.

3.1.3 Kanban-sovelluksen vaatimukset

Sovellukselle asetetut ei-toiminnalliset vaatimukset on esitelty taulukossa 3.1.

Tunniste	Nimi	Kuvaus
KB01	Korttipohjaisuus	Sovelluksen data esitetään kortteina.
KB02	Tilojen sisältö	Tilat koostuvat Kanban-korteista
KB03	Hierarkkiset työkokonaisuudet	Kanban-kortit voivat sisältää toisia Kanban-kortteja.
KB04	Korttien sisällön määrittäminen	Korttien sisällöt voidaan uudelleenmäärittää sovelluksen ohjelmakoodia muokkaamatta.
KB05	Monen käyttäjän yhdenaikainen käyttö	Korttien tietojen ja tilojen tulee päivittyä muille käyttäjille muutoksen jälkeen.

Taulukko 3.1 *Kanban-sovelluksen ei-toiminnalliset vaatimukset*

Vaatimus **KB01** määrittelee, että kaikki sovelluksen käyttäjälle näytettävä tieto sijaitsee korttipohjaisissa käyttöliittymäelementeissä. Kortteja ovat siis Kanban-kortit sekä korttien tilat.

Vaatimus **KB02** määrittelee tilakorttien sisällön. Tilakortit koostuvat hierarkkisesti Kanban-korteista. Varsinaista omaa, käyttäjälle näytettävää tietoa tilakorteissa on tilan nimi.

Vaatimus **KB03** määrittelee, että Kanban-kortit ovat hierarkkisia kortteja. Työkokonaisuudet voidaan usein pilkkoa joukoksi pienempiä itsenäisiä työtehtäviä. Kanban-korttien hierarkkisuudella voidaan toteuttaa työkokonaisuuden jakaminen pienempiin työtehtäviin.

Vaatimus **KB04** määrittää, että korttien käyttäjälle näytettävän datan tulee olla määriteltävissä sovelluksen ulkopuolella. Vaatimuksella halutaan mekanismi, jolla voidaan muokata käyttäjälle näytettävää osuutta koskematta itse ohjelmakoodiin.

Vaatimus **KB05** määrittää sen, että korttien tulee päivittyä näennäisesti reaaliajassa käyttäjien tekemien toimintojen johdosta. Käyttäjän tekemien muutosten myös muille näkyvään korttiin tulee päivittyä muiden käyttäjien vastaavaan korttiin päivittämättä verkkosivua.

Sovelluksen toiminnallisuudet on esiteltynä taulukossa 3.2. Arkkitehtuurin suunnittelussa tulee ottaa huomioon tuki esitettyjen vaatimusten toteuttamiselle.

Vaatimus **KB06** määrittelee, että Kanban-korttien käyttöliittymistä on löydyttävä toiminto, jolla voidaan piilottaa määritelty osa Kanban-kortissa näytettävistä tiedoista.

Tunniste	Nimi	Kuvaus
KB06	Kanban-korttien tietojen piilotus	Osa Kanban-kortin tiedoista voidaan piilottaa kortin käyttöliittymästä.
KB07	Tilojen piilotus	Tilat ja niiden sisällöt voidaan piilottaa käyttöliittymästä. Muut tilat täyttävät vapautuneen tilan Kanban-taulusta.
KB08	Tilasiirtymät	Kanban-kortin tila muutetaan raahaamalla kortti uutta tilaa vastaavaan tilaan. Mikäli raahaus perutaan tai uusi tila on sama kuin vanha, ei tilasiirtymään reagoida.
KB09	Korttien järjestys	Kortit tulee esittää niille määritellyn prioriteetin mukaan laskevassa järjestyksessä. Kortteja voidaan järjestää raahaamalla ja tiputtamalla niitä uuden prioriteetin mukaisiin kohtiin isäntäkorteissa.
KB10	Alikortiksi raahaaminen	Kanban-kortti voidaan sitoa toisen kortin alikortiksi raahaamalla se toisen kortin sisään. Raahatun kortin tilaksi muutetaan sama kuin isäntäkortin tila.
KB11	Alikorttien irroittaminen	Alikortit on kyettävä irroittamaan isäntäkorteista siten, että kortin ja sen isäntäkortin välinen kytkös puretaan.
KB12	Kanban-korttien luominen tilaan	Kanban-kortteja tulee pystyä luomaan tilaan suoraan sen käyttöliittymästä.
KB13	Kanban-korttien luominen alikortiksi	Kanban-kortteja tulee pystyä luomaan suoraan toisen Kanban-kortin alikortiksi.
KB14	Navigointi työkokonaisuuteen	Kanban-korteista tulee kyetä navigoimaan sitä edustavan työkokonaisuuden tarkempiin tietoihin.
KB15	Kanban-korttien tietojen muokkaaminen	Kanban-korteista tulee voida muokata kortin tietoja.

Taulukko 3.2 *Kanban-sovelluksen toiminnalliset vaatimukset*

Vaatimus **KB07** määrittelee, että tiloja voidaan vapaasti piilottaa näkyvistä. Muut tilat reagoivat piilotettuun tilaan siten, että kaikki näkyvät tilat laajenevat samassa suhteessa täyttämään piilotetun kortin jättämän tyhjän tilan sovelluksen käyttöliittymässä.

Vaatimus **KB08** määrittelee, että Kanban-korttien tilasiirtymät tapahtuvat raahaamalla kortti yhdestä tilasta toiseen. Tilasiirtymä päivitetään raahauksen lopussa palvelimelle, mikäli kortti siirretään uuteen tilaan. Raahaus täytyy voida peruuttaa käyttäjän toimesta.

Vaatimus **KB09** määrittelee, että kortissa olevia Kanban-kortteja on kyettävä järjestämään. Korttien järjestys indikoi korttien edustamien työkokonaisuuksien prioriteetteja. Kortit järjestetään oletuksena prioriteetin mukaan laskevassa järjestyksessä. Työkokonaisuuksien prioriteetteja on kyettävä raahaamalla ja tiputtamalla muuttamaan kortin tiputuksen jälkeistä visuaalista tilaa.

Vaatimus **KB10** määrittelee, että Kanban-korttien siirtäminen toisen Kanban-kortin osoittamaan työkokonaisuuteen. Kanban-korttien osoittamat työkokonaisuudet siirretään osaksi toista työkokonaisuutta raahaamalla sen Kanban-kortti uuden työkokonaisuuden Kanban-kortin sisään. Kun kortti siirretään alikortiksi, sen aiemmat viitteet ylempään työkokonaisuuteen puretaan ja sen tila asetetaan vastaamaan uuden isännän tilaa. Kuten vaatimuksessa **KB08**, myös tässä päivitetään tiedot vain, mikäli raahattavan Kanban-kortin isäntäkortti muuttuu. Lisäksi raahaus on pystyttävä perumaan.

Vaatimus **KB11** määrittelee Kanban-korttien käyttöliittymään toiminnon, jolla voidaan irroittaa yksi työkokonaisuus erilleen sen sisältävästä työkokonaisuudesta. Irrotus tarkoittaa sitä, että viitteet ylempään työkokonaisuuteen puretaan täysin, ja muokattu työkokonaisuus toimii itsenäisenä työkokonaisuutena.

Vaatimukset **KB12** ja **KB13** määrittelevät sen, että uusia Kanban-kortteja voidaan luoda olemassaoleviin kortteihin, joihin kuuluvat sekä tilat että Kanban-kortit. Operaatiot luovat välittömästi uuden, luomisoperaatiota vastaavan työkokonaisuuden, ja asettaa sen haluttuun kohteeseen.

Vaatimus **KB14** määrittelee, että Kanban-kortista on kyettävä navigoimaan verkkosivulle, jossa on tarkemmat tiedot kortin edustamasta työkokonaisuudesta. Tämä toteutetaan perinteisellä hyperlinkillä, eikä toiminnolle aseteta vaatimuksia asynkronisuuden suhteen.

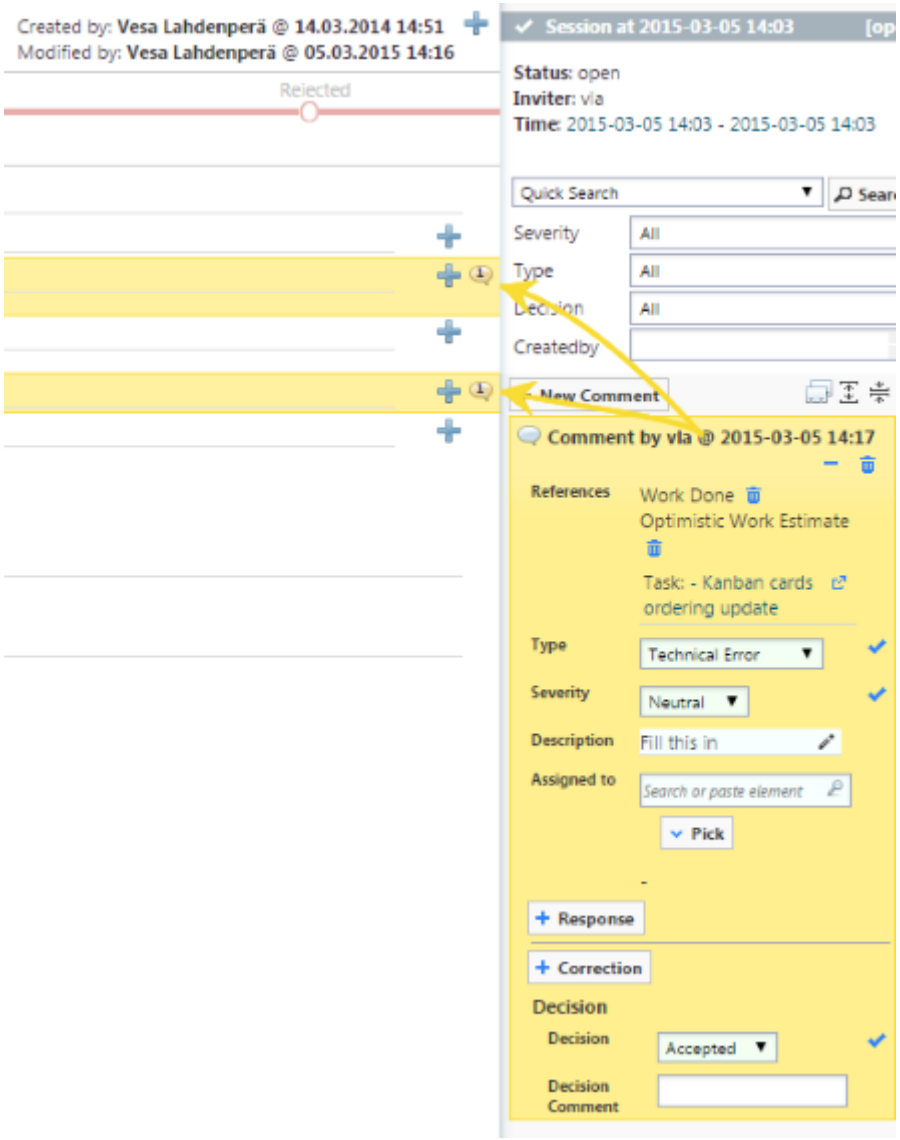
Vaatimus **KB15** määrittelee, että Kanban-korteista on kyettävä muokkaamaan kortin tietoja. Kortin tietojen tulee päivittyä Kanban-kortin metatietoihin tai kortin esittämän työkokonaisuuden tietoihin. Se, kummasta tiedot päivitetään, riippuu siitä, mitä päivitettävä kenttä edustaa.

3.2 Katselmointisovellus

Katselmointisovelluksella on tarkoitus mahdollistaa monen käyttäjän samanaikainen työkokonaisuuksien tekninen katselmus ja kommentointi. Sovelluksessa mahdollistetaan monen käyttäjän samanaikaiseen työkokonaisuuksien kommentointi ja niihin liittyvien graafisten viitteiden luonti.

Sovellus koostuu kommenttikorteista, korjauskorteista sekä korttien vuorovaikutuksesta Realm-järjestelmän kanssa. Kommenttikortin ideana on toimia käyttäjän kommenttina jostain käsiteltävän työkokonaisuuden osasta. Korjauskortit ovat kortteja, jotka yhdistetään kommentteihin. Korjausten ideana on osoittaa kommentin huomioista johtuvat korjaustyöt ja niiden eteneminen.

Kuvassa 3.2 on esitettynä katselmointisovelluksen ulkoasu. Kuvassa oikealla puolella oleva keltainen alue on katselmoinnissa luotu kommentti. Keltaiset nuolet osoittavat sen, minne kommentti kohdistuu.



Kuva 3.2 *Katselmointisovelluksen ulkoasu*

3.2.1 Kommenttikortit

Kommenttikortit esittävät käyttäjien kommentteja käsiteltävästä työkokonaisuudesta. Kommenttikortteja halutaan kyetä liittämään graafisilla viitteillä käsiteltävän työkokonaisuuden elementteihin.

Kommenttikortit ovat hierarkkisia, joten kommenttiin voidaan vastata uusilla kommenttikorteilla. Toistaiseksi halutaan vain yhden tason kommenttiketjuja, joten kommentin vastauksiin ei voi vastata. Kommenttiin vastaava kommenttikortti voidaan nimetä vastauskortiksi.

Kommenttikorteissa näytettävää dataa on:

- Kortin tekijä
- Kortin päivämäärä
- Kortin graafisten viitteiden lista
- Kommentin tyyppi
- Kommentin vakavuus
- Kortin kommentti
- Henkilö, jolle kommentti kohdistetaan.
- Korjauskortti
- Kommentin vastaukset

Näistä tiedoista aina kortissa näkyvissä ovat graafinen viite, päivämäärä ja tekijä. Muu data on kortissa oletuksena piilotettuna. Vastauskortissa näytettävää dataa on vain tekijä, päivämäärä ja vastauksen sisältö.

Graafiset viitteet ovat verkkosivun päälle piirrettäviä graafisia nuolia, jotka osoittavat kommentista kommentoitavaan kohteeseen. Viitteitä halutaan kyetä vetämään kommentista kohteeseen perinteisellä raahausmenetelmällä. Kommenttiin tai viitattavaan kohteeseen liittyvät viitteet tulee kyetä piilottamaan, ja viitteitä halutaan myös kyetä poistamaan.

Kommenttikortin muita toiminnallisuuksia ovat kortin tietojen poistaminen, tietojen muuttaminen, vastauksen luominen ja korjauskortin luominen. Vastauskortin toiminnallisuuksia ovat vastauksen poisto, vastauksen piilotus ja vastauksen editointi.

3.2.2 Korjauskortit

Korjauskortit esittävät kommenttien havainnosta johtuvia vaadittuja korjaustöitä. Korjauskortissa näytettävää dataa on

- Korjauksen tekijä
- Korjauksen päivämäärä
- Korjaustyön kuvaus
- Korjaustyön deadline
- Korjaustyön tila

Kaikki yllämainitut tiedot ovat aina näkyvissä, mikäli isäntäkortin piilotetut tiedot ovat näkyvissä.

Korjauskorttien toiminnallisuudet rajoittuvat puhtaasti kortin poistamiseen ja tietojen muokkaamiseen. Nämä toiminnallisuudet tulevat muille korteille toteutettujen toiminnallisuuksien yleistyksistä.

3.2.3 Katselmointisovelluksen vaatimukset

Sovellukselle asetetut ei-toiminnalliset vaatimukset on esitelty taulukossa 3.5.

Tunniste	Nimi	Kuvaus
RW01	Korttipohjaisuus	Sovelluksen data esitetään kortteina.
RW02	Kommenttikorttien sisältö	Kommenttikortit koostuvat mahdollisesta korjauskortista ja vastauskorteista.
RW03	Vastauskortit ja korjauskortit eivät hierarkisia	Vastauskortteihin ei haluta uusia vastauskortteja. Korttien vastaukset muodostavat vain yhden kommenttiketjun.
RW04	Korttien sisällön määrittäminen	Korttien sisällöt voidaan uudelleenmäärittää sovelluksen ohjelmakoodia muokkaamatta.
RW05	Monen käyttäjän yhdenaikainen käyttö	Korttien tietojen ja tilojen tulee päivittyä muille käyttäjille muutoksen jälkeen.

Taulukko 3.3 *Katselmointisovelluksen ei-toiminnalliset vaatimukset*

Vaatimus **RW01** määrittelee kaiken sovelluksen käyttäjälle näytettävän datan esitysmuodon. Kuten Kanban-sovelluksessa, myös tässä kaikki käyttäjälle näytettävä sovellukseen liittyvä tieto esitetään korteissa.

Vaatimus **RW02** määrittelee kommenttikorttien sisällön. Kommenttikortit ovat kortteja, joissa voidaan esittää kommenttiin liittyvä keskustelu erillisillä vastauskorteilla. Tämän lisäksi kommenttikorteissa voidaan esittää myös edellä määritetyt korjauskortit. Korjauskortteja voi kerrallaan kortissa olla vain yksi kappale.

Vaatimus **RW03** määrittelee vastauskortit sekä korjauskortit ei-hierarkkisiksi korteiksi. Kortteihin ei voida sovelluksessa lisätä toisia kortteja. Tämä tarkoittaa vastauskorteissa sitä, että yhden kommenttikortin keskustelu tapahtuu vain yhdessä vastausketjussa.

Vaatimus **RW04** on samanlainen vaatimus kuin Kanban-sovelluksessa. Korttien rakennetta pitää pystyä muuttamaan sovelluksen ohjelmakoodin ulkopuolelta. Vaatimus **RW05** on myös sama kuin Kanban-sovelluksessa. Muutosten, viitteiden, kommenttien ja korjauksten tulee päivittyä muutosten ilmetessä kaikille samat kortit näkeville käyttäjille.

Sovelluksen toiminnalliset vaatimukset esitellään taulukossa 3.4.

Tunniste	Nimi	Kuvaus
RW06	Graafiset viitteet	Kommenttikorteista voidaan vetää graafisia viitteitä katselmoitavan työkokonaisuuden kenttiin. Viitata voi useampaan kenttään.
RW07	Viitteiden poisto	Viitteiden poisto kortista mahdollista.
RW08	Viitteiden näyttäminen ja piilotus	Kommenttiin tai kenttään liittyvät viitteet tulee voida kytkeä näkyviksi tai piilotetuiksi.
RW09	Uuden kommenttikortin luonti	Uusia kommentteja on kyettävä luomaan sovelluksesta.
RW10	Vastauskortin luonti	Kommenttikorttiin tulee kyetä luomaan vastauskortteja.
RW11	Korjauskortin luonti	Kommenttikorttiin tulee kyetä luomaan korjauskortti.
RW12	Korttien sisältöjen piilotus	Korttien sisältöjä tulee kyetä piilottamaan.
RW13	Korttien tietojen muokkaus	Korttien ominaisuuksia tulee kyetä muokkaamaan kortin ominaisuuksien kentistä.
RW14	Korttien poisto	Kortteja tulee kyetä poistamaan. Mikäli poistettava kortti on kommenttikortti, tulee myös sen alikortit poistaa.

Taulukko 3.4 *Katselmointisovelluksen toiminnot*

Vaatimus **RW06** on katselmointisovelluksen keskeinen toiminnallinen vaatimus. Viitteet vedetään raahaamalla viite kommenttikortista viitattavaan työkokonaisuuden kenttään. Viitteet esitetään nuolina kommentista viittauksen kohteeseen. Luonnollisesti viittauksen raahaaminen tulee voida perua käyttäjän niin tahtoessa. Onnistuneessa viittauksen raahauksessa päivitetään kommentin tiedot Realm-järjestelmään.

Vaatimukset **RW07** ja **RW08** ovat viittauksien käsittelyn toiminnallisuuksia. Ensimmäisessä vaatimuksessa määritellään viittauksien poistaminen toiminnallisuutena. Toiminnallisuus tulee löytyä kortin käyttöliittymästä. Viitteiden näyttämisen kytkeminen tarkoittaa piiloitettujen näyttämistä tai näkyvien piilottamista käyttäjältä. Kuten poistaminen, tämänkin toiminnallisuuden on löydyttävä kortin käyttöliittymästä.

Vaatimus **RW09** määrittelee toiminnon uuden kommentin luomiseksi. Kommentin luomisen täyttävä toiminto on löydyttävä sovelluksen käyttöliittymästä. Uutta

kommenttia luodessa kommentti tulee luoda ja näyttää käyttäjälle ilman pohjasivun uudelleenpäivittämistä.

Vaatimukset **RW10** ja **RW11** määrittävät toiminnot, joilla kommenttikortteihin lisätään toisia kortteja. Sekä vastaus- että korjauskorttien luomisen toiminnallisuus on löydyttävä kommenttikortista. Korttien luomisen täytyy tapahtua kuin vaatimuksessa **RW08**, eli asynkronisesti, ilman verkkosivun päivitystä.

Vaatimus **RW12** määrittelee samankaltaisen toiminnallisuuden kuin **KB07**. Kaikista korteista tulee voida piilottaa jokin määritelty osa kortin käyttöliittymästä.

Vaatimus **RW13** määrittelee toiminnon korttien tietojen muokkaukselle. Korttien tietoja tulee voida muokata kortista itsestään. Vaatimuksessa **RW13** määritellään, että kortteja pitää pystyä poistamaan. Kortin poiston yhteydessä myös sen osoittama tieto tulee poistua. Mikäli kortilla, tässä tapauksessa kommenttikortilla, on alikortteja, myös niiden tiedot täytyy poistaa.

3.3 MyWork-sovellus

MyWork-sovellus jakaa hyvin paljon yhteistä toiminnallisuutta Kanban-sovelluksen kanssa. Kumpikin on tarkoitettu yksittäisen työntekijän työnohjaukseen. MyWork-sovellus toimii työntekijän niin sanottuna kojelautana, johon kootaan yhteen kaikki kyseisen työntekijän työtehtävät ja niiden toteutusaikataulu.

Sovellus koostuu työkorteista ja tilakorteista. Työkortit edustavat jotain yksittäistä suoritettavaa työkokonaisuutta. Tilakortit taasen edustavat työkortin toteuttamisen aikataulua. Esimerkiksi yksi tila voisi olla "Tänään", jolloin kaikki tilassa olevat työtehtävät tulisi toteuttaa saman päivän aikana. Työkortit ovat tilassa aina prioriteetin määräämässä järjestyksessä siten, että tärkein on ensimmäisenä.

3.3.1 Työkortit

Työkortit edustavat aina yhtä työkokonaisuutta. Työkorttien ideana on määrittää työntekijälle työtehtäviä. Työkortteihin halutaan alustavasti ainoastaan työtehtävää kuvaava nimi ja linkki edustettuun työkokonaisuuteen. Itse kortissa ei ole suurempaa toiminnallisuutta.

Työkortteja tullaan mahdollisuuksien mukaan jatkokehittämään uusien vaatimusten noustessa pinnalle. Tämä tulee ottaa huomioon sekä arkkitehtuurin suunnittelussa että sovelluksen kehittämisessä.

3.3.2 Tilakortit

Tilakortit toimivat samalla tavalla kuin Kanban-sovelluksen tilakortit. Tilakortit koostuvat tilan tiedoista ja työkorteista. Tilakortissa aina näkyy kyseisen tilan nimi.

Työkorttien tiloja voidaan vaihtaa raahaamalla työkortti yhdestä tilasta toiseen. Kortin prioriteettiä tulee kyetä muuttamaan raahaamalla sitä tilan sisällä. Kun kortti raahataan tilasta toiseen, prioriteetti uudessa tilassa määräytyy kortin tiputuskohdan mukaan.

3.3.3 MyWork-sovelluksen korttien vaatimukset

MyWork-sovelluksen vaatimukset ovat lähes identtiset Kanban-sovelluksen vaatimusten kanssa, mutta koska Kanban-sovelluksen yhteydessä määriteltiin vaatimukset Kanban-sovelluksen viitekehyksessä, esitetään MyWork-sovellukselle omat vaatimuksensa käyttämällä sovellukselle ominaista sanastoa. Sovelluksen ei-toiminnalliset vaatimukset esitellään taulukossa 3.5.

Tunniste	Nimi	Kuvaus
MW01	Korttipohjaisuus	Sovelluksen data esitetään kortteina.
MW02	Tilakorttien sisältö	Tilakortit koostuvat työkorteista.
MW03	Työkorttien sisältö	Työkortit eivät ole hierarkkisia.
MW04	Korttien sisällön määrittäminen	Korttien sisällöt voidaan uudelleenmäärittää sovelluksen ohjelmakoodia muokkaamatta.
MW05	Monen käyttäjän yhdenaikainen käyttö	Korttien tietojen ja tilojen tulee päivittyä muille käyttäjille muutoksen jälkeen.

Taulukko 3.5 *Mywork-sovelluksen ei-toiminnalliset vaatimukset*

Vaatimus **MW01** määrittelee sovelluksen korttipohjaiseksi sovellukseksi. Vaatimus **MW02** määrittelee MyWork-sovelluksen tilakortin hierarkkiseksi kortiksi, joka koostuu työkorteista. Vaatimuksen **MW03** mukaan työkortit eivät voi olla hierarkkisia, joten niissä ei voi olla toisia työkortteja alikortteina. Vaaitmus **MW04** määrittelee näkymän muokattavuuden sovelluksen ulkopuolelta ja vaatimus **MW05** määrittelee reaaliaikaisuuden tietojen päivityksen osaksi sovellusta.

MyWork-sovelluksen toiminnallisuus on hyvin yksinkertaista ja koostuu vain muutamasta vaatimuksesta. Tästä huolimatta arkkitehtuuria suunniteltaessa on otettava huomioon mahdollinen jatkokehitys. Sovelluksen toiminnalliset vaatimukset on esitelty taulukossa 3.6.

Tunniste	Nimi	Kuvaus
MW06	Tilakorttien piilotus	Tilakortteja voidaan piilottaa. Tilakorttia piilottaessa myös sen sisältämät työkortit piilottuvat.
MW07	Tilasiirtymät	Työkortin tila muutetaan raahaamalla kortti tilakorttiin.
MW08	Korttien järjestäminen	Työkortteja tulee pystyä järjestämään raahaamalla niitä eri kohtaan tilakortissa.

Taulukko 3.6 *Mywork-sovelluksen toiminnot*

Vaatimus **MW06** määrittelee korttien osien piilottamisen kortin toiminnallisuudeksi. Toiminnallisuuden on löydyttävä kortin käyttöliittymästä.

Vaatimus **MW07** määrittelee tilasiirtymien toteutuksen työkorteille. Työkorttien tilasiirtymät toteutetaan raahaamalla ne toisen tilakortin sisään. Raahaaminen muuttaa raahattavan työkortin tilan uutta tilaa vastaavaksi.

Vaatimus **MW08** määrittelee korttien järjestämisen toiminnallisuuden. Kortteja tulee voida järjestää tilakorteissa raahaamalla ja tiputtamalla niitä uutta järjestystä vastaavaan paikkaan. Oletuksena työkortit järjestetään prioriteetin perusteella laskevassa järjestyksessä.

4. TOTEUTETTAVAN ARKKITEHTUURIN VAATIMUKSET

4.1 Sovellusten yhteisiä ominaisuuksia

Tämän kohdan alikohdissa käydään läpi sovellusten yhteisten ominaisuuksien arviointi. Arviointia voidaan käyttää apuvälineenä sovelluksia tukevan sovellusarkkitehtuurin suunnittelussa. Ominaisuuksia arvioidaan ensin rakenteellisilla yhtäläisyyksillä, joilla tarkoitetaan toteutettavien sovellusten ei-toiminnallisia ominaisuuksia.

Toisena arviointikriteerinä on toiminnalliset yhtäläisyydet, joilla kartoitetaan samoja toimintoja tai toimintatapoja sovellusten välillä. Näistä pyritään johtamaan suunnitteluosassa yleisiä toimintojen käyttötapauksia toteuttavia komponentteja.

4.1.1 Rakenteelliset yhtäläisyydet

Sovelluksissa on keskeisenä käsittenä kortti (**KB01**, **RW01**, **MW01**). Näkymäarkkitehtuuri tulee rakentumaan korttipohjaisten käyttöliittymien ympärille. Kortti voidaan määritellä yhdentyyppisen mallidatan visuaaliseksi esitykseksi. Koska arkkitehtuuri suunnitellaan web-sovelluksia varten, korttien toteutustekniikat rajoittuvat hyvin suuresti. Korttien toteutuksen mahdollistavat tekniikat ovat HTML-merkkaus, CSS-tyylimääreet ja toimintaa määrittävä JavaScript.

Kortit ovat hierarkkisia (**KB02**, **KB03**, **RW02**, **MW02**). Korttien rakentamisessa voidaan ottaa huomioon se, että kortit koostuvat muista korteista. Kuitenkin on olemassa kortteja, joissa hierarkkisuus on evätty (**RW03**, **MW03**). Nämä ominaisuudet on huomioitava arkkitehtuurin suunnittelussa.

Korttien sisällöt tulee pystyä määrittelemään sovelluksen ulkopuolisessa lähteessä (**KB04**, **RW04**, **MW04**). Ulkopuolinen lähde voi olla esimerkiksi verkkoresurssi tai asetustiedosto. Arkkitehtuurin tulisi olla riippumaton tavasta, jolla sille syötetään tiedot kortin rakenteesta.

Sovelluksia yhdistää myös vaatimus reaaliaikaisesta päivittämisestä (**KB05**, **RW05**, **MW05**). Käytännössä tämä tarkoittaa ajoittaista muutosten pyytämistä palvelimelta tai jatkuvaa yhteyttä. Valitun metodin suhteen pyritään arkkitehtuurissa olemaan riippumattomia siten, että valittua metodia voidaan jälkikäteen helposti vaihtaa.

4.1.2 Toiminnalliset yhtäläisyydet

Sovelluksen korteilla on huomattava määrä yhteistä ja yleistä toiminnallisuutta. Korttien sisältöjen piilottaminen (**KB06**, **RW11**) ja kokonaisten korttien piilottaminen (**KB07**, **MW06**) ovat hyvin samankaltaisia toiminnallisuuksia. Käyttäytyminen voidaan toteuttaa määrittelemällä piilotettavalle osalle CSS-tyylimääreellä näkyvyyttä muokkaava sääntö. Näissä tapauksissa ainoa ero on piilotettavan osan CSS-valitsin. Tämä ei kuitenkaan ole este sille, etteikö samaa ohjelmakoodia voitaisiin käyttää kumpaankin tapaukseen.

Korttien raahaaminen ja järjestäminen on myös yleinen teema (**KB08**, **KB09**, **KB10**, **MW07**, **MW08**). Raahaaminen voidaan toteuttaa HTML5-standardiin työversioon määritellyllä Drag and Drop -mekanismilla. [21] Samalla mekanismilla voidaan toteuttaa myös graafisten viitteiden raahaaminen (**RW05**).

4.1.3 Yhtäläisyyksien vaikutus arkkitehtuurin suunnitteluun

Sovelluksien yhteisille ominaisuuksille pyritään luomaan yleisiä toteutusmenetelmiä. Arkkitehtuuri tulee havaittujen ominaisuuksien valossa painottumaan korttipohjaisten näkymien rakentamiseen selaimessa. Koska järjestelmässä on jo käsitteet mallielementistä ja näkymistä, voidaankin pääasialliseksi sovellusarkkitehtuurimalliksi valita mukailtu MVC-arkkitehtuurimalli. Näkymän rakenne määräytyy täten elementtipohjien perusteella rakennetun HTML-merkkauksen mukaiseksi ja niiden ulkoasua muokataan näkymäelementeiksi tallennetuilla CSS-säännöillä.

Useimmiten kortit edustavat aina Realm-järjestelmän palvelimelta saatavia mallielementtejä. Koska MVC-arkkitehtuurimallissa on valmiiksi termit **Malli** ja **Näkymä**, voidaan pääasialliseksi arkkitehtuurimalliksi valita kohdassa 2.6.2 esitelty mukautettu MVC-arkkitehtuurimalli. Malli tarkoittaa Realm-järjestelmän palvelimelta saatavaa mallielementtiä sekä näkymä tarkoittaa mallielementistä elementtipohjan avulla muodostettua käyttöliittymäelementtiä. Arkkitehtuuriin määritellään termi **Kortti**, joka tarkoittaa mallin ja näkymän muodostamaa entiteettiä, johon voidaan liittää

toiminnallisuutta entiteetti-komponentti-järjestelmä -arkkitehtuurimallin mukaisilla komponenteilla.

Sovellusten toiminnallisuus toteutetaan täten korteilla, jotka vastaavat käyttäjän verkkoselaimesta tuleviin tapahtumiin. Toiminnallisuus tuotetaan JavaScriptillä siten, että HTML5-tekniikan mukana tulleita uusia ominaisuuksia käytetään mahdollisimman paljon. Tämä nostaa sovellusten suorituskykyä siirtämällä toiminnallisuuden toteutusvastuuta tulkatusta JavaScript-kielestä verkkoselainten natiivitoteutuksille.

Koska erityyppisille malleille halutaan erilaisia näkymiä, on näkymien rakenne teknisesti helpoin toteuttaa mallipohjilla. Mallipohjien ideana on määrittää näkymän rakenne siten, että kaikilla samaa mallipohjaa käyttävillä näkymillä rakenne on identtinen. Voidaankin määritellä, että mallipohjat vastaavat Realm-järjestelmän elementtipohjien perusteella rakennettua HTML-merkkausta. Mallipohjien käyttö sovelluksen käyttämien näkymäelementtien rakentamisessa mahdollistaa näkymän ulkoasun helpot muutokset, sillä muutokset elementtipohjissa vaikuttavat kaikkiin samaa mallipohjaa käyttäviin näkymiin.

4.2 Ei-toiminnalliset vaatimukset

Arkkitehtuurin vaatimukset esitetään taulukoissa, joissa luodaan jokaiselle vaatimukselle yksikäsitteinen tunniste. Näitä tunnisteita käytetään työssä viittaamaan tiettyihin vaatimuksiin. Arkkitehtuurille luodut vaatimukset voidaan jakaa kahteen kategoriaan, toiminnallisiin ja ei-toiminnallisiin vaatimuksiin. Vaatimukset käsitellään omissa alakohdissaan. Vaatimuksista luodaan tarvittaessa skenaarioita kuvaamaan vaatimuksen vaikutusta käytännössä.

Ei-toiminnalliset vaatimukset ovat järjestelmän vaatimuksia sille, miten tulevien ohjelmistojen tulee täyttää toiminnalliset vaatimukset. Ei-toiminnalliset vaatimukset voidaankin nähdä pohjana sille, mitä rajoituksia ja vaatimuksia toiminnallisten vaatimusten täytyy noudattaa. Ei-toiminnalliset vaatimukset tässä työssä jaotellaan neljään kategoriaan:

- Yleiset vaatimukset
- Suorituskykyvaatimukset
- Erikoistamisvaatimukset
- Käytettävyysvaatimukset

Toiminnalliset vaatimukset ovat kaikille arkkitehtuurille rakennettujen ohjelmistojen yhteisiä konkreettisiä vaatimuksia. Vaatimukset koskettavat kaikkien ohjelmistojen yleisiä elementtejä, ja vaikuttavat suuresti kaikkien arkkitehtuurille kehitettyjen ohjelmistojen yleiseen käyttötuntumaan.

4.2.1 Yleiset vaatimukset

Taulukossa 4.1 kuvataan arkkitehtuurin yleiset vaatimukset. Yleisten vaatimusten tunnisteet alkavat aina kirjaimella **A**. Yleisillä ei-toiminnallisilla vaatimuksilla tässä työssä tarkoitetaan arkkitehtuurivaatimuksia ja kehitystyön vaatimuksia. Arkkitehtuurivaatimukset kuvaavat sitä, miten arkkitehtuurin päälle rakennetut ohjelmistot liittyvät toimintaympäristöönsä. Kehitystyön vaatimukset kuvaavat sallittuja ohjelmistotekniikan menetelmiä arkkitehtuurin päälle rakennettavien ohjelmistojen toteutuksessa. Vaatimukset eivät liity ohjelmiston käyttöön, vaan ne toimivat piilevinä rajoittavina tekijöinä ohjelmistojen rakenteessa.

Yleiset vaatimukset varmistavat sen, että tuotettu arkkitehtuuri on yhteensopiva olemassaolevan mallipohjaisen ympäristön kanssa. Yleisissä vaatimuksissa myös otetaan kantaa varsinaisen kehitystyön raameihin. Kehitystyön toimintatapoja ja toteutustekniikoita rajoittavat eniten valittu selainpohjainen suoritusalusta. Myös jatkokehityksessä halutut mobiili pohjaiset ominaisuudet on otettu huomioon.

Tunniste	Nimi	Kuvaus
A01	Yleinen arkkitehtuuri	Arkkitehtuuri on yleinen ja mahdollistaa korttipohjaisten sovellusten ja käyttöliittymien nopean kehityksen ja muokkaamisen.
A02	Liitos olemassaoleviin sovelluksiin	Arkkitehtuurin täytyy toimia erityisesti Cometan mallipohjaisen ympäristön päällä.
A03	Ajoympäristö	Ajoympäristönä toimivat nykyaikaiset selaimet, sekä yleisimmät mobiililaitteet natiivisti.
A04	Kehitysympäristö	Ohjelmiston kehitykseen vaaditaan Windows- tai Linux-ympäristö.
A05	Mobiilikehitys	Mahdollinen tuki mobiililaitteille jatkokehityksessä huomioitava.
A06	Testattavuus	Arkkitehtuurin komponenttien ja sillä toteutettujen sovelluksien tulee olla testattavia.

Taulukko 4.1 *Yleiset vaatimukset*

Vaatimus **A01** muodostaa pohjan, jonka päälle koko arkkitehtuuri rakennetaan. Arkkitehtuurilla on mahdollista kehittää ja muokata korttipohjaisia sovelluksia ja käyttöliittymiä.

Vaatimus **A02** määrittää arkkitehtuuria pohjustavan tekniikan. Arkkitehtuurin tulee toimia ensisijaisesti vaatimuksessa määritellyn mallipohjaisen ympäristön päällä. Ympäristöä ei sidota vahvasti työssä käytettävään pohjustavaan tekniikkaan, eli arkkitehtuuria voisi käyttää muissakin ympäristöissä. Työn puitteissa ei arkkitehtuuria kehitetä muille ympäristöille.

Vaatimus **A03** määrittää arkkitehtuurin ajoympäristön. Ajoympäristönä ovat modernit verkkoselaimet sekä yleisimmät mobiililaitteet. Tämä tulee ottaa huomioon sovelluksen teknisten ratkaisujen päätöksissä. Pyrkimyksenä on toteuttaa toiminnallisuuksia kaikissa moderneissa ympäristöissä toimivilla tekniikoilla ja rajapinnoilla.

Vaatimus **A04** määrittää arkkitehtuurille soveltuvan kehitysympäristön. Kehitysympäristö rajoittuu Windows- ja Linux-ympäristöihin.

Vaatimus **A05** määrittää mahdollisen tuen mobiilikehitykselle. Arkkitehtuurissa tulee ottaa huomioon mobiililaitteiden erilaiset laitekannat ja yleisimmät ajoympäristöt.

Vaatimus **A06** määrittää arkkitehtuurin valmiiden komponenttien ja sillä toteutettujen sovelluksien testauksen mahdollisuuden. Komponenttien tulisi olla testattavia yksikkötestitasolta alkaen. Sovelluksissa pyritään myös toteuttamaan tuki yksikkötason testeille. Mahdollinen korkeamman tason testiautomaatio tulee ajankohtaiseksi web-sovellusten testausmenetelmien kehittyessä.

4.2.2 Suorituskykyvaatimukset

Arkkitehtuurille halutaan asettaa vaatimukset sillä toteutettujen sovellusten suorituskyvylle. Suorituskyky web-ympäristössä koostuu suoritettavien sovellusten toteutuksesta, niiden muistijäljestä sekä ladattavien resurssien määrästä. Vaatimukset asettavat rajoitteita edellämainituille suorituskyvyn tekijöille.

Tunniste	Nimi	Kuvaus
S01	Käyttäjän havaitsema tehokkuus	Suorituskyky on hyvä eikä hidastele.
S02	Toiminta huonoilla yhteyksillä	Järjestelmän tulee toimia myös huonoilla yhteyksillä datamäärän suuruudesta välittämättä.

Taulukko 4.2 *Suorituskykyvaatimukset*

Taulukossa 4.3 on kuvattu suorituskykyvaatimusten skenaariot. Skenaariot konkretisoivat suorituskyvyn vaatimukset käytännön esimerkeiksi.

Tunniste	Vaatimus	Kuvaus
SC1	**S01**	Käyttäjä siirtää kortin paikasta toiseen. Kortti liikkuu sulavasti.
SC2	**S02**	1000 korttia halutaan Kanban-tyyliseen muotoon. Kortit edustavat erityyppisiä työkokonaisuuksia. Käyttökelpoisen näkymän avaaminen ei saa kestää yli 5 sekuntia.
SC3	**S02**	Käyttäjä avaa kortin. Kortti avautuu alle sekunnissa.

Taulukko 4.3 *Suorituskykyvaatimusten skenaariot*

Vaatimus **S01** määrittää käyttäjälle näkyvän sovelluksen suorituskyvyn. Sovellusten ei tulisi hidastua missään vaiheessa sovelluksen ajoa. Tarkempi kuvaus esitellään skenaariossa **SC1**.

Vaatimus **S02** määrittää sovellusten toiminnan huonoilla yhteyksillä. Sovellusten tulisi toimia kaikilla yhteyksillä datamääristä riippumatta. Vaatimukseen löytyy tarkemmat skenaariot, **SC2** ja **SC3**.

4.2.3 Erikoistamisvaatimukset

Arkkitehtuurille on laadittu vaatimukset näkymien ja sovellusten erikoistamiseen. Erikoistamisvaatimukset koskevat sekä käyttäjien toteuttamaa ajonaikaista erikoistamista, kuten näkymäelementtien ulkomuodon muokkaamista tai koko ulkoasun uudelleensommittelua. Suuremmat vaatimukset erikoistamiselle tulevat edistyneen

käyttäjän erikoistamisesta. Edistyneen käyttäjän erikoistamiset koskevat muun muassa olemassaolevien käyttökohteiden huomattavaa muokkaamista kaikille käyttäjille, ja uusien käyttökohteiden luomista.

Erikoistamisvaatimukset on esitelty taulukossa 4.4. Erikoistamisvaatimusten tunnisteet alkavat aina kirjaimella **E**.

Tunniste	Nimi	Kuvaus
E01	Normaalikäyttäjän muokkaukset	Käyttäjän tulee pystyä tekemään kevyitä erikoistuksia näkymiin.
E02	Kehittyneen käyttäjän muokkaukset	Kehittyneen käyttäjän pitää pystyä määrittämään yhteen käyttökohteeseen uusia elementtejä, tilakäyttäytymistä ja ulkoasumuutoksia.
E03	Uusi käyttökohde	Järjetelmään on kyettävä ohjelmoimaan uusia käyttökohteita, joissa uusia kenttiä ja käyttäytymistä.
E04	Uusi käyttötapaus	Järjestelmään on kyettävä ohjelmoimaan täysin uusia käyttötapauksia.
E05	Uuden käyttötapauksen yleistäminen	Mikäli järjestelmään luotu uusi käyttötapaus koetaan yleiseksi, se tulee voida sisällyttää osaksi järjestelmän kehystä.
E06	Uudet sovellukset	Järjestelmään on kyettävä luomaan uusia sovelluksia.
E07	Uusien sovellusten käyttöliittymät	Järjestelmän sovelluksiin on kyettävä luomaan erilaisia käyttöliittymiä eri kohdelaitteille.

Taulukko 4.4 *Erikoistamisvaatimukset*

Vaatimukset avataan erikseen ja yhdistetään niitä vastaaviin skenaarioihin. Skenaariot erikoistamisvaatimuksille on esiteltynä taulukossa 4.5.

Vaatimus **E01** määrittää normaalille sovelluksen käyttäjälle mahdolliset erikoistamistoiminnot. Vaatimuksella määritellään mahdollisuus muokata sovelluksen käyttäjälle näkyvien osien ulkonäköä. Skenaariota vaatimukselle ei ole määritelty.

Vaatimus **E02** määrittää kehittyneelle käyttäjälle mahdolliset erikoistamistoiminnot. Vaatimuksella määritellään mahdollisuus toteuttaa uusia näkymiä olemassaolevan ohjelmakoodin avustuksella. Uusi näkymä siis koostetaan olemassaolevasta

Tunniste	Vaatimus	Kuvaus
SC4	**E02**	MyView-näkymän, jossa ei uusia kenttiä, tekeminen mahdollista olemassaolevaa ohjelmakoodia uudelleenkäyttämällä.
SC5	**E03**	MyView-näkymän, jossa kokonaan uusia kenttiä, tekeminen 2 päivässä. Uuden ohjelmakoodin minimimäärällä. (Yleinen käyttäytyminen ei muutu)
SC6	**E03**	Review-näkymän, jossa kokonaan uutta käyttäytymistä ja uusia kenttiä, tekeminen viikossa. (Uusia käyttötapauksia)
SC7	**E04**	Käyttäjä pystyy tekemään kortin ja elementin/kentän/-muun sisällön välille toiminnon, jolla vedetään graafisia viitteitä elementtien välille. Käyttötapauksen toteutukselle ei aseteta aikarajaa, mutta tyypillisessä tapauksessa ensimmäisen alustavan version pitäisi olla valmis päivässä.
SC8	**E05**	Mikäli uusi käyttötapaus todetaan yleiseksi, tulee se kyetä sisällyttämään osaksi kehystä.
SC9	**E06**, **E07**	Käyttäjä haluaa organisaation CRM-ohjelmiston, jossa on Android- ja web-pohjainen yksinkertainen käyttöliittymä. Toteutukselle ei aseteta aikarajaa, mutta tavoitteena on, että ensimmäinen alustava versio pitäisi valmistua X päivässä, missä X on käyttötapauksien määrä.

Taulukko 4.5 *Erikoistamisvaatimusten skenaariot*

ohjelmakoodista. Tarkempi kuvaus esitetään skenaariossa **SC4**.

Vaatimus **E03** määrittelee mahdollisuudet kehittää arkkitehtuurilla uusia sovelluksia uusilla käyttötapauksilla ja näkymillä. Tarkempi kuvaus esitetään skenaarioissa **SC5** ja **SC6**.

Vaatimus **E04** määrittelee mahdollisuudet toteuttaa arkkitehtuurilla uusia käyttötapauksia. Käyttötapaukset tässä yhteydessä ovat sovellussidonnaisia ja niiden täytyy toimia vain tietyssä sovelluksessa. Tarkempi kuvaus esitetään skenaariossa **SC7**.

Vaatimus **E05** määrittelee uuden käyttötapauksen yleistämisen kaikille sovelluksille yhteiseksi toiminnallisuudeksi. Uusi toiminnallisuus kirjastoitaisiin täten mahdollisimman yleisenä osaksi arkkitehtuuria. Tarkempi kuvaus esitetään skenaariossa **SC8**.

Vaatimus **E06** määrittelee mahdollisuuden luoda arkkitehtuurilla täysin uusia sovelluksia. Arkkitehtuurilla pitäisi pystyä toteuttamaan itsenäisesti suurempiakin so-

velluksia, jotka koostuvat merkittävästä määrästä uusia käyttötapauksia. Tarkempi kuvaus esitetään skenaariossa **SC9**.

Vaatimus **E07** määrittelee mahdollisuuden toteuttaa sovelluksille vaihtoehtoisia käyttöliittymiä eri kohdelaitteille. Kohdelaitteita ovat esimerkiksi älypuhelimet ja tabletit. Tarkempaa skenaariota ei ole vaatimukselle määritelty, mutta uudet kohdelaitteet on mainittu skenaariossa **SC9**.

4.2.4 Käytettävyysvaatimukset

Arkkitehtuurilla toteutettujen sovelluksien käytettävyydelle halutaan asettaa vaatimuksia. Vaatimuksilla pyritään luomaan konkreettiset tavoitteet sovellusten käytettävyydelle, sekä pyritään luomaan yhtenäinen ohjeistus sovellusten välisen käytön ja toimintojen toteutuksien yhtenäisyydelle.

Käytettävyysvaatimukset on esitelty taulukossa 4.6. Käytettävyysvaatimusten tunniste alkaa aina kirjaimella **K**.

Tunniste	Nimi	Kuvaus
K01	Työvaiheiden määrä	Uusi kortti on kyettävä luomaan mahdollisimman pienellä määrällä työvaiheita.
K02	Intuitiivisuus	Kortteihin pohjautuva sovellus on intuitiivinen ja yksinkertainen käyttää.
K03	Vikasietoisuus	Käyttäjä ei pysty toiminnoillaan sekoittamaan sovelluksen toimintaa.
K04	Avustava käyttöliittymä	Käyttöliittymä avustaa käyttäjää ymmärtämään, mitä tietoa näytetään, mitä kenttiä pitää näyttää ja miten ohjelmaa yleisesti käytetään.
K05	Käyttöliittymän yleisyys ja ymmärrettävyys	Korttien käyttötapa on yleinen ja ymmärrettävä. Kortit toimivat kaikkialla johdonmukaisesti samalla tavalla.

Taulukko 4.6 *Käytettävyysvaatimukset*

Vaatimukset avataan erikseen ja niistä johdetaan tarpeen vaatiessa skenaarioita selvittämään vaatimuksen vaikutusta käytännössä. Käytettävyysvaatimusten skenaariot ovat esiteltynä taulukossa 4.7.

Tunniste	Vaatimus	Kuvaus
SC10	**K01**	Käyttäjä luo uuden kortin. Kortti ilmestyy ruudulle ja on kohdistettuna muokkausta varten.
SC11	**K02**	Käyttäjä käyttää ensimmäistä korttipohjaista sovellustaan ja osaa pienen kokeilun jälkeen käyttää tärkeimpiä toimintoja.
SC12	**K03**	Käyttäjä käyttää korttipohjaista sovellusta ensimmäistä kertaa ja kokeilee kaikenlaista. Käyttäjä ei silti pysty sotkemaan sovelluksen tilaa peruuttamattomasti.
SC13	**K04**	Käyttäjä hooveroi kentän nimen päälle ja saa kuvauksen kentän sisällöstä.
SC14	**K04**	Käyttäjä näkee kortista suoraan pakolliset täytettävät kentät.
SC15	**K05**	Korttien avaus ja sulkeminen, siirtäminen, luonti, jne. toimii kaikkialla samoin.

Taulukko 4.7 *Käytettävyysvaatimusten skenaariot*

Vaatimus **K01** asettaa rajoituksen uusien korttien luomiseen vaadittujen työmäärien määrään. Tavoitteena on toteuttaa toiminnallisuus, jossa korttien luominen on nopeaa, luonnollista ja työntekijälle tehokasta. Tarkempi käytännön kuvaus vaatimukselle esitetään skenaariossa **SC10**.

Vaatimus **K02** määrittää sovelluksien intuitiivisuuden ja yksinkertaisuuden rajoitteet. Sovelluksista pyritään luomaan mahdollisimman yksinkertaisia ja intuitiivisia käyttää. Mahdollisuuksien mukaan sovelluksien toteutuksessa voidaan käyttää käyttäjäläheisiä kehitysmenetelmiä varmistamaan vaatimuksen mukainen intuitiivisuuden ja yksinkertaisuuden taso. Konkreettisempi kuvaus vaatimuksesta on esiteltynä skenaariossa **SC11**.

Vaatimus **K03** määrittelee sovelluksille vaaditun vikasietoisuuden tason. Vaatimuksen mukaisesti käyttäjän ei tulisi koskaan pystyä sekoittamaan sovelluksen toimintaa. Kuvaus vaatimuksesta on esiteltynä skenaariossa **SC12**.

Vaatimus **K04** asettaa sovellukselle vaatimuksia sen antamasta ohjeistuksesta käyttäjälle. Vaatimuksen mukaan sovelluksen tulee opastaa käyttäjää sovelluksen käytöstä. Konkreettisempi kuvaus vaatimuksesta on esiteltynä skenaarioissa **SC13** ja **SC14**.

Vaatimus **K05** asettaa vaatimuksia arkkitehtuurilla toteutettujen sovellusten välisten toiminnallisuuksien johdonmukaisuudelle. Korttien tulee käyttäytyä jokaisessa arkkitehtuurilla luodussa sovelluksessa johdonmukaisesti, ja käyttäjän tekemien samanlaisten syötteiden pitäisi jokaisessa sovelluksessa johtaa samankaltaiseen sovelluksen tilaan. Kuvaus vaatimuksesta esitellään skenaarioissa **SC14**.

4.3 Työssä vaaditut toiminnalliset vaatimukset

Korttien toiminnalliset vaatimukset on kuvattu liitteessä **A**. Toiminnallisten vaatimusten tunnus alkaa aina kirjaimella **T**.

Toiminnalliset vaatimukset eivät ota kantaa arkkitehtuurin toteutukseen, mutta ne esitellään arkkitehtuurilla toteutettavien sovellusten yleiskuvan välittämiseksi. Toiminnalliset sovellukset eivät rajoitu esitettyihin toiminnallisuuksiin, vaan toiminnallisuuksia lisätään uusien sovelluksien kehityksen yhteydessä.

5. TOTEUTETTAVAN ARKKITEHTUURIN SUUNNITTELU

Tässä luvussa kuvataan suunniteltu arkkitehtuuri ja suunnittelussa käytetyt periaatteet. Tämän lisäksi kuvataan suunnittelun kannalta oleellisia ratkaisuja, toteutustekniikoita sekä toteutettujen komponenttien toimintaa.

5.1 Suunniteltu arkkitehtuuri

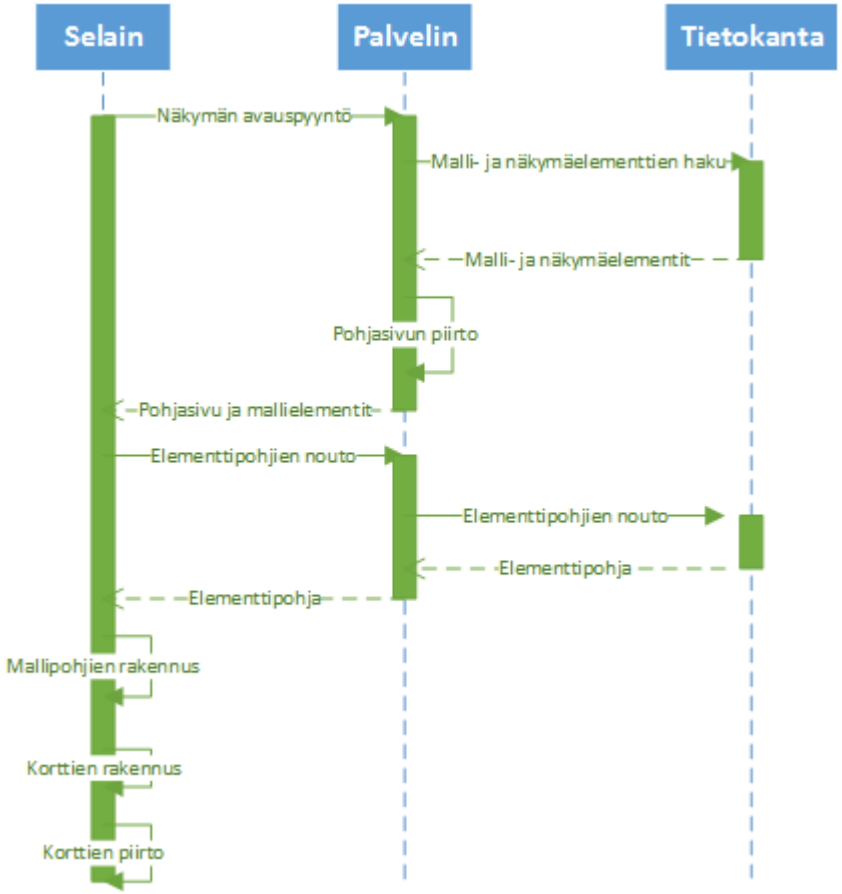
Suunniteltu arkkitehtuuri vastaa luvussa 2.3 kuvatun Realm-järjestelmän nykyistä yleisarkkitehtuuria, koska muutokset tapahtuvat fyysisen järjestelmän sijaan toimintatavoissa. Palvelimen vastuu sivunpiirrosta vähenee huomattavasti, sillä mallielementtien piirto siirretään selaimessa ajettavien sovellusten vastuulle. Palvelimessa tapahtuvassa piirrossa pääpaino siirtyy pohjasivun piirtämiseen. Pohjasivu siis tarkoittaa verkkosivua, jossa on yleisiä Realm-järjestelmässä toteutettavia toimintoja kuten navigointia. Tämän lisäksi pohjasivussa on arkkitehtuurin sovelluksien piirtoalueita, joihin selainsovellukset piirtävät mallielementeille generoituja kortteja.

Mallipohjaisessa tietokannan vastuut ovat käytännössä identtiset aikaisempaan. Muutos tapahtuu näkymäelementtien käyttöönotossa. Näkymäelementteihin halutaan tallentaa ainoastaan kyseiselle mallielementille tehtyjen ulkoasumuokkausten CSS-sääntöjä. Näillä CSS-säännöillä on mahdollista muokata vain kyseisen mallielementin kortin ulkoasua. Muutoin vastuualueet pysyvät identtisinä.

Palvelimen uutena vastuuna on asettaa käytettävien näkymäelementtien CSS-säännöt osaksi pohjasivua. Tämä siksi, että selaimessa ajettavissa sovelluksissa on haastavaa muokata verkkosivun globaaleja CSS-sääntöjä. Tämä myöskin vähentää huomattavasti selaimesta tehtävien pyyntöjen määrää. Toisena vastuuna palvelimella on ilmoittaa selaimessa toimivalle sovellukselle, missä näkymässä näytettävät mallielementit sijaitsevat. Muutoin palvelimen vastuualueet pysyvät samoina.

Selaimen vastuulla on uutta arkkitehtuuria toteuttavien sovellusten ajo, mallielementtien HTML-merkkauksen generointi elementtipohjien perusteella, korttien rakentaminen ja korttien piirto. Koska näkymäelementteihin tallennetaan vain CSS-

säännöt, joudutaan jokaisella ajokerralla rakentamaan korteille yhteinen mallipohja elementtipohjan avulla. Korttien ulkonäölliset erikoistukset siis tapahtuvat niiden mallielementtien näkymäelementeistä tulevien CSS-sääntöjen pohjalta.



Kuva 5.1 *Näkymän luonti arkkitehtuurilla*

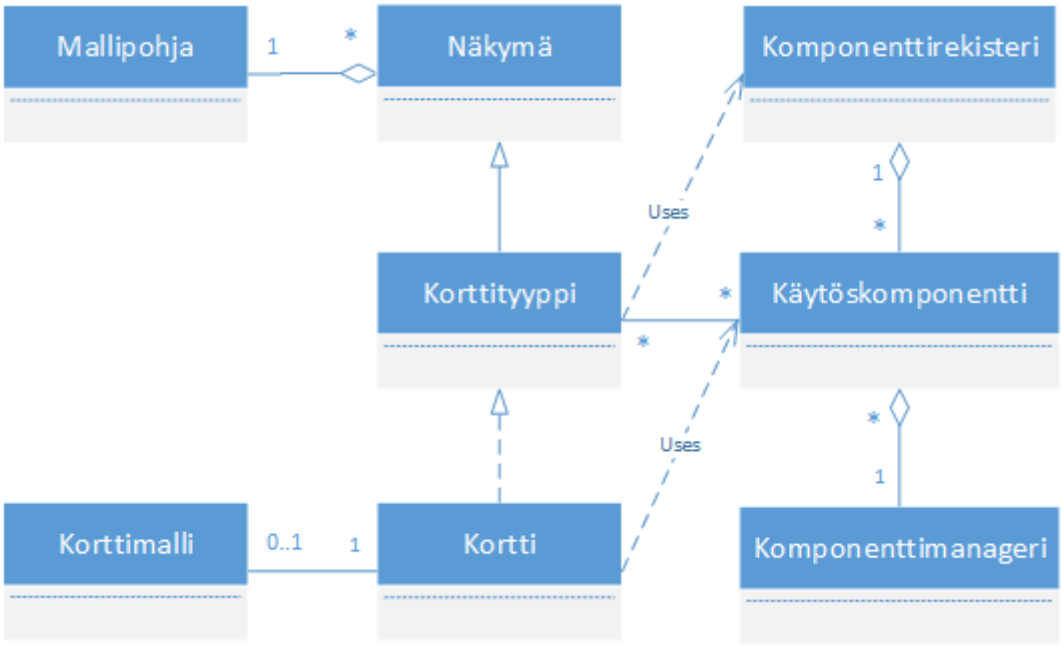
Kuvassa 5.1 on kuvattuna yksittäisen näkymän luonti arkkitehtuurilla. Kuten voidaan huomata, näkymän rakentaminen poikkeaa huomattavasti kohdassa 2.3 esitellystä tavasta.

Näkymän luonnin ensimmäisenä vaiheena on pyyntö näkymän avaukseen selaimesta. Tämän seurauksena palvelin pyytää näkymän rakentamiseen vaadittuja tietoja tietokannasta. Tietojen haun jälkeen palvelin piirtää pohjasivun ja asettaa valmiiksi mallielementtien näkymäelementeistä saadut CSS-säännöt osaksi pohjasivua. Tämä pohjasivu palautetaan selaimelle. Kuvasta poiketen mallielementtejä ei ole tässä vaiheessa vielä pakko toimittaa selaimeen, vaan sen sijaan voidaan selaimelle ilmoittaa mallielementtien sijainti. Ominaisuus toteutetaan, jotta sisällön dynaaminen hakeminen heikoilla verkkoyheyksillä mahdollistuu.

Kun selain saa mallielementtejä, lähetetään jokaisesta mallielementtityypistä pyyntö, jossa pyydetään mallielementin elementtipohjaa. Elementtipohjan noudon jälkeen rakennetaan niiden perusteella mallipohjat korttityypeille. Kun vaihe on suoritettu, aletaan rakentaa itse näkymään kuuluvia kortteja, jotka rakennuksen jälkeen piirretään selaimeen.

5.2 Käsitteellinen malli

Kuvassa 5.2 on esitettynä selaimen sovelluksissa käytettävän sovellusarkkitehtuurin käsitteellinen malli. Kuvassa esitellään arkkitehtuurin keskeiset käsitteet ja niiden väliset suhteet.



Kuva 5.2 *Arkkitehtuurin käsitteellinen malli*

Näkymä on kaikille korttityypeille yhteinen rakenne. Näkymän tehtävänä on toimia korttien visuaalisena osana. Näkymän rakenne määritellään Mallipohjan avulla. Korttityyppi on näkymän ja toiminnallisuuden yhteenliitos. Korttityypin pohjalta rakennetaan Kortteja, jotka voivat ovat sidoksissa Korttimalliin. Korttityyppien toiminnallisuus toteutetaan käytöskomponenteilla, joita korttityypeille tarjoilee Komponenttirekisteri. Jokaiselle käytöskomponentille on olemassa Komponenttimanageri.

Käsittellisessä mallissa esitellään vain keskeiset selainsovelluksien käsitteet. Käsitteitä käytetään arkkitehtuurissa vasta kohdassa 5.1 esitellyn mallipohjien rakentamisen

jälkeen.

5.2.1 Näkymä

Kortit ovat visuaalisia elementtejä, joihin on sidottu toiminnallisuutta. Korteista halutaan kyetä muuttamaan visuaalista esitysmuotoa siten, ettei se vaikuta kortin toiminnallisuuteen. Vaaditaan siis mekanismi, jolla voidaan tehokkaasti erottaa kortin visuaalinen osa toiminnallisuudesta.

Tämän ongelman ratkaisee Näkymä, joka vastaa työssä käytetyn MVC-arkkitehtuurimallin näkymää. Näkymä toteutetaan Backbone-kirjaston Näkymä-oliolla. Näkymä itsessään tarjoaa kaikille tietyn tyypin korteille yhteisen visuaalista puolta käsittelevän rajapinnan ja primitiivisiä menetelmiä visuaalisen puolen rakentamiseen. Näkymä myös tarjoaa sovellukselle tarvittavan piiroa hallitsevan rajapinnan.

Näkymä-luokalla on mahdollista erottaa kortin esitysmuodon hallinta kortin varsinaisesta toiminnallisuudesta. Näin kortin näkyvä osuus ja toiminnallisuus sidotaan löyhästi toisiinsa. Näin varmistetaan, että toisen puolen muokkaus vaikuttaa minimaalisesti toiseen puoleen.

5.2.2 Mallipohja

Näkymä vastaa kortin visuaalisen esityksen vaatimien asetuksien säilömisestä sekä kortin piirtämisestä käyttäjälle. Näkymä itsessään ei ota kortin HTML-merkkaukseen kantaa, eli näkymä ei luo kortille DOMia. Näkymälle voidaan määrittää kortin rakenteen määrittävä mallipohja. Mallipohja ratkaisee ongelman, jossa näkymän hallinta on vahvasti sidottuna näkymän rakenteeseen.

Mallipohja määrittelee näkymän konkreettisen HTML-merkkauksen. Mallipohja koostetaan tietokannasta saatavalla mallielementin näkymädatalla eli näkymäelementillä, jossa on määriteltynä halutulle korttityypille yleinen esitysmuoto. Luonnollisesti mallipohjan koostamista käytetään myös mallielementin metamallin dataa hyväksi. Näin voidaan määritellä mallielementin näkymäelementissä näytettävät attribuutit sekä valita niille oikeat esitysmuodot.

Koostettu mallipohja voidaan täydentää mallielementin datalla ajonaikaisesti siten, että kortin näkymässä näkyvät valitut mallielementin tiedot.

5.2.3 Korttityyppi

Korteille halutaan visuaalisen puolen lisäksi toiminnallisuutta. Kuitenkaan toiminnallisuutta ei haluta sitoa kiinteästi näkymiin. Vaatimuksissa mainitaan myös, että korttimekanismin tulee olla yleinen ja helposti muokattava. Korttityyppi luodaan ratkaisemaan edellämainitut ongelmat.

Yksi ratkaistava ongelma on toiminnallisuuden sitominen kortteihin. Korttien toiminnallisuuksia halutaan yleistää käyttötapausten yleisyyden mukaan. Toisaalta tämä on ristiriidassa suorituskyvyn vaatimusten kanssa, joissa vaaditaan pientä muistijälkeä korttien osalta. Yleisiä toiminnallisuuksia ei täten haluta sitoa osaksi itse korttia siten, että toiminnallisuudet ovat aina kortissa mukana, vaikkei niitä sovelluksissa käytettäisikään.

Korttityyppien avulla toiminnallisuutta ei tarvitse vahvasti sitoa näkymiin, vaan toiminnallisuus koostetaan osaksi näkymää käyttämällä toiminnallisuutta määritteleviä käytöskomponentteja.

5.2.4 Kortti

Kortti on näkymän ja toiminnallisuuden yhteensidottu konkreettinen entiteetti. Kortti edustaa aina jotain korttityyppiä. Kortin tehtävänä tietosisältönsä ja näkymänsä ajonaikainen hallinta. Kortti myös vastaa käyttäjän tekemiin syötteisiin sille määriteltyjen käytöskomponenttien mukaan.

Kortit ovat sovelluksen ainoa osa, jonka sovelluksen käyttäjä sovelluksesta näkee. Kortti täten sisältää kaikki vaaditut interaktiot käyttäjän syötteisiin. Voidaankin yleistää, että kortti on sovelluksien käyttöliittymien peruskomponentti.

Koska kortit voivat olla hierarkkisia, kortti voi sisältää muita kortteja. Hierarkkisissa korteissa isäntä- ja lapsikortit voivat olla eri korttityyppiä.

5.2.5 Korttimalli

Korttimalli on kortin konkreettinen tietosisältö ja vastaa työssä käytetyn MVC-arkkitehtuurimallin mallia. Korttimalli toteutetaan käyttämällä Backbone-kirjaston Malli-oliota. Korttimalli tarjoaa yksinkertaisen rajapinnan kortin tietojen lukemiseen ja päivitykseen. Sovelluksen rakenteessa korttimallien tietosisällöt vastaavat mallielementtejä, joita palvelin selaimelle tarjoaa.

Korttimalli vastaa mallidatan asianmukaisesta säilömisestä selaimessa sekä toimintojen aiheuttamien, mallielementtiä päivittävien operaatioiden välittämisestä palvelimelle. Näin saadaan eristettyä sovelluksen kommunikaatio palvelimelle omaan moduuliinsa.

5.2.6 Komponenttirekisteri

Komponenttirekisteri on tietosäiliö, joka tarjoaa rajapinnan komponenttien lisäämiseen ja hakemiseen. Komponenttirekisterin tarkoitus on säilöä kaikki sovelluksessa käytettävät komponentit, ja rekisteristä sovellus voi siten hakea korttityypin toiminnallisuuksien vaatimat komponentit.

Komponenttirekisteri populoidaan sovelluksen alustusvaiheessa sovelluksen vaatimilla komponenteilla. Populointia varten komponenttirekisteri tarjoaa rajapinnan, jonka avulla komponenttirekisteriin voidaan määrittää komponentteja.

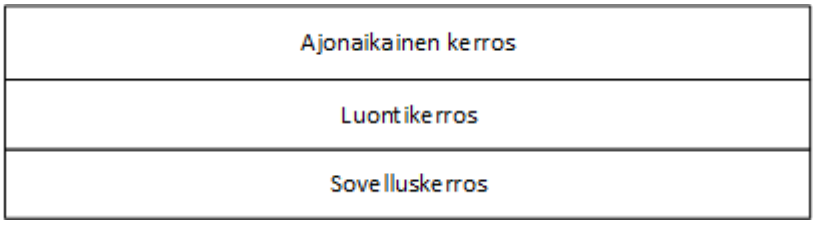
5.2.7 Käytöskomponentti

Käytöskomponentit ovat itsenäisiä ohjelmakomponentteja, joiden tehtävänä on tarjota korteille käyttäytymistä. Käytöskomponentit ovat olioita, joiden pääasiallinen tehtävä on määrittää rajapinta toiminnallisuuden toteuttamiseksi. Käytöskomponenteille on määritelty oletustoiminnallisuutta, mutta tarvittaessa komponentin toiminnallisuus voidaan ylikuormittaa sovelluksen niin vaatiessa. Käytöskomponenttien toimintamekanismeja voidaan käyttää myös muissa kohteissa kuin korteissa. Esimerkiksi malleille voisi tehdä omia komponentteja, joilla voisi käsitellä mallien sisältämiä tietoja.

Käytöskomponentit ovat tapoja pienentää sovelluksen muistijälkeä siten, että korttien toiminnallisuudet tulevat kaikille korteille yhteisistä komponenteista. Komponentit myös mahdollistavat sovelluksen ladattavan ohjelmakoodin määrän pienentämisen siten, että sovelluksen latauksen yhteydessä ladataan vain sovelluksen vaatimat komponentit.

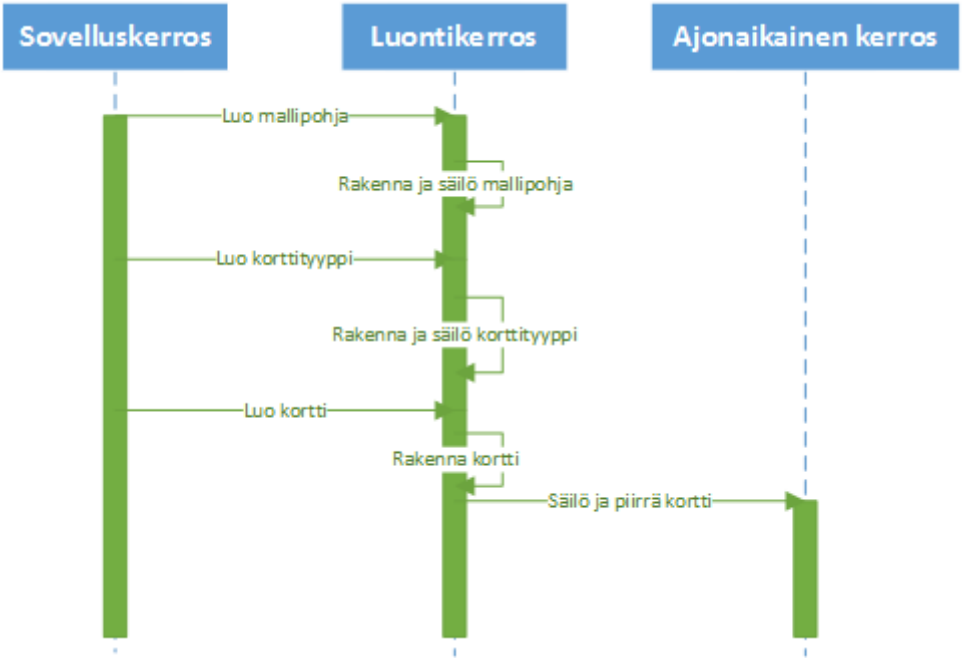
5.3 Selainsovellusarkkitehtuuri

Selainsovellusarkkitehtuuri rakenteellisesti jakautuu kolmeen kerrokseen. Kerroksista alimpana on sovelluskerros. Keskikerros on luontikerros, ja päällimmäisenä on ajonaikainen kerros. Arkkitehtuurin kerrosmalli on kuvattu kuvassa 5.3.



Kuva 5.3 *Suunniteltu sovellusarkkitehtuurin kerrosmalli*

Kerrosten tarkoitus on erotella arkkitehtuurista selkeästi eri abstraktiotasolla toimivat osat toisistaan. Kuvassa kerroksen abstraktiotaso laskee noustaessa kerroksissa. Täten sovellustason abstraktiotaso on korkein, ja ajonaikaisen kerroksen abstraktiotaso on matalin.



Kuva 5.4 *Kerrosten välinen kommunikointi*

Kerrosten välinen kommunikaatio tapahtuu kuvan 5.4 osoittamalla tavalla. Kuvassa myös hahmotellaan, kuinka kerrosten vastuualueet jakautuvat.

5.4 Sovelluskerros

Sovellusarkkitehtuurin perustana toimii Sovelluskerros. Sovelluskerroksen ideana on kääriä matalammat abstraktiotasot itseensä ja täten kapseloida yksittäinen sovellus yhden olion sisään. Sovelluskerros tarjoaa yksinkertaisen rajapinnan itse sovelluksen

rakentamiseen sekä sen ajonaikaiseen hallintaan. Sovellukset, ja täten sovelluskerros, otetaan käyttöön vasta kohdassa 5.1 esitellyn pohjasivun siirron jälkeen.

Sovelluskerros mahdollistaa useamman sovelluksen ajamisen samalla web-sivulla. Näin arkkitehtuuria voisi käyttää suurempien ja huomattavasti monimutkaisempien ohjelmien koostamiseen pienemmistä sovelluksista, jotka toimivat itsenäisesti.

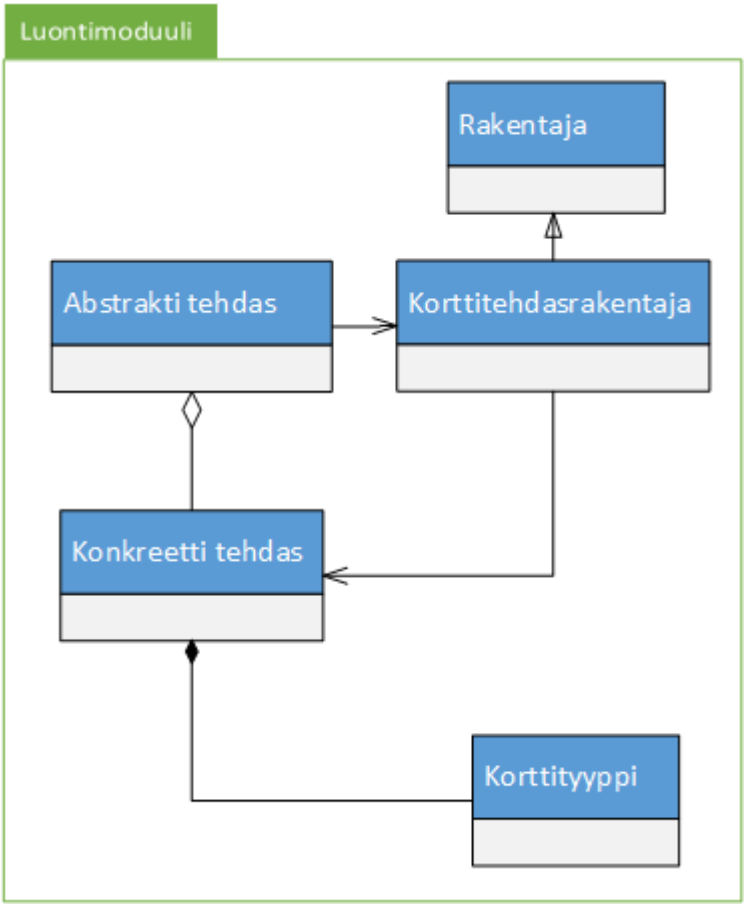
Sovelluskerros on vastuussa sovelluksen ajamisesta ja luontikerroksen ajonaikaisesta hallinnasta. Taso voidaan toteuttaa siten, että sovelluksen mallielementit voidaan injektoida sovellukseen palvelimen tuottaman pohjasivun yhteydessä, tai vaihtoehtoisesti sovellus voi itse noutaa vaaditut mallielementit sille annetun verkko-osoitteen perusteella.

Datan sisällyttäminen sovellukseen tapahtuu käyttämällä Strategia-suunnittelumallia, jolla valitaan oikea funktio joko noutamaan tiedot ulkoisesta lähteestä tai ottamaan vastaan injektoitu data. Strategia-suunnittelumallilla datan sisällyttämistä ei rajoiteta edellämainittuihin menetelmiin, vaan suunnittelumalli mahdollistaa myös muut datan sisällyttämisen menetelmät.

Sovelluskerroksen tarkoitus on luoda sovelluksesta yhtenäinen komponentti käyttäjälle. Kerroksen toteuttaman rajapinnan avulla voidaan yhdelle sivupohjalle upottaa monta eri sovellusta toimimaan samanaikaisesti. Arkkitehtuurilla luodut sovellukset eivät siis ole pelkästään yksittäisen sivun sovelluksia (**SPA**) [14], vaan sivut voidaan koostaa monista erillisistä sovelluksista.

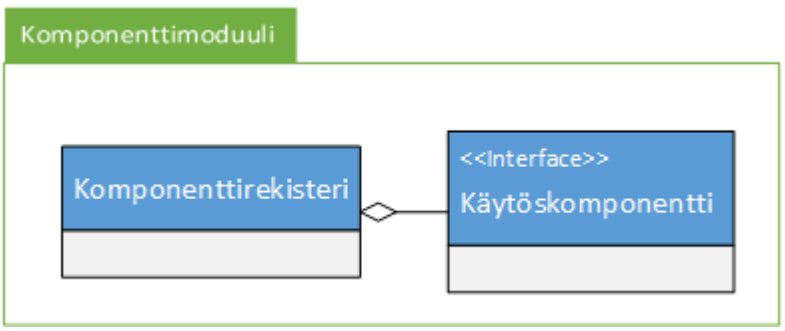
5.5 Luontikerros

Sovelluksen varsinainen toiminnallisuus rakennetaan luontikerroksessa. Luontikerros koostuu kolmesta erillisestä moduulista. Kuvassa 5.5 on kuvattuna luontimoduuli. Luontimoduulin vastuulla on muodostaa sovelluksen parametrien pohjalta vaadittavia kortteja tuottavat tehtaat. Korttitehtaat luodaan dynaamisesti siten, että tehtaan luonnin yhteydessä annettujen asetusten perusteella rakennetaan asetusten mukaistia kortteja tuottava tehdas Rakentaja-suunnittelumallin mukaisesti. Luontikerroksen ensimmäinen tehtävä arkkitehtuurissa on kohdassa 5.1 esitetty mallipohjien rakennus. Tämän lisäksi kerros myös rakentaa korttitehtaat ja kortit.



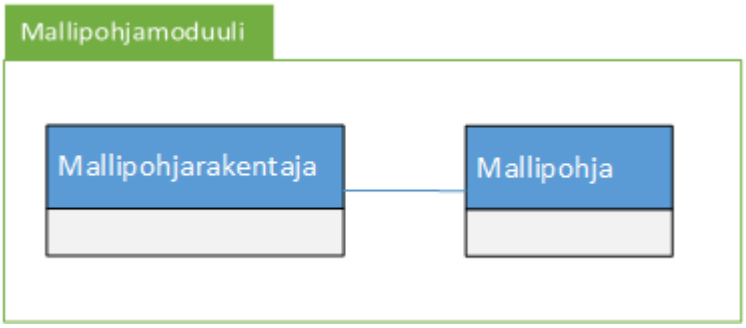
Kuva 5.5 *Luontimoduulin keskeiset luokat*

Kuvassa 5.6 on kuvattuna komponenttimoduuli. Komponenttimoduulin vastuulla on tarjota korttityypeille toiminnallisuutta. Komponentit sidotaan kortteihin korttityyppien rakennuksen yhteydessä. Moduuli koostuu komponenttirekisteristä, jonka vastuulla on säilöä ja tarjoilla komponentteja. Komponentit taasen sisältävät toiminnallisuuden, joka toteutetaan sitomalla se korttiin. Komponentteja käytetään konkreettisesti vain ajonaikaisella tasolla.



Kuva 5.6 *Komponenttimoduulin keskeiset luokat*

Viimeinen luontikerroksen moduuli on mallipohjamoduuli. Mallipohjamoduulin vastuulla on rakentaa sille annetun elementtipohjan perusteella kortille HTML-muotoinen merkkaus. Mallipohjat liitetään kortteihin korttityypin rakennuksen yhteydessä. Moduuli koostuu mallipohjarakentajasta, jonka vastuulla on rakentaa mallipohja sille annetun elementtipohjan perusteella. Mallipohja taasen on vastaa elementtipohjasta rakennettua HTML-merkkuasta, joka voidaan asettaa korttityypille mallipohjaksi. Kortin piirron yhteydessä mallipohja täytetään kortin korttimallin datalla. Moduuli on kuvattuna kuvassa 5.7.



Kuva 5.7 *Mallipohjamoduulin keskeiset luokat*

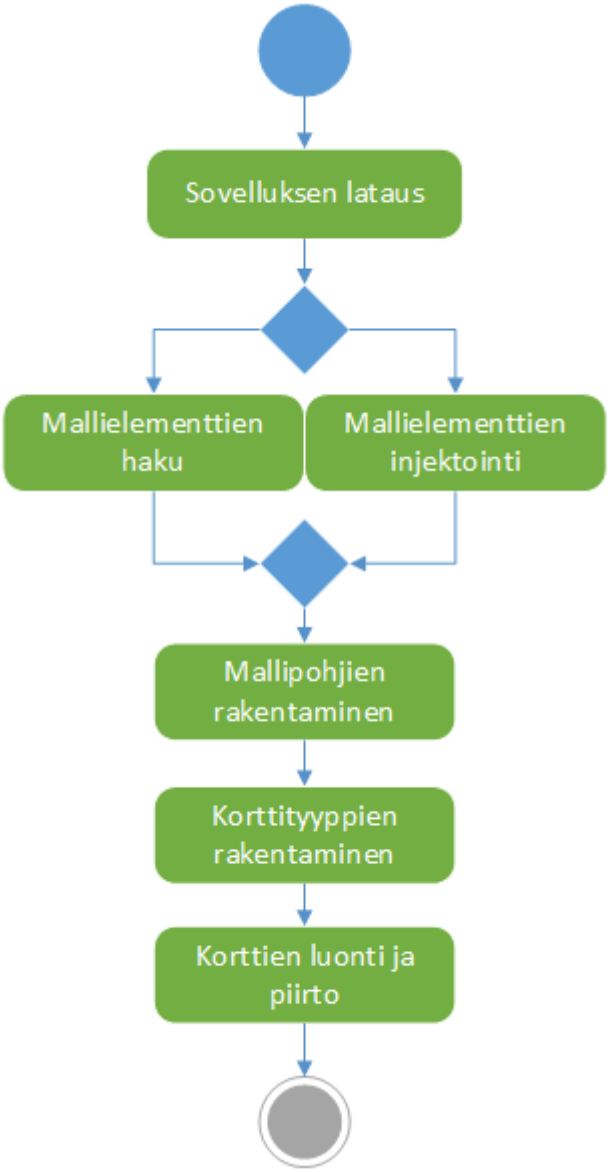
5.5.1 Sovelluksen luonti

Kuvassa 5.8 kuvataan sovelluksen rakentamisen eri vaiheet. Kaikkien vaiheiden tulee olla suoritettuna ennen kuin varsinaisen sovelluksen käyttö voi alkaa.

Vaiheissa ensimmäisenä on sovelluksen lataus. Sovelluksen latauksessa sovelluksen uusin versio ladataan käyttäjän selaimeen. Latausta nopeuttamaan luodaan arkkitehtuurin pysyvistä osista oman erillinen ja kaikille toteutetuille sovelluksille yhteinen ydin. Ydin voidaan sijoittaa välimuistiin, joko palvelimen päässä tai selaimessa HTML5-rajapintoja hyväksikäyttäen, sovelluksen lataamisen nopeuttamiseksi.

Seuraavassa vaiheessa on sovelluksen ajamiseen vaadittavien mallielementtien hankinta. Tämä voidaan toteuttaa joko noutamalla mallielementit ulkoisesta lähteestä, tai injektoimalla ne sovelluksen alustuksen yhteydessä.

Mallielementtien lataamisen jälkeen seuraa sovelluksen mallipohjien rakennus. Sovelluksen mallielementeille rakennetaan mallipohjat niiden elementtipohjia hyväksikäyttäen. Mallipohjien rakennuksen jälkeen voidaan rakentaa korttityyppi. Tässä vaiheessa korttityyppiin liitetään mallipohja sekä toiminnallisuus käytöskomponenteilla. Korttityyppien luonnin jälkeen voidaan rakentaa mallielementeille kortit, ja piirtää ne käyttäjän nähtäväksi selaimeen.



Kuva 5.8 *Sovelluksen käynnistyksen toiminnot*

5.5.2 Komponenttirekisteri

Sovelluksessa komponentit sijoitetaan komponenttirekisteriin. Rekisterin tehtävänä on sekä kääriä käytöskomponentti-moduuli yksittäiseksi komponentiksi että tarjota näkymille komponenttiolioita tarpeen vaatiessa. Komponenttirekisteri tarjoaa myös rajapinnan ajonaikaisesti uusien komponenttien luomiseen.

Komponenttirekisterin ei tarvitse välittömästi rakentamisen yhteydessä sisältää konkreettisia komponentti-olioita. Komponentit voidaan ladata komponenttirekisteriin ajonaikaisesti laiskan lataamisen periaattein.

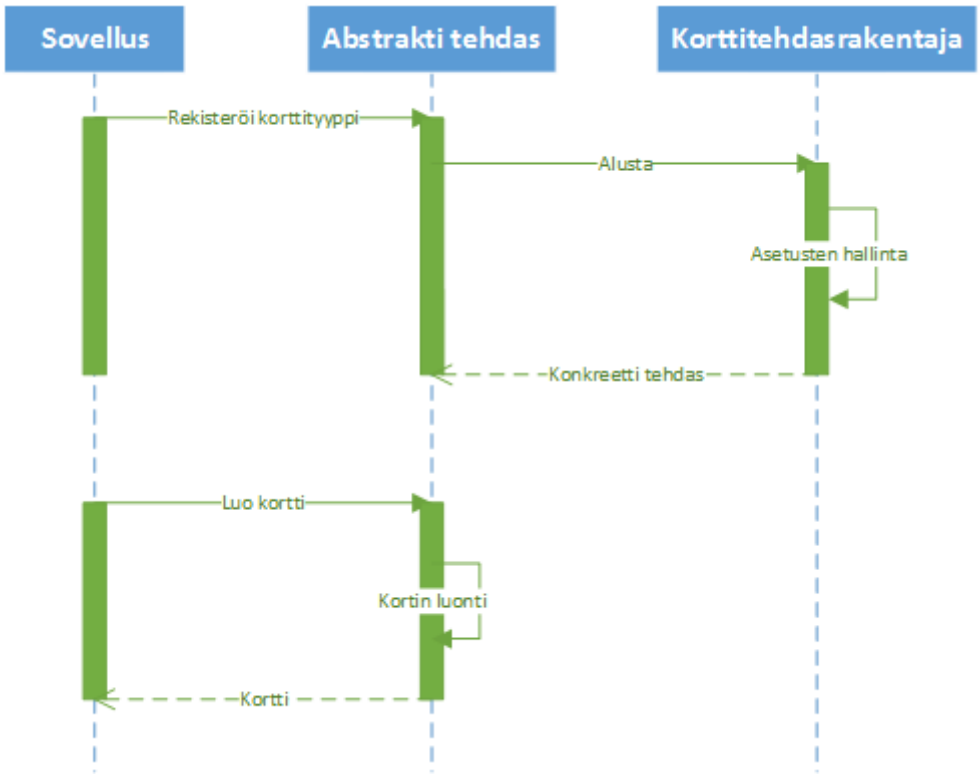
Tämä laiska lataaminen ulkopuolisesta komponenttilähteestä luonnollisesti lisää

HTTP-pyyntöjen määrää ja näin omalta osiltaan kasvattaa sovelluksen rakentamisen vaatimaa aikaa. Hyödyt laiskasta komponenttien lataamisesta syntyvät sovellusten ja käytettyjen komponenttien määrien kasvaessa. Näin saavutetaan myös tilanne, jossa varsinainen arkkitehtuurin vaatimat kriittiset ohjelmakoodit voidaan tarjoilla suoraan palvelimen välimuistista.

5.5.3 Korttityypin ja kortin rakentaminen

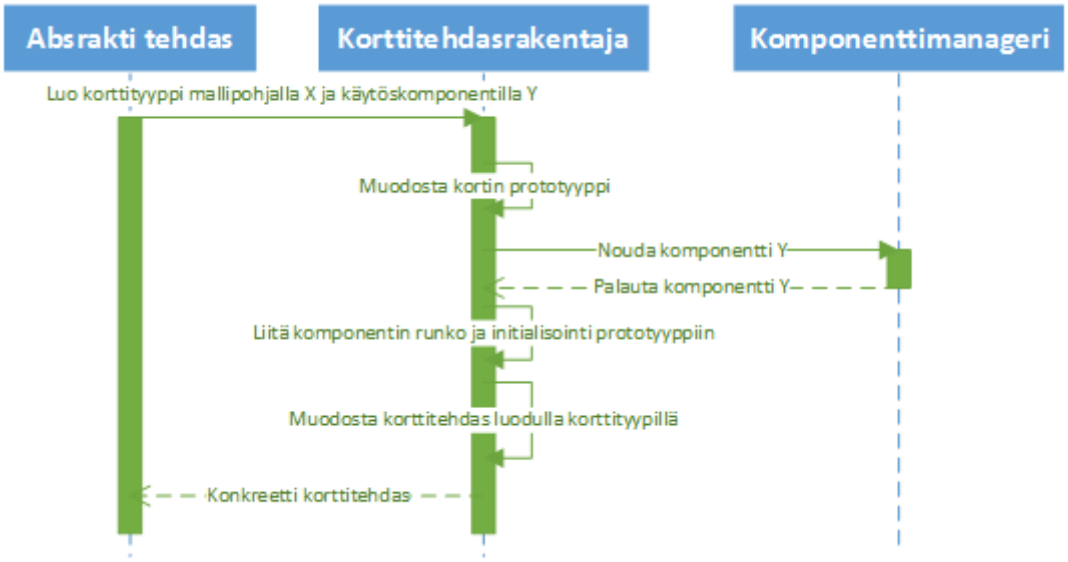
Kuvassa 5.9 on kuvattu yleiset vaiheet, jotka vaaditaan korttien rakentamiseen sekvenssikaavion muodossa. Kortin rakentamisessa ensimmäinen vaihe on kutsua sovelluksesta abstraktin tehtaan tehdastyypin rekisteröintimetodia luotavan korttityypin asetuksilla. Abstrakti tehdas kutsuu korttitehtaan rakentajaa, joka asetusten perusteella rakentaa korttitehtaan. Korttitehtaan rakentaja palauttaa abstraktille tehtaalle tehdas-olion, joka tuottaa määriteltyä korttityyppiä. Abstrakti tehdas säilöö luodun tehdas-olion omaan tietorakenteeseensa. Luominen tapahtuu kutsumalla abstraktin tehtaan luomis-metodia. Luomismetodi käyttää haluttua tehdasta tuottamaan ja paluttamaan kortin.

Kortit rakennetaan aina Abstrakti tehdas -suunnittelumallin mukaisissa olioissa. Tehtaat säilötään Abstraktin tehtaan sisään, joka annetun mallidatan perusteella valitsee oikean tehtaan tuottamaan kortin.



Kuva 5.9 *Korttien rakentaminen yleisesti*

Kuvassa 5.10 on kuvattu korttityypin rakentamisen vaiheet sekvenssikaavion muodossa. Korttitehtaan rakentaminen tapahtuu abstraktin tehtaan tekemän kutsun kautta. Kutsussa käsketään korttitehtaita tuottavan rakentajan rakentamaan tehdas tuottamaan korttia. Kutsussa myös määritellään kortin käyttämä elementtipohja ja mahdolliset käytöskomponentit. Korttitehdasrakentaja muodostaa korttityypin prototyyppi Backbonen View-oliosta. Tämän jälkeen elementtipohjan perusteella generoidaan kortin HTML-merkkaus ja asetetaan se osaksi prototyyppiä. Käytöskomponentit noudetaan komponenttimanagerilta, ja niiden rungot sekä alustus-funktiot lisätään osaksi prototyyppiä. Tämän jälkeen muodostetaan konkreetti tehdas tuottamaan muodostettua korttityyppiä. Rakentajan luoma konkreetti tehdas palautetaan abstraktille tehtaalle säilömistä varten.



Kuva 5.10 *Korttitehtaan rakentaminen*

5.6 Ajonaikainen kerros

Ajonaikaisen kerroksen vastuulla on sovelluksen ajonaikaisten toiminnallisuuksien toteuttaminen. Kerros koostuu korteista ja niiden käyttämistä komponenteista. Käyttäjän kaikki toiminnallisuus tapahtuu ajonaikaista kerrosta vasten. Ajonaikainen kerros aktivoituu ensimmäisen kortin piirrossa tämä tapahtuu kohdassa 5.1 määritellyssä korttien piirrossa.

5.6.1 Kortit

Kortit rakennetaan Backbone-kirjaston View-luokan päälle. Kortit ovat arkkitehtuurilla rakennetun sovelluksen keskeisin termi, joissa yhdistyvät sekä mallielementti, elementtipohja, näkymäelementti että kaikki kortille määritelty toiminnallisuus. Kortin rakenne on kuvattu kuvassa 5.11.

Kuva 5.11 *Kortin rakenne*

Kortteihin toteutetaan toiminnallisuus asettamalla korttityypin prototyyppiin viitteet haluttuihin käytöskomponentteihin. Ratkaisu vähentää sovelluksen muistinku-

lutusta huomattavasti siten, että kortit eivät itsessään sisällä toiminnallisuutta määrittteleviä komponentteja. Kaikki kortit siis voivat jakaa saman käytöskomponentin toteuttamaan haluttu toiminnallisuus.

Kortin luominen tapahtuu kuvan 5.12 osoittamalla tavalla. Kortti rakennetaan antamalla sille korttimalli, jota kortti edustaa. Kortti ensin säilöö itseensä sille välitetyn korttimallin. Tämän jälkeen kortti alustaa sille määrätyt käytöskomponentit ajamalla niiden alustus-metodit. Tämän jälkeen kortissa täytetään sille määritelty mallipohja korttimallin tiedoilla. Näin kortti luo itselleen varsinaisen HTML-merkkauksen.

Kuva 5.12 *Kortin luonti*

5.6.2 Korttien erikoistaminen

Kortteja voidaan erikoistaa korttityyppien asetuksia määrittelemällä. Asetuksilla voidaan laajentaa korttien toiminnallisuutta, ja sille voidaan määritellä korttityyppikohtaisia käytöskomponentteja, joilla voidaan toteuttaa sovelluskohtaisia toiminnallisuuksia. Näin sovelluksen toiminnallisuus voidaan luoda kokonaan korttipohjaisina käyttöliittymäelementteinä. Asetusdatalla tehtävä erikoistaminen on ensisijaisesti varattu käyttötapauksille, jotka katsotaan olevan liian sovelluskohtaisia ja vaikeasti yleistettäviä.

Kortin ulkoasu määräytyy oletuksena sen muodostaman mallipohjan mukaan. Kuitenkin korttien ulkoasua täytyy voida erikoistaa määrittelemällä sille CSS-sääntöjä. Kortin CSS-säännöt tallennetaan osaksi sen edustaman mallielementin näkymäelementiksi. Näin voidaan toteuttaa yksittäisten korttien ulkonäön erikoistamista. Tätä varten Realm-järjestelmän mukana tulleessa pohjasivussa on mukana kortteja erikoistavia CSS-sääntöjä. Näkymäelementtien CSS-säännöt sidotaan kortteihin korttien HTML-merkkauksen ID-attribuutin avulla.

5.6.3 Käytöskomponentit

Käytöskomponentit ovat itsenäisiä ohjelmakomponentteja, jotka tuovat lisätoiminnallisuutta olioihin. Käytöskomponentit ovat oma, sovellustason päälle rakennettu moduulinsa, joka koostuu käytöskomponenteista ja käytösmanagereista. Käytöskomponenteilla voidaan toteuttaa yleisiä toimintoja näkymäelementeille. Esimerkiksi näkymäelementtien raahaaminen, sisällön piilottaminen ja viitteiden tekeminen on mahdollista toteuttaa itsenäisinä komponentteina.

Komponentit mahdollistavat ajonaikaiset näkymätyyppien erikoistamiset dynaamisella entiteetti-komponentti-järjestelmä -arkkitehtuurimallin käytöllä. Käytöskomponentit vastaavat arkkitehtuurimallin komponentteja ja kortit vastaavat entiteettejä.

Käytöskomponentit ladataan käyttäjän selaimeen ainoastaan sovelluksen niin vaatiessa. Tästä syystä käytöskomponentit ajetaan anonyymissä funktiossa välittömästi latauksen jälkeen. Funktion tarkoitus on säilöä komponentti komponenttirekisteriin. Kuvassa 5.13 on kuvattu erään käytöskomponentin ohjelmakoodi.

```
(function () {
    "use strict";
    var component = {
        initializeComponent: function () {
            var event = 'focus input',
                proto = Object.getPrototypeOf(this);

            if (!proto.events[event]) {
                proto.events[event] = '_toggleActive';
            }
        },
        body: {
            _toggleActive: function (event) {
                event.stopImmediatePropagation();
                event.preventDefault();
                var activeClass = this.components.active.options.activeClass;
                $('.' + activeClass).toggleClass(activeClass);

                this.$el.toggleClass(this.components.active.options.activeClass);
            }
        },
        manager: {}
    };

    window.ComponentStorage.addComponent('active', component);
})();
```

Kuva 5.13 *Active-komponentin ohjelmakoodi.*

Kuvassa nähdään käytöskomponentin ohjelmakoodin koostuvan alustusmetodista (initializeComponent), rungosta (body) ja sitä hallitsevasta komponenttimanagerista (manager). Esitetyssä käytöskomponentissa ei vaadita komponenttimanageria välittämään tietoa korttien välillä.

5.6.4 Komponentti-olio

Komponentti-olio tarjoaa yksinkertaisen rajapinnan komponentin käyttöön. Komponentti-olio rakentuu kolmesta osasta: alustusmetodista, rungosta sekä mahdollisesta käytösmanagerin ohjelmakoodista.

Alustusmetodi kertoo, kuinka komponentti alustetaan osaksi korttia. Funktiossa määritellään, mitä tapahtumia ja mistä DOM-elementistä komponentti kuuntelee. Tapahtumat on sidottu komponentin rungossa määriteltyihin funktioihin.

Komponentin runko on varsinainen kortin käytöksen luovan osuus. Olio sisältää kaikki komponentin toiminnan vaatimat funktiot. Tämä tekee komponentista itsenäisen. Komponentin rungon funktiot toimivat vasteina initialisaatio-funktion mää-

rittelemille DOM-tapahtumille.

Käytösmanageri on olio, joka periytetään Backbone-kirjaston Events-oliosta, joka on toteutus Tarkkailija-suunnittelumallista. Käytösmanageri mahdollistaa komponenttien keskustelun muiden komponenttien kanssa. Käytösmanagerin tarkoitus on kuunnella sille määritellyistä näkymäelementeistä komponenttien laukaisemia tapahtumia. Näillä tapahtumilla on mahdollista pitää kirjaa siitä, mitä toimintoja tietyn komponentin omaavilla olioilla tapahtuu, ja tarvittaessa reagoida niihin manageriin määriteltyjen toiminnallisuuksien mukaisesti.

Käytösmanageri myös välittää tietoa erityyppisten komponenttien väleillä. Tämä mahdollistaa erityyppisten komponenttien yhteentoimivuuden. Esimerkiksi raahaus ja tiputus ovat erillisiä itsenäisiä komponentteja, mutta varsinaisen ajonaikaisen toiminnallisuuden kannalta kummankin täytyy kyetä keskustelemaan. Käytösmanagerin tehtävänä raahauksen ja tiputuksen käyttötapauksessa on kuunnella raahattavien näkymäelementtien raahaus-tapahtumaa, asettaa muistiin raahattava näkymäelementti ja vapauttaa näkymäelementti tiputus-tapahtuman tapahtuessa. Käytösmanageri luodaan mahdollistamaan edellämainittu käyttötapaus.

6. ARKKITEHTUURILLA TOTEUTETUT SOVELLUKSET

Arkkitehtuurin kehityksen yhteydessä toteutettiin muutama sovellus jo olemassaolevien sovellusten tilalle. Olemassaolevat sovellukset eivät noudattaneet yhteistä arkkitehtuuria ja tämä olikin suurin syy suunnitella ja toteuttaa työssä toteutettu sovellusarkkitehtuuri.

Kappaleessa käydään ensin läpi sovelluksien toteutuksiin vaadittavat komponentit ja toiminnalliset vaatimukset, joihin toteutetuilla komponenteilla pyritään vastaamaan. Näiden jälkeen käsitellään komponenteista koostamalla toteutetut sovellukset.

6.1 Toteutetut komponentit

Toteutetut komponentit luotiin täyttämään liitteessä A määritellyt sovellusten toiminnalliset vaatimukset. Suurin osa vaatimuksista toteutettiin sovelluksiin arkkitehtuurin mukaisilla käytöskomponenteilla. Samankaltaiset tai samaa toiminnallisuutta toteuttavat vaatimukset toteutettiin useimmiten samassa komponentissa.

6.1.1 Alinäkymä-komponentti

Vaatimus **T01** toteutettiin omassa **Subview**-nimisessä komponentissaan. Komponentin tarkoitus on jakaa kortti omiin alueisiinsa, joissa voidaan näyttää muita kortteja. Komponentti alustetaan korttiin siten, että sille määritellään korttityypin asetuksissa halutun alinäkymän säiliön DOM-elementin CSS-valitsin. Annettu CSS-valitsin toimii myös kortin kyseisen alinäkymän tunnisteena. Alinäkymiin voidaan ajonaikaisesti lisätä tai poistaa kortteja komponentin kortille määrittelemän rajapinnan avulla.

Subview-komponenttille voidaan myös asettaa valitsinkohtaisesti asetukset alinäkymän korttien järjestämiseen. Järjestämistä varten täytyy asetuksissa määritellä

kortin mallidatasta tietoalkio, jonka mukaan alinäkymä järjestetään. Tämän lisäksi määritellään järjestyksen suunta ja mahdollinen mukautettu järjestysmetodi. Tämä toiminnallisuus täyttää vaatimuksen **T05**.

6.1.2 Menu-komponentti

Vaatimus **T03** ja **T04** on toteutettu **Menu**-nimisessä komponentissa. Menun idea on liittää korttiin oma menunsa, josta tarjotaan oikotiet korttien toiminnallisuuksiin. Menun asetuksissa määritellään CSS-valitsin, jonka osoittamaa elementtiä klikkaamalla voidaan menu näyttää tai piilottaa.

Luonnollisesti mahdollisuus uusien korttien tai alikorttien luomiseen on mahdollista, vaikkakin oletusmetodeita toiminnallisuuksille ei olla toteutettu. Oletusmetodit vaativat tarkemmin määritellyn palvelinpuolen rajapinnan, jota ei tämän työn yhteydessä vielä päätetty tehdä.

6.1.3 Poistettava-komponentti

Vaatimus **T07** toteutettiin omassa **Deletable**-nimisessä komponentissa. Komponentti mahdollistaa kortin poistamisen järjestelmästä siten, että komponentille määritellään alustuksen yhteydessä CSS-valitsin, jonka mukaista elementtiä klikkaamalla tapahtuma välittyy komponentille käsiteltäväksi.

Käyttäjän tapahtuma käsitellään komponentissa siten, että komponentti kutsuu kortin korttimallin poistamismetodia. Tämä lähettää palvelimelle pyynnön kyseisen korttimallin poistamiselle. Korttimallin poistamisen jälkeen komponentti poistaa kortin DOM-rakenteesta ja laukaisee kortin poistamiseen liittyvät tapahtumat, jonka mukaan muut komponentit ja sovellus itsessään reagoivat kortin poistamiseen toteuttamalla poistamisesta johtuvat pakolliset toiminnot.

6.1.4 Minimointi-komponentti

Vaatimus **T08** toteutettiin omassa **Minify**-nimisessä komponentissa. Komponentti mahdollistaa kortin jonkin osan piilottamisen ja esittämisen käyttäjän syötteiden mukaan. Kortin alustuksen yhteydessä annetaan komponentille CSS-valitsin, jonka mukaista elementtiä klikkaamalla tapahtuma välittyy komponentille käsiteltäväksi. Muita asetuksia ovat minimoitavan elementin CSS-valitsin, kortin alkutila ja mahdolliset luokat klikattavan elementin aktivoinnin näyttämiseksi. Alkutila määrittelee, onko minimoitava osa kortista oletuksena näkyvissä vai ei.

Käyttäjän klikkaus käsitellään komponentissa siten, että komponentti itsessään käsittelee käyttäjän klikkauksen ohjaamalla syötteen komponentin määrittelemään rajapintaan. Rajapinnan toteutuksen avulla voidaan kytkeä haluttu tila minimoitavalle elementille. Toiminnosta laukaistaan tapahtumat, jotka voidaan havaita muissa komponenteissa ja näin suorittaa tapahtumasta aiheutuvat muiden komponenttien pakolliset toiminnot.

6.1.5 Hierarkkia-komponentti

Vaatimus **T09** toteutettiin omassa **Hierarchy**-nimisessä komponentissaan. Komponentti mahdollistaa korttien hierarkkioiden hallitsemisen. Komponentti ei vaadi alustustietoina mitään ja toimii vain rajapintana korttien lisäämiseksi ja poistamiseksi halutun kortin hierarkkiaan.

Rajapinta siis tarjoaa ohjelmallisen tavan muokata kortin sisältämää hierarkkiaa. Kortti voidaan lisätä joko isäntäkortiksi tai lapsikortiksi. Toiminnot laukaisevat kortissa tapahtumia, joita voidaan havaita muissa komponenteissa, ja niiden pohjalta voidaan toteuttaa erillistä toiminnallisuutta.

6.1.6 DragAndDrop-komponentti

Vaatimus **T10** toteutettiin omassa **DragAndDrop**-nimisessä komponentissaan. Komponentti mahdollistaa korttien raahaamisen ja tiputtamisen toisten korttien sisälle. Toteutettu komponentti oli työssä toteutetuista komponenteista huomattavasti monimutkaisin, ja se koostuukin kahdesta erillisestä komponentin rungosta, joista toinen määrittelee kortin raahattavaksi ja toinen tiputettavaksi. Kortissa käytettävä runko valitaan korttityypin asetusten perusteella. Luonnollisesti kortti voidaan koostaa molemmista rungoista, jolloin kortti on sekä raahattava että sisältää alueen, jonne muita kortteja voidaan tiputtaa.

Raahattavan kortin korttityypin asetuksissa määritellään CSS-valitsin, josta korttia voidaan raahata. Muita asetuksia raahattavalle kortille ei vaadita. Tiputettavalle kortille määritellään korttityypin asetuksissa CSS-valitsin, joka määrittelee kortin rakenteesta tiputusalueen raahattaville korteille.

Toiminnallisuus toteutetaan käytännössä komponentin komponenttimanagerissa, joka kuuntelee korteista tapahtumia. Managerin tarkoitus on pitää kirjaa raahattavasta kortista sekä luoda ja raahata osoitinta, jolla indikoidaan käyttäjälle raahauksen

aikana kohta, jonne kortti on sillä hetkellä tippumassa. Tiputuksen yhteydessä laukaistaan kortissa tapahtuma, jonka avulla muut komponentit voivat toteuttaa tiputuksesta aiheutuvat toiminnot. Näitä toimintoja ovat esimerkiksi kortin siirtäminen näkymästä toiseen sekä hierarkkian päivittäminen.

6.1.7 Viitteet-komponentti

Vaatimukset **T11**, **T12** ja **T13** toteutettiin omassa **VisualReferences**-nimisessä komponentissaan. Komponentin tehtävänä on mahdollistaa visuaalisten viitteiden näyttäminen, luominen ja poistaminen komponenttien välillä. Komponentille annetaan asetusdatana CSS-valitsin, joka osoittaa kortissa elementtiin, josta viitteitä voidaan raahata.

Komponentin toiminta toteutetaan käytännössä komponentin komponenttimanagerissa, joka kuuntelee korteista tapahtumia. Manageri sisältää jsPlumb-kirjaston mukaisen olion, jonka avulla voidaan HTML-elementtien väliin piirtää SVG-pohjaisia viivoja ja nuolia [9]. Manageri myös alustaa sekä kortit että mahdolliset viitattavat elementit jsPlumb-kirjaston avulla joko lähteiksi tai kohteiksi. Näytettävistä viitteistä pidetään managerissa kirjaa, ja eri elementtien viitteitä voidaan vapaasti kytkeä näkyviin tai piilottaa.

6.2 Kanban-sovellus

Toteutetun arkkitehtuurin avulla kehitettiin Kanban-sovellus osaksi Realm-järjestelmää. Kanban-sovellus oli ensimmäinen arkkitehtuuria hyödyntävä sovellus, ja siten toimi tehokkaasti arkkitehtuurin konkreettisessa testauksessa.

Kanban-sovellus koostuu tilakorteista ja Kanban-korteista. Tilakortit koostuvat komponenteista

- Subview
- DragAndDrop

Tilakortit toteuttavat alinäkymän Subview-komponentilla, jossa näkyvät kyseisessä tilassa olevat Kanban-kortit. DragAndDrop-komponentista käytetään tiputuksen runkoa, jolla määritellään tilakorttiin alue Kanban-korttien tiputusta varten.

Kanban-kortit koostuvat seuraavista käytöskomponenteista:

- Subview
- DragAndDrop
- Minify
- Hierarchy
- Menu

Kanban-kortit toteuttavat alinäkymän Subview-komponentilla, jossa näytetään kortin lapsikortit. Yksittäisen kortin hierarkkiaa pidetään yllä Hierarchy-komponentissa. Minify-komponentin avulla kortin tarkemmat tiedot voidaan piilottaa tai näyttää käyttäjälle. Oletuksena kortti on latauksen yhteydessä kiinni. Menu-komponentti tarjoaa käyttäjälle oikotiet uusien alikorttien luomiseen. DragAndDrop-komponentista käytetään kumpaakin runkoa, jonka johdosta Kanban-kortteja voi sekä raahata että tiputtaa toisten Kanban-korttien sisään.

6.3 Katselmointisovellus

Katselmointisovelluksen totetuttamiseen voitiin käyttää hyvin paljon jo Kanban-sovelluksen kehityksessä luotuja komponentteja. Käytännössä uutta toiminnallisuutta tuli VisualReferences-komponentin muodossa. Sovellus koostuu kolmesta korttityypistä, jotka ovat kommentti-, vastaus- ja korjauskortti.

Kommenttikortti koostuu seuraavista käytöskomponenteista:

- Subview
- Minify
- VisualReferences
- Deletable

Subview-komponentilla luotiin korttiin alinäkymät vastaus- ja korjauskorteille. Vastauskortteja voi olla yhdessä kommenttikortissa useita, ja korjauskortteja vain yksi. Minify-komponentin avulla tarkemmat kortin tiedot voidaan piilottaa tai näyttää käyttäjälle. VisualReferences-komponentti vastasi visuaalisten viitteiden piirtämisestä ja niiden hallinnasta. Deletable-komponentin avulla kommenttikortteja voidaan poistaa järjestelmästä.

Vastauskortti ja korjauskortti koostuvat seuraavista käytöskomponenteista:

- Minify
- Deletable

Kumpikin komponentti toimi samalla tavalla kuin kommenttikorteissakin. Korteista voidaan siis piilottaa tarkempia tietoja tai esittää niitä, sekä kortteja voidaan poistaa käyttöliittymästä.

6.4 MyWork-sovellus

MyWork-sovelluksen toteuttaminen Kanban-sovelluksen jälkeen oli hyvin triviaalia. Sovellus voitiin koostaa jo Kanban-sovelluksen yhteydessä toteutettujen komponenttien avulla erittäin nopeasti. Komponenteilla voitiinkin toteuttaa kaikki sovellukselle vaadittu toiminnallisuun kirjoittamatta arkkitehtuuriin tai komponentteihin uutta ohjelmakoodia. Sovellus koostui tilakorteista sekä työkorteista.

Tilakortit koostuvat seuraavista komponenteista:

- Subview
- DragAndDrop

Toiminnallisuudet komponenteissa ovat samat kuin aiemmin esitellyissä sovelluksissa. Subview-komponentilla voitiin sijoittaa työkortteja tilakortin sisään. Raahattavia työkortteja kyettiin tiputtamaan tilakortin sisään käyttämällä DragAndDrop-komponentista tiputettavan kortin runkoa.

Työkortit koostuvat seuraavista komponenteista:

- Minify
- DragAndDrop

Minify-komponentilla saatiin työkortin sisältö piilotettua tai näytettyä käyttäjälle. DragAndDrop-komponentista käytettiin raahattavan kortin runkoa, jonka avulla kortteja pystyi raahaamaan.

7. TULOSTEN ARVIONTI

Tässä luvussa arvioidaan konkreettisen työn tuloksia sekä ei-toiminnallisten vaatimusten että yleisen tehokkuuden kannalta.

7.1 Toteutuksen vaihe

Arkkitehtuuri toteutettiin iteratiivisesti rakentamalla ensin asiakaspään sovelluskehys tukemaan sovellusten rakentamista. Toistaiseksi mallipohjia ei rakenneta suunnitellun arkkitehtuurin mukaisesti, vaan mallielementtien rakenteet luodaan Realm-järejstelmän valmiskomponenteilla. Osia vaatimuksista ei täten toteutetuissa iteraatioissa olla vielä täytetty, eikä niiden onnistumista siten voida arvioida. Suurin osa ei-toiminnallisista vaatimuksista on kuitenkin saatu täytettyä.

7.2 Vaatimusten täyttymisen arviointi

Vaatimuksen toteutumista arvioidaan vaatimuskattavuusmatriisilla. Arviointi on jaettu neliportaiseen asteikkoon. Asteikko on kuvattu taulukossa 7.1.

Arvo	**Kuvaus**
+	Vaatimuksen täyttämisessä onnistuttiin hyvin
0	Vaatimuksen täyttämisessä onnistuttiin kohtuullisesti
-	Vaatimuksen täyttämisessä onnistuttiin heikosti
E	Vaatimusta ei täytetty

Taulukko 7.1 *Vaatimusten täyttymisen arvioinnissa käytetty asteikko*

Taulukossa 7.2 on kuvattu vaatimukset ja niiden täyttymisen arvio. Vaatimukset on ryhmitelty kategorioittain.

Yleiset vaatimukset saatiin täytettyä suhteellisen hyvin. Arkkitehtuurista saatiin yleinen tehokkaalla arkkitehtuuri- ja suunnittelumallien käytöllä. Toteutettu sovelluskehys ei ole vahvasti sidoksissa Realm-järjestelmään, joten sitä voidaan tarvittaessa mukauttaa toimimaan muillakin järjestelmillä korttipohjaisten käyttöliittymien toteutukseen. Luonnollisesti yhteistyö Cometa Solutions Oy:n järjestelmien

Tunniste ja nimi	Arvio
Yleiset vaatimukset	
A01, Yleinen arkkitehtuuri	+
A02, Liitos olemassaoleviin sovelluksiin	+
A03, Ajoympäristö	+
A04, Kehitysympäristö	+
A05, Mobiilikehitys	**0**
A06, Testattavuus	+
Suorituskyvyn vaatimukset	
S01, Käyttäjän havaitsema tehokkuus	+
S02, Toiminta huonoilla yhteyksillä	**0**
Erikoistamisvaatimukset	
E01, Normaalikäyttäjän muokkaukset	**E**
E02, Kehittyneen käyttäjän muokkaukset	**0**
E03, Uusi käyttökohde	+
E04, Uusi käyttötapaus	+
E05, Uuden käyttötapauksen yleistäminen	**0**
E06, Uudet sovellukset	+
E07, Uusien sovellusten käyttöliittymät	**E**
Käytettävyysvaatimukset	
K01, Työvaiheiden määrä	**0**
K02, Intuitiivisuus	**0**
K03, Vikasietoisuus	+
K04, Avustava käyttöliittymä	**-**
K05, Käyttöliittymän yleisyys ja ymmärrettävyys	**0**

Taulukko 7.2 *Vaatimusten täyttyminen*

kanssa on myös hyvällä mallilla. Ajoympäristönä toimivat nykyaikaiset selaimet, joissa onnistuttiin hyvin käyttämällä mahdollisimman paljon standardoituja tekniikoita tai yhteensopivia ohjelmistokirjastoja. Mobiilikehitykseen ei nykyisessä iteraatiossa asetettu suurempaa painoarvoa, ja oletettavasti järkevä tuki mobiilikehitystä varten vaatii pidempää suunnittelua erillisen mobiilikäyttöliittymän tai responsiivisen käyttöliittymän [13] toteuttamiseksi. Sovelluskehyksen ohjelmakoodi on luotu testattavaksi, ja sille työn aikana toteutettiin yksikkötason testit.

Suorituskyvyn vaatimukset saatiin myös täytettyä suhteellisen hyvin. Käyttäjälle näkyvä korttipohjaisen sovelluksen tehokkuus on nykyisessä iteraatiossa hyvä. Tähän päädyttiin minimoimalla ohjelmakoodin DOM-kutsut siten, että kaikki eivälttämätön toteutetaan puhtaasti ohjelman sisäisillä tietorakenteilla. Heikkojen yhteyksien toimintaa ei päästy toistaiseksi testaamaan tarpeeksi perusteellisesti, mutta voidaan olettaa latenssien vaikutusten hieman nousevan suuremman tietomäärän johdosta. Tätä voidaan tulevissa iteraatioissa optimoida välimuistin tehokkaalla

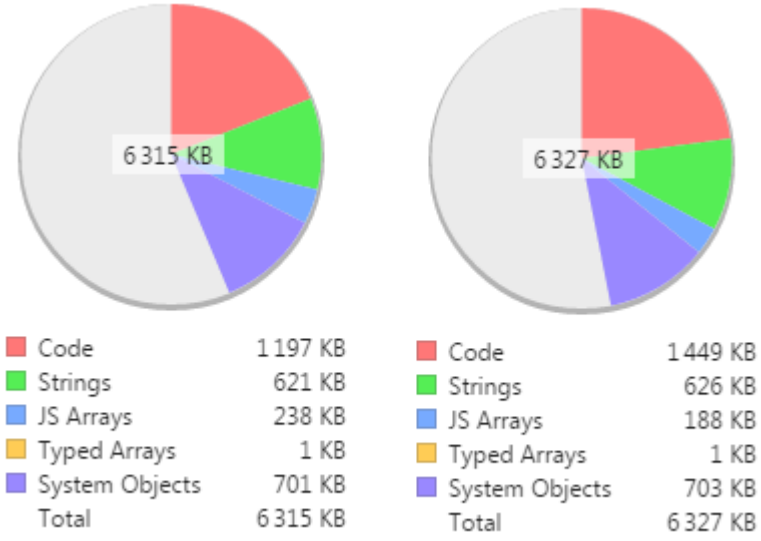
käytöllä.

Erikoistamisvaatimukset saatiin täytettyä kohtuullisesti suurimpana ongelmana on puutteellinen mallipohjien rakentaminen. Tästä syystä osaa erikoistamisista ei vielä saada toteutettua. Erikoistamisvaatimusten täyttäminen onkin seuraavan toteutusiteraation keskeisin tavoite.

Käytettävyysvaatimukset saatiin myös täytettyä kohtuullisesti. Työn puitteissa ei järjestetty suurempaa käytettävyystestausta, joten arviot perustuvat pitkälti kollegoiden anekdootteihin sovellusten käytettävyydestä.

7.3 Tehokkus

Tehokkuuden mittaukset toteutettiin Google Chromen kehittäjätyökaluilla [7]. Tehokkuutta voidaan vertailla Kanban-sovelluksen avulla, josta on toteutettu aikaisempi uutta sovellusta vastaava versio. Muistinkäytön osalta oletettu tulos oli lievä muistinkäytön lisääntyminen yhteisen ohjelmistokehyksen ohjelmakoodin lisääntyessä. Mittaustulokset olivatkin yllättäviä, sillä muistinkäyttö pysyi käytännössä samana sovellusten välillä.



Kuva 7.1 *Kanban-sovellusten muistinkäyttö*

Kuvassa 7.1 on esitetty kahden eri Kanban-sovelluksen version muistinkäytöt sovelluksen käynnistyksen jälkeen. Vasemmalla kuvassa on uusi suunnitellulla arkkitehtuurilla toteutettu versio ja oikealla on vanha versio. Kuvista näkee hyvin selkeästi, ettei muistinkäytössä ole tapahtunut suuria muutoksia.

Suoritinkäytössä muutokset latausaikaan jäivät hyvin vähäisiksi, eikä tarkempaa latenssin vaikutusta voida nykyisessä ympäristössä vielä tehokkaasti testata. Suurimmat muutokset tapahtuivat sovelluksen eri osien suorituksen painoissa.

Kuva 7.2 *Uuden Kanban-sovellusten suoritinkäyttö*

Kuvassa 7.2 on esitelty uuden Kanban-sovelluksen suoritinkäyttö sovelluksen käynnistyksen aikana. Kuvasta on eritelty kolme osaa, joista 1. osa on sovelluksen käynnistäminen. 2. osa on sovelluksen alustaminen ja 3. osa on korttien piirtäminen.

Kuva 7.3 *Vanhan Kanban-sovellusten suoritinkäyttö*

Kuvassa 7.3 on esiteltynä vanhan Kanban-sovelluksen suoritinkäyttö sovelluksen käynnistyksen aikana. Kuva on jaoteltu samalla tavalla kuin kuva 7.2.

Mittauksista voidaan todeta sovelluksen käynnistämisen vaatiman ajan pysyneen kutakuinkin samana sillä erotuksella, että vanha on noin 50 millisekuntia uutta sovellusta nopeampi. Voidaan kuitenkin päätellä, että korttien määrän lisääntyessä uudella arkkitetuurilla toteutettu sovellus toimii huomattavasti ripeämmin. Tämä johtuu sovelluksen piirron huomattavasta nopeutumisesta. Piirto vanhassa Kanban-sovelluksen versiossa kesti noin 100 millisekuntia, ja uudessa versiossa sama määrä saadaan piirrettyä noin 50 millisekuntiin.

Mittausten pohjalta voidaan väittää, etteivät resurssivaatimukset asiakaspään tietokoneille näytä kasvavan. Suuremmat tehonlisäykset tulevat oletettavasti suuremmalla korttimäärällä ja optimoimalla nykyisiä ratkaisuja.

7.4 Tavoitteiden saavutus

Työssä toteutettu arkkitehtuuri on vielä kehitystyön alla. Kuitenkin työssä asetettu tavoite uudelleenkäytettävistä ohjelmakoodista ja vahvemmasta selainpään sovellusten integraatiosta on saatu jo saavutettua.

Toteutettu mekanismi ladata ja hyödyntää käytöskomponenteilla toteutettua jaettua toiminnallisuutta korttien välillä on todettu kehittäjien puolesta hyväksi. Tämä on nopeuttanut selainmeen kehitettävien sovellusten toteutusta huomattavasti.

Toistaiseksi ei nykyisessä sovellusarkkitehtuurissa olla toteutettu työkalutukea kaikille mallipohjaisen ympäristön työkaluille. Tämä tullaan toteuttamaan seuraavassa iteraatiossa, mutta toistaiseksi tavoitte ei ole onnistunut.

8. YHTEENVETO

Työssä suunniteltiin ja toteutettiin korttipohjainen web-näkymäarkkitehtuuri, joka täyttää suurimman osan sille asetetuista vaatimuksista riittävästi. Arkkitehtuurille ominaista on entiteetti-komponentti-järjestelmä -arkkitehuurimallin käyttö, joka sopikin tarkoitukseen yllättävän hyvin.

Omat haasteensa arkkitehtuurin toteuttamiseen aiheutti kohdejärjestelmän mallipohjaisuus ja siitä aiheutuvat kohdealuekohtaiset ongelmat. Järjestelmän aiheuttamat haasteet kuitenkin saatiin ratkaistua ottamalla niiden asettamat rajoitukset huomioon suunnittelun aikaisessa vaiheessa.

Sovelluksissa siirretään suuri määrä laskutehosta ja muistivaatimuksista käyttäjän selaimelle. Verkon yli siirrettävän datan määrä laskee huomattavasti, sillä arkkitehtuurin mukaisissa sovelluksissa siirretään suuri osa korttien vaatimasta HTML-merkkauksesta elementtipohjana, josta HTML-merkkaus rakennetaan. Näin jokaiselle elementille ei tarvita omaa HTML-merkkausta sivunlatauksen yhteydessä. Arkkitehtuuri vaikuttaisi myös pienentävän sovelluksien muistinkäyttöä, joka johtunee oletettavasti vähennettyjen DOM-kutsujen määrästä, mitä ei tosin voida varmentaa tutkimatta syytä tarkemmin.

Arkkitehtuurissa käytetyt ratkaisut ovat käyttötarkoitukseen sopivia ja sovellusten luonnista ja kehityksestä tuli huomattavasti aikaisempaa helpompaa. Luonnollisesti arkkitehtuurin yhteyteen luoduissa työkaluissa on puutteita, joihin pyritään vastaamaan jatkokehityksen yhteydessä. Kokonaisuutena työssä onnistuttiin vastaamaan arkkitehtuurille määriteltyihin vaatimuksiin hyvin.

LÄHTEET

[1] *Backbone.js.* URL: http://backbonejs.org/ (viitattu 13.04.2015).

[2] S. Bilas. *A Data-Driven Game Object System.* URL: http://gamedevs.org/uploads/data-driven-game-object-system.pdf (viitattu 18.01.2015).

[3] M. Calore. *How Do Native Apps and Web Apps Compare?* 2010. URL: http://www.webmonkey.com/2010/08/how-do-native-apps-and-web-apps-compare/ (viitattu 30.01.2015).

[4] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1.* 1999. URL: http://tools.ietf.org/html/rfc2616 (viitattu 26.12.2014).

[5] P. Fischer. *Building Cross-Platform Apps with HTML5.* 2013. URL: https://software.intel.com/en-us/html5/articles/building-cross-platform-apps-with-html5 (viitattu 19.05.2015).

[6] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Allison-Wesley Professional, 1995.

[7] Google. *Web Developer Tools.* URL: https://developer.chrome.com/devtools (viitattu 16.05.2015).

[8] I. Haikala ja J. Märijärvi. *Ohjelmistotuotanto.* Helsinki: Talentum Media Oy, 2006, s. 268–269.

[9] *jsPlumb.* URL: http://www.jsplumb.org/doc/home.html (viitattu 01.04.2015).

[10] J. Koivula. *Microsoft Wordin laajentaminen ohjelmistojen mallinnustyökaluksi.* Diplomityö. Tampere: Tampereen teknillinen yliopisto, 2012.

[11] K. Koskimies ja T. Mikkonen. *Ohjelmistoarkkitehtuurit.* Helsinki: Talentum Media Oy, 2005.

[12] K. Koskimies et al. *UML työvälineenä ja tutkimuskohteena.* URL: http://www.cs.tut.fi/~ohar/kirjallisuutta/UML%20tyovalineena%20ja%20tutkimuskohteena.pdf (viitattu 19.05.2015).

[13] E. Marcotte. *Responsive Web Design.* 15. toukokuuta 2010. URL: http://alistapart.com/article/responsive-web-design (viitattu 13.04.2015).

[14] A. Mesbah ja A. van Deursen. "Migrating multi-page web applications to single-page Ajax interfaces". Teoksessa: *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on.* IEEE. 2007, s. 181–190.

[15] Mozilla Developer Network. *Drag and drop.* URL: https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Drag_and_drop (viitattu 19.05.2015).

[16] Object Management Group. *OMG Unified Modeling Language (OMG UML), Infrastructure*. 2011. URL: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/.

[17] C. Schulte zu Berge et al. *CAMPVis – A Game Engine-inspired Research Framework for Medical Imaging and Visualization*. Tekninen raportti. Technische Universität München, 2014. URL: http://campar.in.tum.de/Main/CAMPVis (viitattu 18.01.2015).

[18] W3C. *Document Object Model (DOM)*. 2005. URL: http://www.w3.org/DOM/ (viitattu 12.01.2015).

[19] W3C. *HTML & CSS*. URL: http://www.w3.org/standards/webdesign/htmlcss (viitattu 01.01.2015).

[20] W3C. *HTML5 Recommendation*. 2014. URL: http://www.w3.org/TR/2014/REC-html5-20141028/ (viitattu 26.12.2014).

[21] W3C. *HTML5 Working Draft*. 2014. URL: http://www.w3.org/TR/html51/ (viitattu 13.01.2015).

[22] W3C. *Offline Web Applications*. 2008. URL: http://www.w3.org/TR/offline-webapps/ (viitattu 19.05.2015).

LIITE A. TOTEUTETTAVIEN SOVELLUSTEN TOIMINNALLISET VAATIMUKSET

Tunniste	Nimi	Kuvaus
T01	Näkymien kooste	Näkymät koostuvat korteista.
T02	Korttien tyypit ja tiedot	Kortit voivat olla eri tyyppisiä ja sisältää eri tietoja.
T03	Uusien korttien luonti	Näkymään on kyettävä luomaan uusia kortteja.
T04	Alikorttien luonti	Kortteihin on kyettävä luomaan alikortteja.
T05	Korttien järjestely ja ryhmittely	Kortteja on kyettävä järjestelemään ja ryhmittelemään.
T06	Korttien sisällön muokkaus	Korttien sisältöä on kyettävä muokkaamaan.
T07	Kortin poistaminen	Kortteja on kyettävä poistamaan.
T08	Kortin kollapsointi	Korttien varsinainen sisältö on kyettävä piilottamaan. Kortti ei katoa ruudulta kollapsoitaessa.
T09	hierarkkisuus	Kortit voivat koostua korteista. Korttien alikortit on kyettävä näyttämään kortissa sekä kokonaan että viitteinä.
T10	Drag&Drop	Kortteja tulee pystyä siirtämään määriteltyjen säiliöiden välillä kokonaan tai viittauksina.
T11	Graafiset viittaukset	Korttien välillä on kyettävä näyttämään viittauksia graafisesti.
T12	Graafisten viittausten luonti	Graafiset viitteet on kyettävä luomaan vetämällä viittaavasta kortista viitattavaan elementtiin.
T13	Graafisten viittausten poisto	Graafisia viittauksia on kyettävä poistamaan kortista itsestään.

Toiminnalliset vaatimukset