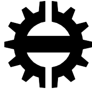


TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ROBERT NURMINEN

WEB-SOVELLUKSEN ARKKITEHTUURIN KEHITYS JA YLLÄPITO

Diplomityö

Tarkastaja: professori Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa 4. marraskuuta 2015

TIIVISTELMÄ

ROBERT NURMINEN: Web-sovelluksen arkkitehtuurin kehitys ja ylläpito
Tampereen teknillinen yliopisto
Diplomityö, 56 sivua
Toukokuu 2016
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Tommi Mikkonen

Avainsanat: web, ohjelmistoarkkitehtuuri, refaktorointi, suunnittelumallit, koodin hajut

Kun ohjelmisto on saatu ylläpitovaiheeseen, vaatii ohjelma edelleen työpanosta. Ohjelman konteksti muuttuu ja sen kautta ohjelmaan tulee uusia toiminnallisia vaatimuksia. Ohjelmasta löytyy myös virheitä, jotka tulee korjata. Sekä uusien toiminnallisuuksien toteuttaminen, että bugien korjaaminen on välttämätöntä, jotta ohjelma saadaan pidettyä jatkuvasti tuotantokäytössä. Muutoksien tuloksena ohjelman koodin laatu tippuu, ellei laadun ylläpitämiseksi tehdä erikseen toimenpiteitä.

Tässä diplomityössä on tutkittu erään ylläpitovaiheessa olevan web-sovelluksen arkkitehtuuria ja sen laatua. Tarkasteltavassa sovelluksessa on runsaasti ajan mittaan kerääntynyttä teknistä velkaa, jota on kertynyt siksi, että järjestelmää tehdessä on panostettu huomattavasti enemmän uusien toiminnallisuuksien tekoon, kuin olemassa olevan koodin refaktorointiin. Teknisen velan vähentämiseksi haettiin lähdekirjallisuudesta tyypillisiä ratkaisuja suunnitteluongelmiin sekä suunnittelumallien että koodin hajujen näkökulmista. Lisäksi työssä pyrittiin tutkimaan arkkitehtuuriratkaisuja, jotka tukevat järjestelmän tulevaisuudensuunnitelmia. Kirjallisuudesta opittua teoriaa kokeiltiin käytännössä kahdella eri tehtävällä, jolla saatiin tuettua työssä tehtyä oppimista. Ensimmäisessä käytännön osassa refaktoroitiin yksi osa sovelluksen koodista ja toisessa osassa luotiin REST-rajapinta.

Työssä havaittiin, että tarkasteltavassa järjestelmässä on paljon ongelmia, jotka johtuvat erityisesti liian suurista luokista ja pitkistä metodeista. Liian suuret luokat aiheuttavat tarkasteltavassa järjestelmässä myös sen, että monilla luokilla on useita vastuualueita. Liian pitkät metodit taas vaikeuttavat huomattavasti koodin luettavuutta erityisesti sisäkkäisten ehtolausekkeiden vuoksi. Pitkissä metodeissa joutuu myös ymmärtämään paljon koodia kerralla, mikä hidastaa koodin ymmärtämistä. Ongelmat ovat pohjimmiltaan teknistä velkaa ja johtuvat osittain siitä, että koodia tehdessä ei ole täysin ymmärretty tarvetta tehdä yksinkertaisempia luokkia ja metodeja. Suunnittelumallien ja arkkitehtuuriratkaisujen osalta työssä saavutettiin uutta osaamista, jonka avulla voidaan tulevaisuudessa kehittää järjestelmään yhä parempia ratkaisuja.

ABSTRACT

ROBERT NURMINEN: Development and maintenance of web application's architecture
Tampere University of Technology
Master of Science Thesis, 56 pages
May 2016
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Tommi Mikkonen

Keywords: web, software architecture, refactoring, design patterns, code smells

When an application is in maintenance phase, it still requires work. New functional requirements appears because of change in the application's context. Also new bugs are found when application is being used. Creating new functionalities and fixing bugs are needed in order to keep the application operational. When new code is added to the application, the quality level decreases if some additional work is not done to keep the quality high.

In this thesis the architecture of a web application was investigated so that its quality can be maintained. In the application, some technical debt has built up and some work is needed to decrease it. Technical debt has been building up because too much work has been invested in creating new functionality rather than refactoring the existing code base. To decrease the technical debt, some common solutions for common design problems useful in the context were seeked from literature. The solutions were seeked especially from design pattern and code smell point of view. Additionally, some architectural solutions were investigated so that they can possibly be used when implementing new features in the system. The theory investigated from the literature was experimented in practice with two separate tasks. In the first one a specific area of the code was refactored and in the second one a REST API was created.

It was noticed in the thesis that the investigated application had multiple problems which were caused by too complex classes and too long methods. Too complex classes introduce the situation where some classes have multiple responsibilities. On the other hand too long methods cause readability problems because of too deep indentations. Too long methods are also too complicated to understand, since there simply is too many things to understand at once. Problems with complex classes and long methods are partly caused by technical debt and partly because the programmers have not sufficiently understood the problems that complex and long methods introduce. During the thesis the team also learned design patterns and architectural solutions. These patterns and solutions will help to create better solutions in the system in the future.

ALKUSANAT

Diplomityö projektina lähti liikkeelle kun yritys, jossa olin ennen diplomityön aloitusta työskennellyt puolitoista vuotta, lupautui toimimaan yrityksenä, johon saisin toteuttaa diplomityöni. Pohdimme esimieheni kanssa yhdessä mahdollista työn aihetta, joksi lopulta valikoitui liittymään järjestelmän laadun kehittämiseen ja uusien arkkitehtuuriratkaisujen etsimiseen.

Diplomityön teko oli iso projekti. Haasteita diplomityössä tuotti itse työn lisäksi työn kirjoittamisen itsenäinen aikatauluttaminen täysipäiväsen töidenteon rinnalla iltaisin ja viikonloppuisin. Työaikaa käytin yrityksessä vain diplomityön käytännönosuuksien toteuttamiseen. Aikataulu pääsikin venymään alkuperäisestä suunnitelmasta useilla kuukausilla. Loppukirin tuloksena työ valmistui viimeinkin huhtikuussa 2016.

Kiitokseni diplomityön suorituksen mahdollistajina tahdon osoittaa yritykselle, johon sain toteuttaa diplomityöni ja diplomityön tarkastajalleni Tommi Mikkoselle, joka antoi paljon rakentavaa palautetta työn eri vaiheissa.

Tampereella, 18.4.2016

Robert Nurminen

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. TAUSTAA 2
2.1 Web-sovellusten konteksti 2
2.2 Web-sovellusten arkkitehtuuri 3
2.2.1 Dynaaminen web 4
2.2.2 Palvelinpuolen ohjelmointi 5
2.2.3 Pysyvä tieto 6
2.2.4 Arkkitehtuurin dokumentointi ja laadun mittaaminen 6
2.3 Suunnitteluperiaatteet 8
2.3.1 Suunnittelumallit 8
2.3.2 MVC 10
2.3.3 REST ja SOAP 10
2.3.4 SOLID 11
2.4 Web-sovellusten evoluutio ja ylläpito 11
2.4.1 Ohjelmistojen evoluution kahdeksan lakia 12
2.4.2 Vaikutusanalyysi 13
2.4.3 Refaktorointi 13
3. TARKASTELTAVA JÄRJESTELMÄ 15
3.1 Järjestelmän arkkitehtuuri 15
3.2 Arkkitehtuurin laatu 16
3.2.1 Ylläpidettävyys 17
3.2.2 Suorituskyky 18
3.2.3 Helppokäyttöisyys 18
3.3 Mittarit 18
3.3.1 Mittareita arkkitehtuurin nykytilasta 19
3.3.2 Vertailu avoimen lähdekoodin projekteihin 21
4. POTENTIAALISESTI HYÖDYNNETTÄVÄT SUUNNITTELUMALLIT JA REFAKTOROINNIT 24
4.1 Suunnittelumallit 24
4.2 Ohjelman hajujen refaktorointi 25
4.3 Web-sovellusten arkkitehtuurisuunnittelun trendit 26
4.3.1 Mikroservicet 27
4.3.2 Docker 27
4.3.3 DevOps 28
4.3.4 Big data 29
5. KÄYTÄNNÖN KOKEILUT 30
5.1 FormModel.php:n refaktorointi 30
5.1.1 Ensimmäinen refaktorointi-iteraatio 30
5.1.2 Toinen refaktorointi-iteraatio 32
5.1.3 Refaktoroinnin lopputulos 33

5.2 REST-rajapinnan toteuttaminen ulkoisten palveluiden käytettäväksi 34
5.2.1 Arkkitehtuuripäätöksiä 35
5.2.2 Syntynyt komponentti: rajapinta 36
5.2.3 REST-rajapinnan jatkohyödyntäminen 37
6. JÄRJESTELMÄN ARKKITEHTUURIN TULEVAISUUS 39
6.1 Arkkitehtuurin laadun jatkokehittäminen 39
6.1.1 Ylläpidettävyys 39
6.1.2 Suorituskyky 40
6.1.3 Helppokäyttöisyys 41
6.1.4 Hajujen refaktorointi 41
6.1.5 Refaktorointitaktiikka 43
6.2 Suunnittelumallien hyödyntäminen 44
6.3 Tulevaisuudessa hyödynnettävät teknologiat ja toimintatavat 45
7. YHTEENVETO JA JOHTOPÄÄTÖKSET 47
LÄHTEET 48

.

1. JOHDANTO

Ohjelmistoa ylläpidettäessä ohjelmiston konteksti muuttuu, joka johtaa tarpeeseen muokata ohjelmistoa. Kun ohjelmistoon tehdään muutoksia, sen laatu hiljalleen heikkenee, ellei sen laadun ylläpitämiseen ja edistämiseen erityisesti panosteta.

Tämän diplomityön tarkoituksena on kartoittaa ja suunnitella miten tarkasteltavan ohjelman rakennetta tulisi kehittää koodin ja arkkitehtuurin laadun parantamiseksi. Diplomityössä on tutkittu erään yritystoiminnassa käytetyn web-sovelluksen arkkitehtuuria, johon on ajan mittaan ehtinyt kerääntyä runsaasti teknistä velkaa. Haasteita tarkasteltavassa järjestelmässä tuottaa järjestelmän tiettyjen osien heikko ymmärrettävyys ja suunnitteluratkaisut, jotka hankaloittavat muutosten tekoa.

Suunnitteluongelma ohjelmistokehityksessä koostuu kolmesta osasta: toiminnallisesta vaatimuksesta, ratkaisusuunnitelmasta ja lopullisesta toteutuneesta ratkaisusta [1]. Tässä diplomityössä keskitytään kehittämään ratkaisusuunnitelmia ja perustellaan niitä toiminnallisilla vaatimuksilla, hyvillä ohjelmointitavoilla ja yleisillä suunnitteluratkaisuilla.

Tässä työssä otetaan huomioon järjestelmän tulevaisuudensuunnitelmat, kuten uudet toiminnalliset vaatimukset. Pääasiassa kyse on siis refaktoroinnista sekä koodi-, että arkkitehtuuritasolla. Työhön liittyy myös kevyitä käytönnönosuuksia, joissa joitakin osia suunnitelmasta toteutetaan ja yritetään arvioida niiden hyötyjä. Työn aluksi, luvussa 2, esitellään työhön liittyvää taustateoriaa. Luvussa 3 esitellään tarkasteltavan järjestelmän arkkitehtuuria ja tulkitaan sen laatua. Järjestelmän laatua tulkitaan sekä mittaamalla koodin laatua, että pohtimalla mitä ongelmia järjestelmän laadussa on. Luvussa 4 pyritään etsimään kirjallisuudesta tapoja järjestelmän laadun kehittämiseksi. Luku 5 on tämän diplomityön käytännönosuus, jossa refaktoroidaan valittu osa järjestelmästä ja luodaan kokonaan uusi toiminnallisuus järjestelmään. Luvussa 6 pohditaan, kuinka työtä tehdessä havaittuja ongelmia tulisi ratkaista. Ratkaisuja haetaan työssä tutkittujen suunnittelumallien, hajujen ja arkkitehtuuriratkaisujen avulla. Lopuksi luvussa 7 on yhteenveto tästä diplomityöstä ja sen onnistumisesta.

2. TAUSTAA

Tässä luvussa esitellään diplomityön ymmärtämiseen tarvittavia käsitteitä ja konsepteja. Vaikka työ keskittyy pääasiassa palvelinpuoleen, esittely lähtee liikkeelle kohdasta 2.1 web-sovelluksille keskeisestä asiakaspään selaimessa olevasta käyttöliittymäpuolesta. Selaimessa oleva käyttöliittymäpuoli vaikuttaa kontekstina myös palvelinpuolen ohjelmointiin. Kohdassa 2.2 esitellään web-sovellusten arkkitehtuuria lähtien liikkeelle dynaamisesta webistä ja palvelinpuolen ohjelmoinnista. Näiden jälkeen esitellään pysyvän tiedon käsittelyä, päätyen lopulta käsittelemään arkkitehtuurin dokumentointia. Web-sovellusten arkkitehtuurin jälkeen esitellään web-sovellusten arkkitehtuuriin liittyviä suunnitteluperiaatteita kohdassa 2.3. Kohdassa 2.4 tuodaan vielä esille ohjelmien evoluutioon liittyvää teoriaa. Näin saadaan käsitys siitä, miten ohjelmat tyypillisesti muokkautuvat uusien vaatimusten noustessa esiin ja miten muutosta käsitellään.

2.1 Web-sovellusten konteksti

Web-sovellukset ovat verkkoselaimissa toimivia sovelluksia [2], jotka ovat tavallisia staattisia verkkosivuja monimutkaisempia kokonaisuuksia. Web-sovelluksia on olemassa valtavasti erilaisiin käyttötarkoituksiin erilaisilla, ja erilaisia resursseja vaativilla, ominaisuuksilla. Web-sovellukset voivat olla esimerkiksi blogeja, ammattikäyttöön tarkoitettuja yritysjärjestelmiä, lehtien verkkoversioita, foorumeita, valtavalla käyttäjäkapasiteetilla olevia sosiaalisen median sovelluksia tai vaikkapa hakukoneita.

Web-sovellusteknologiat ovat kehittyneet voimakkaasti koko olemassaolonsa ajan. Aluksi verkkosivut olivat pelkkää tekstiä sisältäviä dokumentteja, josta ne kehittyivät rakennetta ja sivulta toiselle vieviä linkkejä sisältäviksi HTML-pohjaisiksi dokumenteiksi.

Nykyään web-sovellukset ovat parhaimmillaan visuaalisesti kehittyneitä dynaamisia kokonaisuuksia. Kehitys on edelleen kovaa ja web-sovellukset kykenevät jatkuvasti uudenkaltaisiin toimintoihin eri teknologioiden kehittyessä.

Web-sovellusten voimakas kehitys selittyy sen mahdollistamilla ominaisuuksilla, johon tavalliset tietokoneille asennettavat sovellukset eivät kykene. Yksi web-sovellusten merkittävimmistä vahvuuksista löytyy siitä, kuinka hyvin ne pystyvät saavuttamaan ihmisiä. Jos web-sovellus on julkisessa verkossa, pystyy sitä käyttämään kuka tahansa mistä tahansa päin maailmaa. [2]

Web-sovelluksia voi kehittää, päivittää ja ylläpitää kaikille järjestelmän käyttäjille ilman tarvetta asentaa järjestelmää erikseen jokaiselle loppukäyttäjälle. Tämä on yksi pääsyistä web-sovellusten suosiolle sekä yksityisillä käyttäjillä, että yrityksissä. [2]

Web-sovellukset eivät myöskään ole riippuvaisia käyttöjärjestelmistä, sama koodi toimii kaikissa käyttöjärjestelmissä, joissa vain on nykyaikainen verkkoselain. Yleisesti käytössä olevia verkkoselaimia ovat Internet Explorer, Google Chrome, Safari ja Mozilla Firefox [3].

Vaikka suurin osa toiminnallisuuksista toimii suoraan eri selaimilla, joutuu web-sovelluksiin usein kuitenkin tekemään selaimesta riippuvia ehdollisuuksia. Ehdollisuuksia tarvitaan, jotta kaikki yksityiskohdatkin toimivat täysin samalla tavalla kaikilla selaimilla.

Nykyään eri selainten uusimpien versioiden toteuttamat perusominaisuudet noudattavat HTML5-standardia kohtuullisesti. Monimutkaisemmissa ja uusimmissa ominaisuuksissa selainten kehittäjät ovat ottaneet joitakin vapauksia selaimen toiminnassa, jotka aiheuttavat web-kehittäjille ajoittain ongelmia.

Eri selainten välisiä eroavaisuuksia enemmän ongelmia verkkosivuilla aiheuttaa vanhojen selainversioiden käyttö. Erityisen ongelmallisia ovat Internet Explorerin vanhempien versioiden käyttö, joka on kuitenkin edelleenkin suhteellisen yleistä. Vanhempiin Windows-käyttöjärjestelmiin ei voi asentaa uusimpia versioita Internet Explorer -selaimesta. Esimerkiksi Windows XP:llä pystyy käyttämään parhaimmillaan Internet Explorerin versiota 8, kun taas tämän tekstin kirjoitushetkellä uusin olemassa oleva versio on Internet Explorer 11. Microsoftin Windows–käyttöjärjestelmän version 10 julkaisun yhteydessä siirryttiin Windows-käyttöjärjestelmissä käyttämään Internet Explorer –selainten sijaan vielä tuoreempaa Microsoft Edge [4] –selainta. Usein syystä tai toisesta käyttäjän järjestelmään on myös jäänyt asentamatta uusin järjestelmän tukema versio vaikka järjestelmä tukisikin uudempaa.

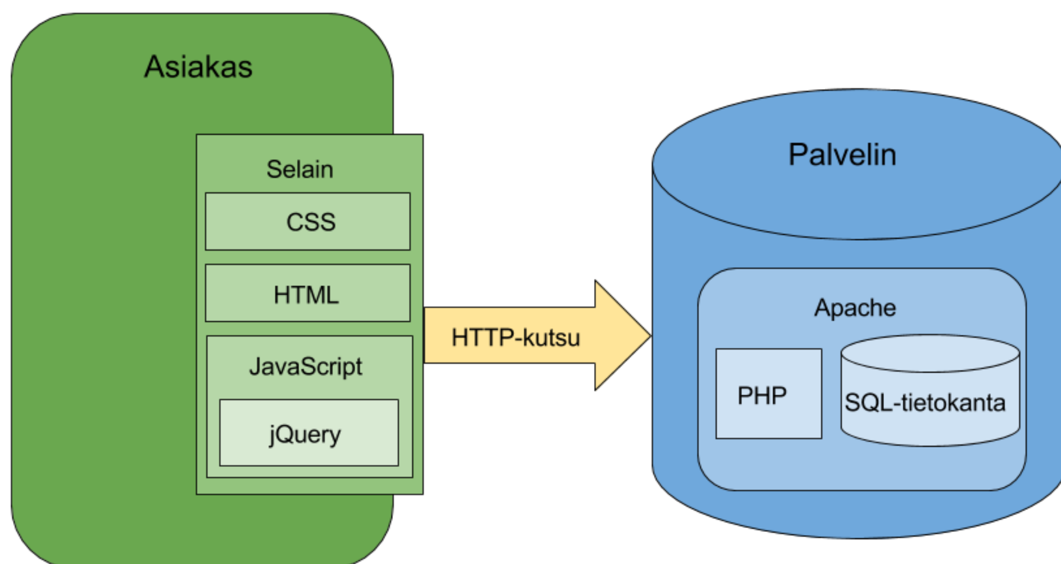
Koska laaja joukko web-sovellusten käyttäjistä käyttää vanhempia selainversioita, täytyy web-sovelluksen kehittäjän tehdä kompromissi tuettujen selaimien ja järjestelmän ominaisuuksien välillä. Vanhojen selainversioiden tukeminen vaatii lisäksi suurempaa määrää testailua ja ohjelmointia, jotta saadaan järjestelmä toimimaan samalla tavalla vanhoissa ja uusissa selainversioissa.

2.2 Web-sovellusten arkkitehtuuri

Myös web-sovelluksilla on tarve selkeälle ja hyvin rakennetulle arkkitehtuurille, vastaavasti kuin muuallakin ohjelmistoalalla kehitettävillä ohjelmistoilla. Web-sovelluksiin pätee melko hyvin muun ohjelmistoalan arkkitehtuuriteoriat. Web-sovelluksilla on kuitenkin lisäksi omia yksityiskohtiaan, joita arkkitehtuuria suunnitellessa tulee ottaa huomioon.

Web-sovellusten logiikka on jakaantunut kahteen eri kohteeseen: asiakkaalle ja palvelimelle. Asiakas on jokin taho, joka käyttää palvelimen palveluja, esimerkiksi verkkosivun vierailija. Kun käyttäjä pyrkii siirtymään verkkosivulle, selain lähettää osoitekenttään asetettua osoitetta vastaavan kutsun palvelimelle. Kutsuun saattaa sisältyä parametreja, jotka palvelin prosessoi palvelimelle määritellyn koodin ja mahdollisen tietokannan tilan perusteella. Lopulta palvelin palauttaa tilanteenmukaiset datat, tässä tapauksessa verkkosivun asiakkaalle. [5]

Kuvassa 1 kuvataan sitä, kuinka asiakas ja palvelin liittyvät toisiinsa, sekä mitä teknologioita asiakkaaseen ja palvelimeen liittyy. Asiakas-palvelinmallissa asiakas, joka on käytännössä web-sovelluksissa käyttäjän käyttämä selain, pyytää palvelimelta tietoa. Palvelin käsittelee asiakkaan pyynnön ja palauttaa sen perusteella siihen liittyvän tiedon.



Kuva 1. *Asiakas-palvelinmalli ja tässä työssä käsiteltävien perusteknologioiden sijoittuminen niihin*

Käyttäjälle näkyvää tietoa selaimessa käsitellään siis aluksi palvelimella, jonka jälkeen näytetyn verkkosivun tila saattaa muuttua vielä asiakkaan päähän määriteltyjen HTML:n, CSS:n ja JavaScriptin perusteella. Prosessointi on nykyaikaisilla laitteilla sekä asiakkaalla että palvelimella sen verran tehokasta, että loppukäyttäjälle verkkosivulla tapahtuvat muutokset vaikuttavat tapahtuvan välittömästi 5.

2.2.1 Dynaaminen web

Viime vuosikymmenen lopulla alettiin puhua yhä enemmän ja enemmän dynaamisista web-sivuista ja Web 2.0:sta. Osia web-sivun sisällöstä saatiin ladattua milloin tahansa ilman sivun päivittämistä, mikä paransi verkkosivujen käyttökokemusta.

Yksi keskeisistä avainsanoista Web 2.0:ssa on AJAX, joka on lyhenne sanoista “Asynchronous JavaScript and XML”. AJAX tasapainottaa perinteistä Web-palveluiden asiakas-

palvelin -suhdetta sallimalla kommunikaation asiakkaan ja palvelimen välillä asynkronisesti taustalla ilman sivulatausta, samalla kun käyttäjä käyttää Web-palvelua [6, s. 14].

Perinteisesti AJAX toimii niin, että selaimella tehdään taustalla JavaScriptillä XMLHttpRequest-pyyntö palvelimelle, joka käsittelee pyynnön ja palauttaa sitä vastaavan tiedon XML-muodossa. Tämän jälkeen selain käsittelee palvelimen palauttaman tiedon JavaScriptillä määritetyllä tavalla ja muuttaa tarvittaessa selaimessa ajettavan sovelluksen tilaa. Nykyään AJAX-viesteissä kulkee XML:n sijaan usein viestejä JSON-muodossa. [6, s. 107]

Käytännössä XMLHttpRequest-pyynnöissä käytetään palvelimen kanssa kommunikointiin RPC:tä. RPC on mekanismi, jolla lähetetään kutsuja toiselle tietokoneelle tai ohjelmalle, kuten palvelimelle [7]. Sen käyttäminen vaatii tyypillisesti vastauksen odottamista palvelimelta ennen kuin suoritusta jatketaan RPC-kutsun tehneen sovelluksen päässä [7].

Nykyään Web 2.0 on harvoin käytetty termi, sillä sen sisältämät ominaisuudet ovat sulautuneet muuhun verkkosivuun yleisesti käytetyiksi teknologioiksi, eikä näiden teknologioiden käyttö ole mitenkään poikkeuksellista.

JavaScriptin käyttö dynaamisen verkkosivun luonnissa on työlästä, JavaScriptiä ei alunperin edes tarkoitettu laajojen verkkosivujen luontiin [6, s. 36]. Tätä varten on luotu JavaScriptiä abstrahoivia kirjastoja, joilla tietyt usein tarvitut toiminnot saa tehtyä yksinkertaisesti. Näistä merkittäviä ovat esimerkiksi jQuery [8] ja AngularJS [9].

jQuery on hyvin yleisesti käytössä oleva minimalistinen (varmista lähteestä) JavaScript-kirjasto DOM:in (Document Object Model) käsittelyn yksinkertaistamiseen [6, s. 224]. Se sisältää myös loistavat puitteet pelkkää JavaScriptiä yksinkertaisemman AJAX:in hyödyntämiseen erityisesti JSON-pohjaisilla viesteillä.

Tällä hetkellä trendikkäintä JavaScript-kirjastoa, AngularJS:ää voi kutsua jo ohjelmistokehykseksi, sillä se määrittelee koko selainpään arkkitehtuurin. AngularJS tähtää sivulatausten minimalisointiin lataamalla kaiken tarvittavan sivun näyttämiseen etukäteen ja varsinaiset yksittäiset sivujen sisällöt myöhemmin AJAX-pohjaisesti.

2.2.2 Palvelinpuolen ohjelmointi

Web-sovelluksen palvelinpuolen ohjelmointiin on olemassa sekä erityisesti siihen luotuja kieliä, että web-ympäristöön esimerkiksi ohjelmistokehyksellä mukautettuja kieliä. Palvelinpuolta ohjelmoidaan paljon sekä käännettävillä, että skriptikielillä. Yleisiä ohjelmointikieliä web-sovellusten palvelinpuolen ohjelmointiin on muun muassa Java, PHP, Python, ja C# [10]. Tässä diplomityössä käsitellään ohjelmistoa, jonka palvelinpuoli on ohjelmoitu PHP:lla.

PHP on helposti opittava vakaa ohjelmointikieli [1]. PHP on ylivoimaisesti suosituin palvelinpään ohjelmointikieli, jonka pohjalla web-palvelimia ajetaan. Suuri syy PHP:n käyttömäärien ylivoimaisuudelle on CMS:illä (Content Management System) luodut sivustot, eli esimerkiksi PHP-pohjaisilla WordPressillä, Joomla!:lla tai Drupalilla luodut sivustot. Muitakin PHP:lla luotuja sivustoja on valtavasti, yhtenä merkittävimmistä Wikipedia. Myös Facebook on alunperin luotu PHP:lla. [5]

PHP ei ohjaa ohjelmoijaa erityisen voimakkaasti käyttämään hyvää ohjelmointityyliä tai arkkitehtuuria. Monimutkaisia PHP-sovelluksia luodessa ohjelmoijan tuntemus kielestä korostuu, sillä ohjelmoijan tulisi tunnistaa tilanteita joissa PHP saattaisi käyttäytyä yllättävästi.

2.2.3 Pysyvä tieto

Kuten muissakin sovelluksissa, joissa on tarve tallettaa suuria määriä tietoja pitkäaikaisesti, myös web-sovelluksissa käytetään yleisesti SQL-tietokantoja. Samoin web-sovelluksissa hyödynnetään usein tietokantoja abstrahoivia ORM:eja (Object Relational Mapping). ORM mahdollistaa SQL-kyselyiden muuttamisen ohjelmoijan silmissä olioiden käsittelyksi tai yksinkertaisemmaksi ORM:in omaksi versioksi SQL:stä. [11]

Sen lisäksi, että ORM helpottaa tietokannan tietojen käsittelyä, helpottaa ORM:in käyttäminen myös tietokannan vaihtoa. Vaikka ORM:in käyttö helpottaa tietokannan vaihtamista, on tietokannan vaihtaminen kuitenkin käytännössä hyvin haastavaa varsinkin jos vaihdettavassa tietokannassa on jo paljon tietoa. PHP:ssa yleisimmin käytetty ORM on Doctrine. Doctrine toimii sekä itsenäisenä kokonaisuutena PHP:ssa, että yleisimmin käytettyjen PHP:n ohjelmistokehysten kanssa. [11]

2.2.4 Arkkitehtuurin dokumentointi ja laadun mittaaminen

Ohjelmistojen laadun kehittäminen ja ylläpitäminen on tärkeää. Keskeisiä apuvälineitä tähän on arkkitehtuurin dokumentointi ja erilaisten mittareiden käyttö. Dokumentoinnilla jaetaan tietoa ja mittareilla selvitetään järjestelmän laadun nykytilannetta.

Ohjelmiston arkkitehtuurin dokumentointi on kannattavaa, jotta pystytään ylläpitämään tietoa siitä, mitä arkkitehtuuripäätöksiä on tehty ja miksi ne on tehty. Lisäksi dokumentoinnin avulla pystytään jakamaan tietoa ihmisten välillä ja perehdyttämään uusia ohjelman parissa töitä tekeviä henkilöitä.

Arkkitehtuuria voidaan dokumentoida esimerkiksi tekstimuotoisena tai UML-kaavioilla. Tekstimuotoinen dokumentaatio on hyvä arkkitehtuuriratkaisujen perusteluun ja UML-kaaviot taas järjestelmän nykytilan dokumentointiin.

UML toimii yhteisenä kielenä arkkitehtuurin esittämiseen ohjelmistokehittäjien ja muiden ohjelman teknisestä näkökulmasta kiinnostuneiden sidosryhmien kesken. UML:llä pystyy dokumentoimaan ohjelman arkkitehtuuria monesta eri näkökulmasta. Sillä voi havainnollistaa arkkitehtuuria komponenttitasolta yksityiskohtaisempaan olion tai luokan dokumentaatioon tai vaikkapa ohjelman käyttötapauksia. UML:llä voi myös havainnollistaa kuinka tieto liikkuu eri komponenttien tai luokkien välillä. [12]

Järjestelmän laatua voidaan mitata monesta eri näkökulmasta. Voidaan mitata esimerkiksi koodin laatua, kehitysprosessin laatua tai vaikkapa dokumentaation laatua. Tässä diplomityössä keskitytään mittaamaan koodin laatua.

Koodin laadun mittaamisesta voi oikein käytettynä olla runsaasti hyötyä. Koodin laadun mittaaminen on kuitenkin haastavaa ja koodin laadun tason määritteleminen vertailtaviksi numeroiksi vielä haastavampaa. Laatu koodissa on monimutkainen käsite ja sitä mitatessa täytyykin ottaa paljon eri näkökulmia huomioon. Jokin mittari saattaa toimia hyvin jossakin kontekstissa ja toinen toisessa, mutta yleisesti hyväksyttävää joka kontekstiin sopivaa mittaria on vaikeaa löytää [13].

Koodin laadun mittaamisen hyödyistä on kiistelty paljon. Mittareita täytyy käyttää oikein ja oikeassa mielentilassa, jotta niistä pystytään hyötymään. Keskittymällä pelkästään mittareiden parantamiseen laiminlyödään helposti sellaisia tärkeitä asioita, joita mittarit eivät mittaa. Mittareita voi myös huijata niin, että saavutetaan hyviä mittaustuloksia vaikka ohjelman laatu ei olekaan parantunut [14].

Koodin laadun mittaamiseen liittyy useita yleisiä virheitä. Mittauksia ei kannata tehdä yksilötasolla, jotta psykologiset vaikutukset tuo yksilöille tarvetta huijata mittareita. Samasta syystä huonoista mittaustuloksista ei kannata myöskään rangaista. [14]

Mittauksia tehdessä kannattaa käyttää useita eri mittareita, jotta saadaan laajempi käsitys monimutkaisen ohjelman laadusta, eivätkä kehittäjät keskity vain yhden mittarin tulosten kehittämiseen. Mittarit kannattaa valita tavoitteiden mukaan. [14]

Yksi vanhimmista ja yksinkertaisimmista koodin mittareista on koodirivien määrä [13]. Koodirivien määrällä saadaan helposti selville ohjelman laajuuden mittaluokka, mutta se ei välttämättä ole suoraan verrannollinen ohjelman monimutkaisuuteen, laajennettavuuteen, tai testattavuuteen.

Yksi yleisimmistä käytetyistä tavoista mitata ohjelman monimutkaisuutta on mitata ohjelman syklomaattista kompleksisuutta [13]. Syklomaattinen kompleksisuus tarkoittaa ohjelman vaihtoehtoisten ja itsenäisten suorituspolkujen määrää. Esimerkiksi yhden ehtolauseen sisältävän ohjelman syklomaattinen kompleksisuus on 2, koska ohjelman voi suorittaa kahdella eri tavalla.

Luokan vastuualueiden määrää voidaan arvioida LCOM-mittarilla (Lack of Cohension in Methods). LCOM-mittarin arvo saadaan laskettua luokan toisiinsa liittyvien metodien avulla. LCOM-mittarin avulla saadaan tunnistettua luokkia, joita todennäköisesti pitäisi jakaa useampiin osiin. [15]

Ylläpidettävyysindeksillä saa mitattua nimensä mukaisesti järjestelmän ylläpidettävyyttä suuntaa antavasti. Ylläpidettävyysindeksissä otetaan huomioon useita eri metriikkoja, kuten Halsteadin kompleksisuus, syklomaattinen kompleksisuus, rivien määrä ja joskus myös kommenttien lukumäärä. [16]

Ohjelmakoodia voidaan mitata ohjelmistollisesti ja esimerkiksi PHP-kielelle tarkoitettuja laadun mittausohjelmia löytyy useita. Laadunmittausohjelmista saatavilla luvuilla pystyy ainakin havainnollistamaan muutosta ja mittaustuloksia analysoimalla ehkä perustelemaan työn saavutuksiakin. Niitä käytetäänkin tässä diplomityössä työn tulosten esittelyn tukena kun pohditaan ohjelmaan tehtyjen muutosten vaikutuksia.

2.3 Suunnitteluperiaatteet

Ohjelmistoalalle on muodostunut käytännössä hyväksi havaittuja tapoja toteuttaa ratkaisuja tiettyihin ongelmiin. Tässä luvussa esitellään tällaisista hyvistä toteutustavoista suunnittelumalleja, MVC-arkkitehtuurisuunnittelumallia, rajapintojen toteutusta, sekä SOLID-periaatteita.

2.3.1 Suunnittelumallit

Suunnittelumallit ovat yleiskäyttöisiä, käytännön kokeilujen kautta hyväksi todettuja ratkaisuja yleisiin ongelmiin [17, s. 7]. Suunnittelumallit ovat tyypillisesti monen ammattilaisen hyväksymiä ja käytännössä hyväksi toteamia [18], joten suunnittelumalleja hyödyntämällä saa todennäköisesti aikaan toimivampia ratkaisuja, kunhan suunnittelumallia käyttäessä vaan sisäistää suunnittelumallin haitat ja hyödyt.

Suunnittelumallit jaetaan usein kolmeen eri pääkategoriaan: luomismalleihin, rakennemalleihin ja käyttäytymismalleihin. Luomismallit keskittyvät uusien olioiden luontiin, rakennemallit olioiden ja oliojoukkojen rakenteeseen, sekä käyttäytymismallit olioiden väliseen kommunikointiin. [19]

Suunnittelumallejakin voi käyttää väärin ja väärän asian ratkaisemiseen. Suunnittelumalleja käytetään tarpeettomasti myös vain siksi että on tyylikästä käyttää suunnittelumalleja oikeasti kuitenkaan hyötymättä käytetystä suunnittelumallista [1]. Suunnittelumalleja ei siis tarvitse viljellä kaikkialle.

Alkuperäiset suunnittelumallit olio-ohjelmointikontekstiin julkaistiin vuonna 1994 Erich Gamman, Richard Helmin, Ralph Johansonin ja John Vlissidesin toimesta kirjassa "Design Patterns: Elements of Reusable Object-Oriented Software". [2]

Pitkällä tähtäimellä ei ole kannattavaa tehdä vain nopeita ratkaisuja ongelmien ratkomiseksi [1]. Suunnittelumallien hyödyntäminen johtaa uudelleenkäytettävämpiin, laajennettavampiin ja ylläpidettävämpiin ratkaisuihin, vaikka suunnittelumalliin perustuvan ratkaisun toteuttaminen saattaakin tuottaa hieman lisätyötä jos tarkastellaan lyhyttä aikaväliä [18].

Toisaalta yleensä sopivan suunnittelumallin löytämiseen ja suunnittelumallin ymmärtämiseen käytettyä aikaa kompensoi se, ettei jo olemassa olevaa ratkaisua tarvitse keksiä uudestaan. Lisäksi kerran opittua suunnittelumallia voi usein hyödyntää muuallakin, eikä opetteluun tarvitse enää seuraavalla kerralla käyttää aikaa. Kun ohjelman eri osissa on ratkaistu samankaltaisia ongelmia, hyödyntämällä samoja suunnittelumalleja saadaan koodista ja rajapinnasta keskenään yhdenmukaisempia. [2]

Koska suunnittelumallit ovat enemmänkin perusrunkoja tietynkaltaisien ongelmien ratkaisulle, tarkat lopulliset ratkaisut poikkeavat toisistaan. Suunnittelumallien hyödyntämiseen on saatavilla runsaasti tietoa siitä miten ja missä tilanteessa suunnittelumallia pitäisi käyttää. [2]

Suunnittelumallien dokumentoinnissa pyritään ilmaisemaan, mikä suunnittelumalli on kyseessä, mitä se ratkaisee, miksi se toimii ja mitä sen hyödyt sekä haitat ovat [2]. Kun suunnittelumalleja oppii lukemaan, löytyy suunnittelumallin kuvauksesta nopeasti tarvittu tieto. Suunnittelumallien kuvauksia voi hyödyntää esimerkiksi selittäessään tai opettaessaan toiselle suunnittelumallien perusteoriat hallitsevalle henkilölle, kuinka ratkaisisi tietyn suunnitteluongelman [2].

Koska suunnittelumalleilla on nimi, yleisesti tunnetut suunnittelumallit myös auttavat luomaan arkkitehtuurin suunnittelijoille nopeasti omaksuttavaa yhteistä sanastoa. Ne auttavat siis kommunikoinnissa, kun pohditaan ratkaisuja ongelmiin. [2]

Kun suunnittelumalleilla pyritään kuvaamaan hyviä toimintatapoja, vastaavasti on olemassa myös malleja huonoista toimintatavoista, eli niin kutsuttuja antipatterneja. Antipatterneilla pyritään kuvaamaan yleisiä toimintatapoja ja ongelmanratkaisutapoja, jotka aiheuttavat negatiivisia sivuvaikutuksia. Antipatterneja tutkimalla pystytään oppimaan toisten virheistä ja ne auttavat löytämään yhteistä sanastoa vastaavasti kuin suunnittelumallitkin. [20]

2.3.2 MVC

MVC (Model-View-Controller) on yksi keskeisimmistä arkkitehtuurin suunnittelumalleista. MVC keskittyy ohjelman arkkitehtuurin jakamiseen kolmeen osaan: malleihin, näkymiin ja kontrollereihin. MVC auttaa ylläpitämään ohjelmassa selkeää rakennetta parantaen ohjelman muokattavuutta ja ymmärrettävyyttä. [21]

Arkkitehtuurin malli-osassa on ohjelman järjestelmän liiketoimintalogiikka. Mallit toteuttavat esimerkiksi tietojen käsittelyn tietokannassa ja kommunikoinnin kolmannen osapuolien rajapintojen kanssa. [21]

Näkymät ovat yksinkertaisesti käyttäjälle näkyvän osan kuvaava osa koodia. Käytännössä ne ovat siis esimerkiksi HTML-, CSS- ja JavaScript-tiedostoja, joihin on mahdollisesti generoitu näkymän tilanteen vaatimaa tietoa. Käyttäjä vuorovaikuttaa järjestelmän kanssa näkymän kautta. [21]

Web-sovellusten kontrollereissa käsitellään käyttäjän HTTP-kutsuja ja haetaan, muokataan, lisätään sekä poistetaan tietoa malleihin kutsumalla sopivia mallien metodeita. Kontrollerit siis yhdistävät näkymät ja mallit toisiinsa käyttäjän pyyntöjen perusteella. [21]

2.3.3 REST ja SOAP

Sen lisäksi, että selaimet voivat hakea verkkosivuja ja esimerkiksi kuvatiedostoja palvelimelta, ne voivat hakea myös rakenteellista dataa. Rakenteellista dataa voidaan käyttää sivulla näytettävien tietojen muuttamiseen ilman erillistä sivulatausta.

Web-sovelluksissa sellaisissa käyttötapauksissa, jossa vaihdetaan dataa selaimen ja palvelimen välillä, on nykyään tyypillisintä käyttää REST-periaatteita. Perinteisempien asiakas-palvelin -arkkitehtuuria hyödyntävissä sovelluksissa yleisesti käytetty SOAP ei saavuttanut vastaavaa suosiota web-sovelluksissa raskaiden ja monimutkaisten viestien rakenteen vuoksi. [22]

REST on arkkitehtuurisuunnittelutyyli, joka soveltuu erityisen hyvin websovelluksiin [22]. Stardardinmukaisen REST:in soveltaminen asettaa tiettyjä rajoitteita sovelluksen rajapinnalle, joiden tulee täyttyä jotta voi kutsua sovellusta RESTful:iksi [22]. Joskus REST-arkkitehtuurin periaatteita voidaankin soveltaa osittain, jolloin arkkitehtuuria kutsutaan RESTish:iksi.

REST-rajapinnan kanssa kommunikoidaan taulukon 1 mukaisten HTTP-metodeiden avulla. Asiakkaan ja palvelimen välinen viestintä REST:issä tapahtuu tilattomasti HTTP:n välityksellä. Palvelin tarjoaa resursseja, joita palvelin hakee niihin liittyvillä tunnisteilla. [22]

Taulukko 1. *REST-rajapintaan lähetettävien HTTP-metodien tyypit*

Kutsun tyyppi	Kutsun aiheuttama operaatio
GET	Hakee resurssin rajapinnasta
POST	Lisää resurssin rajapintaan
PUT	Muokkaa resurssia rajapinnassa
DELETE	Poistaa resurssin rajapinnasta

Palvelimen tarjoamat resurssit saattavat mukautua kontekstiin asiakkaan lähettämien asiakkaan tilaan liittyvien parametrien perusteella. REST-rajapintaan voidaan lisätä, hakea, muokata ja poistaa resursseja. Rajapinta tunnistaa operaation tyypin HTTP-kutsun tyypin perusteella. [22]

2.3.4 SOLID

SOLID on lyhenne viidestä yleisesti hyväksytystä olio-ohjelmoinnin periaatteista. Nämä periaatteet ovat yhden vastuualueen periaate (Single Responsibility Principle), avoin/suljettu-periaate (Open/Closed Principle), Liskovin korvaavuusperiaate (Liskov Substitution Principle), rajapintojen erotteluperiaate (Interface Segregation Principle) ja riippuvuuksien käännettävyysperiaate (Dependency Inversion Principle). [23]

Yhden vastuualueen periaate ilmaisee sitä, että luokalla tulisi olla vain yksi vastuualue. Avoin/suljettu–periaatteen mukaan luokan pitäisi lähtökohtaisesti olla avoin lisäyksille, mutta suljettu muutoksille. Liskovin korvaavuusperiaatteen mukaan funktion pitää toimia edelleen sen jälkeen, kun sille annetaan alkuperäisestä suunnitelmasta poiketen tietystä luokasta periytetty luokka, vaikka alun perin olisi ollut tarkoitus antaa sen kantaluokka. Rajapintojen erotteluperiaate taas tuo esille, että rajapintojen tulisi olla riittävän pieniä. SOLID-suunnitteluperiaatteiden viimeisen periaatteen, riippuvuuksien käännettävyysperiaatteen mukaan luokkien pitäisi olla riippuvaisia abstraktioista konkreettisten luokkien sijaan. [23]

2.4 Web-sovellusten evoluutio ja ylläpito

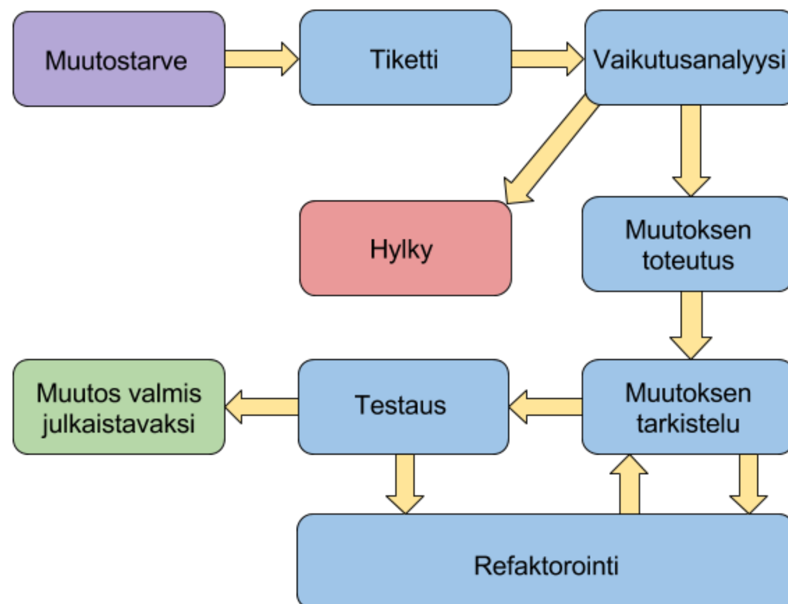
Sovellusten evoluutio tarkoittaa ylläpitotoimia ja -prosesseja, joilla saadaan aikaan uusi ohjelmistoversio, minkä vaikutuksen loppukäyttäjä huomaa uusina ominaisuuksina tai käyttökokemuksen muuttumisena [24].

Kuten kaikki muutkin ohjelmistot, web-sovelluksetkin kehittyvät laajassa käytössä. Kun ohjelmaa käytetään, muuttuu sen konteksti hiljalleen, jonka vuoksi ohjelmistoa täytyy muokata tai siihen täytyy lisätä uusia ominaisuuksia ohjelman tarpeellisuuden säilyttämiseksi tai kehittämiseksi [25].

Olemassa olevien ohjelmistojen ylläpitoon käytetään teollisuudessa enemmän aikaa kuin uusien ohjelmistojen kehittämiseen. Ylläpito onkin ohjelmistokehityksen vaiheista kaikkein eniten resursseja kuluttava. Tämän lisäksi ylläpitovaiheessa ohjelmoijat käyttävät yli puolet ajastaan olemassa olevan koodin ymmärtämiseen. [25]

Ohjelmistoa ylläpidettäessä ohjelmaa täytyy muokata ja ohjelmiston muokkaaminen on haastavaa, sekä riskialtista. Ohjelmistossa olevaa virhettä korjatessa muutokset saattavat vaikuttaa toiseen osaan ohjelmistoa aiheuttaen uusia ongelmia. [25]

On olemassa neljään eri kategoriaan kuuluvia ylläpitotehtäviä. Nämä alun perin E. Burton Swansonin määrittelemät tehtävät ovat korjaavat, mukauttavat, viimeistelevät ja ennaltaehkäisevät ylläpitotehtävät. [24]



Kuva 2. *Prosessi, jolla tässä diplomityössä tarkasteltavaan järjestelmään lisätään ominaisuuksia*

Kuvassa 2 on havainnollistettu prosessia, jonka yksittäinen järjestelmään tehtävä muutos käy läpi järjestelmässä, jota tässä diplomityössä tarkastellaan. Tässä diplomityössä prosessista tarkastellaan erityisesti vaikutusanalyysiä, muutoksen toteutusta, muutoksen tarkistelua ja refaktorointia.

2.4.1 Ohjelmistojen evoluution kahdeksan lakia

Meir Lehman kehitti 70-luvulla kolme teoreemaa ohjelmistoille, joiden tarpeet muuttuvat jatkuvasti ohjelmiston kontekstin muuttuessa. Nämä ohjelmistojen ylläpidon laiksi kutsutut teoreemat laajenivat 1990-luvun loppuun mennessä kahdeksaan. Osat varsinkin alkuperäiseen kolmeen teoreemaan kuulumattomista teoreemoista ovat kiisteltyjä, mutta kuitenkin tietyllä varauksella pääosin edelleenkin päteviä. Lait pätevät erityisen hyvin monoliittisiin, yksittäisen tiimin kehittämiin suljetun lähdekoodin ohjelmistoihin. [24]

Ylläpidon ensimmäinen laki on “jatkuva muutos”. Koska ohjelman ympäristö muuttuu, täytyy ohjelmankin mukautua, jottei sen merkitys pienene [24]. Tällöin ohjelma muuttuu kunnes se päätyy tarpeettomaksi ja se hylätään [13]. Toinen laki “jatkuva monimutkaistuminen” kuvaa jatkuvan muutoksen vaikutuksia. Ohjelman ominaisuuksien määrän kasvaessa ohjelma juuttuu jatkuvasti monimutkaisemmaksi, ellei monimutkaisuuden vähentämiseksi tehdä samanaikaisesti töitä [26].

Kolmas laki, “itseohjautuvuus” esittää, että ohjelman evoluution kehityssuunta on riippuvainen siitä, missä kontekstissa ohjelmaa käytetään. Neljäs laki kuvaa sitä, että ohjelma kehittyy ylläpitovaiheen ajan useimmiten vakionopeudella, koska sen kehittämiseen käytettäviä resursseja ei muuteta. [24]

Viidennessä laissa määritellään, että ohjelmoijan täytyy ylläpitää tiettyä tasoa siinä, kuinka hyvin hän tuntee ohjelman. Mitä nopeammin ohjelma kehittyy, sitä enemmän ohjelmoija joutuu panostamaan muutoksiin perehtymiseen, mikä vie aikaa kehitykseltä. [24]

Ylläpidon kuudes laki kertoo, että ohjelman koko jatkaa kasvua tarpeiden muuttuessa ja lisääntyessä. Seitsemännessä laissa ilmaistaan, että ohjelman laatu laskee jatkuvasti muutoksia tehdessä, ellei ohjelman laatuun tehdä jatkuvasti erikseen parannuksia. Viimeisessä, eli kahdeksannessa ylläpidon laissa, määritellään ylläpitämisen olevan monimutkaista ja vaativan onnistuakseen palautetta useista eri lähteistä. [24]

2.4.2 Vaikutusanalyysi

Kun järjestelmään tehdään ylläpitotoimena muutos, olisi hyvä tehdä siihen liittyen vaikutusanalyysi. Vaikutusanalyysissa tutkitaan, mihin osiin ohjelmaa muutos voisi mahdollisesti vaikuttaa. Vaikutusanalyysin avulla voidaan esimerkiksi tehdä riskianalyysia, kannattaako muutosta tehdä. [24]

Jos muutos toteutetaan, saadaan tietoa myös siitä, että mihin muutokseen liittyvää testausta kannattaisi kohdistaa. Vaikutusanalyysin yhteydessä saatetaan löytää myös muita korjattavia kohteita. [24]

2.4.3 Refaktorointi

Refaktorointi on sellaisten muutosten tekemistä ohjelmistoon, jotka vaikuttavat koodin ymmärrettävyyteen ja ylläpidettävyyteen, mutta joka ei muuta koodin toiminnallisuutta. Refaktorointi sisältää muun muassa koodin uudelleenjärjestelyä, duplikaattikoodin poistamista ja koodin yksinkertaistamista. Refaktorointi on tärkeää, koska ilman sitä ohjelman jatkuva muuttuminen johtaa koodin ymmärrettävyyden merkittävään laskuun hidastaen tulevien muutosten tekoa. [24]

Refaktorointi jää ohjelmistokehityksessä herkästi turhan vähäiseksi, koska ohjelman uudet ominaisuudet menevät refaktoroinnin edelle välittömien hyötyjen vuoksi. Koska refaktoroitavaa voi olla paljonkin ja aikaa vähän, refaktorointia voidaan tehdä heikkolaatuisesti, mikä aiheuttaa taas lisää refaktoroitavaa. [26]

Refaktorointia voidaan toteuttaa kahdella eri taktiikalla. Ensimmäinen tapa on, että sille varataan erikseen aikaa niin, että korjataan jotain selkeää ongelmakohtaa laadukkaammaksi. Toinen tapa tehdä refaktorointia on tehdä sitä jonkin kehitystehtävän yhteydessä parannellen muutokseen liittyvää olemassa olevaa koodia. [27]

Refaktorointi liittyy myös juuri luotuun koodiin. Kun jokin toiminnallisuus on saatu toteutettua, vaatii se usein koodin laadun puolesta vielä viimeistelyä vaikka toiminnallisuus onkin jo toteutettu. Kuvasta 2 voi havaita, miten refaktorointi liittyy juuri tehdyn koodin paranteluun.

Refaktorointiin liittyy käsite “koodin haju”. Koodissa on tiettyjä suunnitteluperiaatteita rikkovia ominaisuuksia, hajuja, joita korjaamalla koodista saadaan ylläpidettävämpää ja ymmärrettävämpää. Toimenpiteet, joita hajujen korjaamiseksi tehdään, ovat refaktorointeja. [26]

Yleisiä koodin hajun aiheuttajia on suunnitteluperiaatteiden rikkominen, suunnittelumallien väärinkäyttö, ohjelmointikielen rajoitteet, proseduraalinen ajattelu olio-ohjelmoinnissa, vaikeasti muokattava koodi, sekä yleisten hyvien toimintatapojen ja prosessien noudattamatta jättäminen [26]. Useimpiin koodin hajun aiheuttajiin pystytäänkin vaikuttamaan suoraan ohjelmoijien ja ohjelmistoarkkitehtien osaamisella.

Tekemätöntä koodin laadun parannusta kutsutaan tekniseksi velaksi. Kun ohjelmoija tekee nopean korjauksen hyvin suunnitellun, ylläpidettävyyden huomioon ottavan ratkaisun sijaan, syntyy teknistä velkaa. Tekninen velka aiheuttaa tulevaisuudessa ongelmia ja kertautuu ajan mittaan suuremmiksi ongelmiksi. Teknistä velkaa voi syntyä koodiin, ohjelman suunnitelmaan, ohjelmiston testitapauksiin ja dokumentaatioon. Tekninen velka voi olla esimerkiksi vaihtelevaa ohjelmointityyliä, kun taas suunnittelusta aiheutunut tekninen velka voi olla vaikkapa koodin hajua aiheuttavaa suunnittelua. Testitapauksista aiheutuu teknistä velkaa, jos testitapauksia tehdään riittämättömästi tai testausta on suunniteltu väärällä tavalla. Dokumentaatioon liittyvä tekninen velka on esimerkiksi tekemättä jätettyä, huonotasoista tai vanhentunutta dokumentaatiota. [26]

3. TARKASTELTAVA JÄRJESTELMÄ

Tässä luvussa käsitellään tutkittavan järjestelmän ominaisuuksia. Aluksi kohdassa 3.1 kuvataan sitä miten järjestelmän arkkitehtuuri on nyt rakentunut. Tämän jälkeen kohdassa 3.2 jatketaan siihen, millaisia ongelmakohtia järjestelmässä on ja lopuksi kohdassa 3.3 mitataan järjestelmän ominaisuuksia.

3.1 Järjestelmän arkkitehtuuri

Käsiteltävää järjestelmää tarjotaan loppukäyttäjille SaaS-palveluna tai vaihtoehtoisesti loppukäyttäjän sisäverkkoon omalle palvelimelle asennettuna. Järjestelmä jakautuu käyttöliittymän osalta neljään eri näkymään: peruskäyttäjien näkymään, raportointinäkymään, käyttäjienhallintaan ja järjestelmän ylläpitonäkymään.

Järjestelmä on toteutettu PHP:lla ja tietokantana käytetään PostgreSQL:ää. Muita järjestelmän keskeisiä teknologioita ovat Apache, Doctrine, Smarty, JavaScript, sekä JavaScriptiin liittyvä jQuery. Apache [28] toimii tarkasteltavan järjestelmän HTTP-palvelinohjelmistona. Apache-palvelin ajaa PHP-sovellusta, joka on tarkasteltavan järjestelmän palvelinosa. Apachen päälle on asennettu myös PostgreSQL. Tarkasteltavassa järjestelmässä on käytetty tietokantaa abstrahoivana ORM:ina PHP-kirjasto Doctrineä [29]. Doctrine toimii hyvin järjestelmässä käytettävän PostgreSQL:n kanssa. Tietokantana, jota Doctrine abstrahoi, voisi käyttää tarvittaessa myös esimerkiksi MySQL:ää tai SQLiteä. Smartya [30] käytetään työkaluna sopivien web-näkymien generointiin. Smartyllä saadaan aikaan selkeämpiä HTML-tiedostoja generoivia templateja, joiden ylläpitäminen on helpompaa kuin se olisi pelkkää PHP-koodia käyttämällä. Smarty-templateista generoituu PHP-tiedostoja automaattisesti, jonka ansiosta ohjelmoijan ei tarvitse itse käsitellä dynaamisia näkymiä suoraan PHP-koodilla.

Järjestelmässä käytetään MVC-arkkitehtuuria. Ohjelmakoodi onkin jakaantunut MCV-arkkitehtuurin mukaisesti kolmeen keskeiseen osaan: malleihin, kontrollereihin ja näkymään. Malleissa on pyritty hallitsemaan tietokantaa Doctrine-objekteilla ja tukemaan Doctrine-luokkien toiminnallisuutta muilla malliluokilla. Kontrollereissa on keskitytty erityisesti käsittelemään käyttäjien syötteitä. Näkymäosiossa on tiedostot, joita kontrollereista generoidaan järjestelmän käyttäjien pyyntöjen perusteella käyttäjien selaimelle palautettavaksi.

Järjestelmässä on myös erillinen Utils-osio, joka sisältää toiminnallisuudet, joita ajetaan eräajona tietyin väliajoin riippumatta käyttäjien toiminnasta järjestelmässä. Utils-osioon kuuluu esimerkiksi sähköpostimuistutusten lähettäminen kerran yössä.

3.2 Arkkitehtuurin laatu

Tarkasteltavaa ohjelmaa on kehitetty jo noin kymmenen vuoden verran, minkä vuoksi arkkitehtuuriin on jäänyt vanhentuneiden ominaisuuksien ja ideoiden jäänteitä. Refaktorointia on tehty usein ajoissa ja viimeistään kun se on ollut välttämätöntä, mutta teknistä velkaa on kuitenkin ehtinyt kerääntyä runsaasti. Teknisen velan määrä ei ole vielä mitenkään ylivoimaista, mutta sen määrän kasvun hidastamiseen täytyy panostaa enenevissä määrissä, ettei se tulevaisuudessa kasva ylivoimaiseksi.

Ohjelmistokehitysresurssit tarkasteltavalle järjestelmälle ovat suhteellisen pieniä: järjestelmää kehittää kolme ohjelmistokehittäjää, joiden aika kuluu myös esimerkiksi testaukseen, asiakastukeen ja asiakkaiden ympäristöjen ylläpitoon. Aika on rajallista ja uudet ominaisuudet tuovat välitöntä lisäarvoa, toisin kuin refaktorointi. Refaktoroinnin hyödyt näkyvät pidemmällä aikavälillä, ja refaktoroinnin huomioitta jättämällä syntyy helposti lisää teknistä velkaa.

Käytössä olevia teknologioita ja kirjastoja päivitetään säännöllisesti tai epäsäännöllisesti niiden merkittävyydestä riippuen silloin, kun niissä tulee ongelmia vastaan tai niissä on uusia tarpeellisia toiminnallisuuksia. Huolellisimmin kiinnitetään huomiota käytettävään versioon käytetyissä ydinteknologioissa PHP:ssa, Apache:ssa ja PostgreSQL:ssä.

Tarkasteltavan ohjelman arkkitehtuuria suunnitellaan melko epäformaalisti. Ideoita arkkitehtuurin kehittämiseen säilytetään pääasiassa tehtävienhallinnassa tikettien kuvauksissa ja kommenteissa, sekä kehittäjien päässä. Kyseessä olevan kokoisessa yrityksessä arkkitehtuurisuunnittelun ei välttämättä tarvitsekaan olla paljoa tämän formaalimpaa. Kokonaisuus pysyy nykyisellään vielä hallinnassa, mikäli arkkitehtuurin hallitseva henkilöstö ei vaihdu.

Dokumentaation vähäisyys johtaa siihen, että järjestelmään liittyen on olemassa paljon hiljaista tietoa, joka taas korostaa henkilöstön säilyvyyden merkittävyyttä. Haasteita syntyy myös uuden teknisen henkilöstön perehdytyksessä. Mikäli yritykseen tulee uusi ohjelmointia tekevä työntekijä, täytyy hänelle selittää ohjelman ratkaisuista paljon kasvotusten. Uuden työntekijän täytyy osata kysellä korostuneen aktiivisesti vanhemmilta työntekijöiltä ohjelman toiminnasta.

Järjestelmän arkkitehtuurisuunnittelua vaikeuttaa pitkäjänteisen suunnitelmallisuuden vähäisyys ohjelman tulevien ominaisuuksien päättämisessä. Usein seuraavaa versiota aletaan miettimään tarkemmin vasta kun juuri kehityksessä oleva versio on loppusuoralla.

Merkittävimmät seuraavan vuoden aikana tehtävät ominaisuudet ovat lähtökohtaisesti jo kohtuullisen hyvin suunniteltuja. Se, että missä järjestyksessä ja missä versiossa suurimmat uudistukset tehdään, päätetään kuitenkin melko myöhään.

Sen lisäksi että version uusia ominaisuuksia päätetään myöhään, tulee järjestelmään lisäksi version aikana asiakkaiden nopealla aikatauluilla tarvitsemia ominaisuuksia. Nämä lisäominaisuudet ovat tyypillisesti liiketoiminnallisesti järkeviä ratkaisuja ja usein niiden toteuttamisesta saadaankin laskutettua työtä asiakkaalta.

Version uusiin ominaisuuksiin kesken version kehittämisen lisättävien ominaisuuksien määrä vaihtelee versiosta toiseen asiakkaiden tarpeiden mukaisesti ja ovat siis ennalta vaikeasti ennustettavissa. Asiakkaiden vaatimien ominaisuuksien määrän ennustettavuutta pitäisi siis saada parannettua ja niihin pitäisi pystyä varautumaan tehokkaammin.

3.2.1 Ylläpidettävyys

Keskeisin ongelma tarkasteltavan järjestelmän koodin ylläpidettävyydessä on niinkin yksinkertainen kuin luokkien ja metodien pituus. Ongelman taustalla on ohjelman ikä ja riittämätön olio-ohjelmoinnin periaatteiden huomioiminen, josta johtuen koodin laatu on hiljalleen päässyt heikkenemään. Samoin uusien toiminnallisuuksien liiallinen priorisointi koodin refaktorointiin verrattuna on aiheuttanut luokkien ja funktioiden paisumista.

Luokkien ja metodien liiallinen pituus on aiheuttanut useita muitakin ongelmia. Eri luokissa on osittain tai täysin toisiaan vastaavia toiminnallisuuksia. Toiminnallisuuden toteuttaminen vain yhteen paikkaan vähentäisi tarvetta tehdä samaa muutosta useaan eri paikkaan. Duplikaattikoodin määrä on onneksi järjestelmässä vielä suhteellisen vähäistä, mutta todennäköisyys duplikaattikoodin syntymiselle kasvaa ohjelman monimutkaistuessa.

Ehkä merkittävin koodin pituuteen liittyvä ongelma järjestelmässä on se, että yksittäiset luokat keräävät itseensä useita vastuualueita. Koska osaan luokista liittyy paljon ominaisuuksia, sattuu niitä muokatessa korostuneen herkästi virheitä.

Metodeista löytyy muitakin ongelmia, jotka ovat osa pitkien metodien ongelmaa. Järjestelmän koodissa esiintyvät usean tason sisäkkäiset ehtolausekkeet ja silmukat vaikeuttavat ymmärrettävyyttä ja koodin luettavuutta.

Koodin ymmärrettävyyttä hankaloittavat myös poikkeavuudet koodin tyylissä. Järjestelmässä käytetty tyyli on lähtökohtaisesti melko yhdenmukaista, sillä ohjelmoijat olivat osanneet itsenäisesti noudattaa vastaavaa ohjelmointityyliä kuin jo olemassa olevassa koodissa. Kaikilla ohjelmoijilla on kuitenkin pieniä poikkeavuuksia ohjelmointityylissä.

Koodin tyylin merkittävyys ymmärrettävyyteen ei korostu kolmen hengen tiimissä niin paljoa kuin se korostuisi esimerkiksi kymmenen hengen tiimissä. Kolmen hengen tiimissä jokainen ohjelmoija joutuu tottumaan vain kahden muun ohjelmoijan tyyliin.

Jos ohjelmoijia olisi nykyistä enemmän, vaatisi eri ohjelmoijien koodin lukeminen enemmän aikaa. Ohjelmoija joutuisi aina koodinpätkän lukemista aloittaessaan totuttelemaan

koodin tyyliin. Jos yrityksen ohjelmistokehitystoiminnan halutaan kasvavan, nousee ohjelmointityyli jatkossa yhä merkittävämpään asemaan ja ohjelmointityylistandardien asettaminen ajoissa on senkin osalta järkevää.

3.2.2 Suorituskyky

Järjestelmän suurimmat ongelmakohdat suorituskyvyn osalta liittyvät pääasiassa tietokantakyselyihin. Tietokannasta täytyy joissakin kyselyissä joidenkin asiakkaiden ympäristöissä hakea ja käsitellä hyvinkin suuria määriä tietoa esimerkiksi raportointia varten.

Yrityksen osaaminen tehokkuusasioiden huomioon ottamisessa ohjelmakoodissa on hyvällä tasolla. Järjestelmän suorituskykyyn on panostettu sen vaatimalla tavalla. Kun jossain ominaisuudessa havaitaan erityisiä suorituskykyongelmia, korjataan se yleensä jo seuraavaan järjestelmäversioon.

3.2.3 Helppokäyttöisyys

Vaikka järjestelmän helppokäyttöisyyteen vaikuttaa erityisesti käyttöliittymäsuunnittelu, voidaan helppokäyttöisyyttä edistää myös arkkitehtuurivalinnoilla. Oikeat arkkitehtuurivalinnat mahdollistavat järjestelmän kehittämisen helppokäyttöisemmäksi.

Järjestelmän helppokäyttöisyydessä on heikkouksia erityisesti järjestelmän ylläpitonäkymässä, jossa pystytään hallitsemaan järjestelmän sisältöä, käyttäjiä ja muita ominaisuuksia. Toisaalta pienetkin ongelmat muissa, useamman käyttäjän käyttämissä näkymissä on kriittisempiä koska ongelmat vaikuttavat useampiin ihmisiin.

Ylläpitonäkymää käyttävät lähinnä järjestelmän kehittäjät ja asiakkaiden pääkäyttäjät. Koska asiakkaiden pääkäyttäjät ovat useimmiten keskeisimpiä henkilöitä siinä, kuinka järjestelmä koetaan asiakkaalla, tulisi myös ylläpitonäkymässä korjata isoimpia ongelmia.

Yksi keskeisimmistä ongelmista ylläpitonäkymässä on se, että sivu latautuu suuressa osassa toiminnallisuuksista toiminnallisuuden suorittamisen jälkeen täysin uudestaan, vaikka vain sivun osan päivittäminen riittäisi. Sivulataukset voivat olla pitkiä, koska sivulla voi näkyä esimerkiksi asiakkaan organisaatiorakenne tai suuria määriä järjestelmän käyttäjiä, mikä tekee sivunlatauksesta entistä ongelmallisempaa.

3.3 Mittarit

Järjestelmän arkkitehtuurin nykytilasta voidaan saada jonkinlaista kuvaa tulkitsemalla koodin laatuun liittyviä, laajalti käytössä olevia mittareita. Verkosta löytyy monia PHP-koodin laadun mittaukseen löytyviä sovelluksia. Tähän diplomityöhön valittiin yleisten koodin laadun mittaamiseksi työkalu PHPMetrics [31].

Koodin laadun mittaamiseen on olemassa paljon eri työkaluja. Tätä työtä varten pyrittiin etsimään työkalu, joka mittaisi PHP-koodia riittävällä ja sopivalla mittarivalikoimalla ja jota olisi mielekäs käyttää. PHPMetrics valikoitui koodia mittaavaksi työkaluksi tähän diplomityöhön erityisesti sen perusteella, että se generoi intuitiivisia kuvaajia koodin laadusta.

Tätä työtä varten olisi ollut mielenkiintoista mitata myös koodin hajujen määrää koodissa. Koska PHP-koodin hajuja mittaavien ohjelmien ominaisuudet tuntuivat rajoittuvan lähinnä luokkien ja metodien pituuksiin, sisennyksiin ja tyyliasioihin, päätettiin se jättää kuitenkin tästä työstä pois.

Luokkien ja metodien pituus, sekä sisennyksien määrä vaikuttavat PHPMetricsin tuottamiin mittareihin, eikä niiden erikseen mittaaminen olisi ollut niin mielenkiintoista lisätietoa. Tyyliin liittyvät mittarit taas olisivat kertoneet lähinnä, että missä kohtaa koodissa on ongelmia tyylistandardeissa, eivätkä ne olisi antaneet yleiskuvaa luokista tai järjestelmästä.

3.3.1 Mittareita arkkitehtuurin nykytilasta

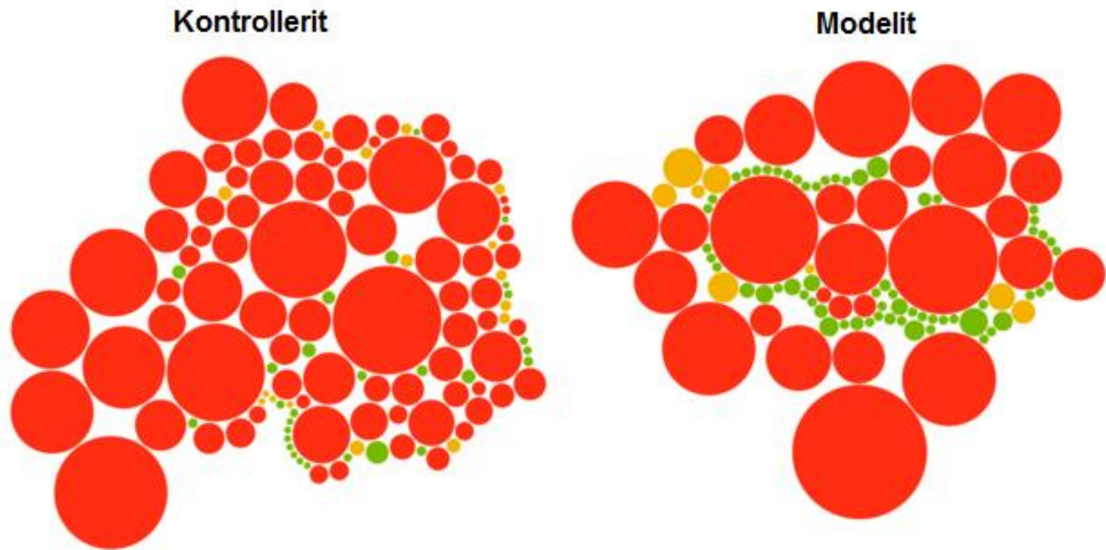
PHPMetrics on työkalu, joka mittaa yleisesti käytössä olevia koodin mittareita, kuten syklomaattista kompleksisuutta ja ylläpidettävyysindeksiä. Työkalu tulostaa mittauksen tulokset HTML-dokumenttiin, johon sisällytetään myös D3.js JavaScript-kirjastolla generoituja kaavioita.

Taulukossa 2 järjestelmän ylläpidettävyys saa kontrollereissa heikot ja malleissa välttävät pisteet. Mallit saavat myös uusien ohjelmoijien oppimiskynnyksessä välttävät pisteet. Muut kohdat koodin laadussa saavat alimmat mahdollisimmat pistemäärät, eli nolla pistettä. Järjestelmän rakenteessa on siis paljon kehitettävää ja hiottavaa.

Taulukko 2. *PHPMetrics-työkalun antamat pisteet järjestelmän laadusta pisteytettynä välille 0-100*

	Kontrollerit	Mallit
Ylläpidettävyys	10	47
Helppokäyttöisyys	0	35
Algoritmien yksinkertaisuus	0	0
Volyymi	0	0
Bugien todennäköisyyden vähäisyys	0	0

PHPMetricsin antamat tilastot järjestelmän tilasta eivät anna hyvää kuvaa järjestelmästä. Lähtökohtaisesti heikolta näyttävien tilastojen takana on aivan liian suuriksi paisuneet luokat ja funktiot.



Kuva 3. *PHPMetrics-työkalun generoimat kaaviot järjestelmän kontrollerien ja mallien laadusta*

Kuvassa 3 on PHPMetricsin generoima kaavio kontrollerien ja mallien luokkien syklomaattisesta kompleksisuudesta ja ylläpidettävyysindeksistä. Jokainen pallo kuvassa on oma luokkansa.

Pallon koko indikoi luokan syklomaattista kompleksisuutta: mitä suurempi pallo sen korkeampi syklomaattinen kompleksisuus. Pallon väri taas indikoi luokan ylläpidettävyysindeksiä. Vihreän pallon ylläpidettävyysindeksi on hyvällä tasolla, keltaisen kohtalaisella ja punaisen heikolla.

PHPMetricsin asettamat rajat sille, että koska luokan ylläpidettävyys on kohtalaisella ja koska punaisella tasolla, ovat yritystoiminnassa luotuihin sovelluksiin ehkäpä jopa liian tiukkoja. Jotta PHPMetrics-työkalun generoimien kuvaajaa 3 vastaavien kuvaajien ympyrät olisivat vihreitä, täytyisi tiedostot jakaa niin moneen osaan, että tiedostojen määrän hallintaan kuluisi jo tarpeettoman paljon aikaa.

PHPMetricsin asetettamat rajat hyvälle ylläpidettävyysindeksille tuntuisikin olevan suunnattu enemmänkin avoimen lähdekoodin projekteihin, joissa tiedostojen ymmärrettävyys on merkittävämpää kuin tiedostojen ylläpitoon kulutettu aika. Avoimen lähdekoodin projekteissa projektia työstävillä henkilöillä ei ole välttämättä niin paljoa aikaa sisäistää koko projektia kuin liiketoimintaa varten kehitettävien ohjelmistojen kehittäjillä. Avoimen lähdekoodin projekteja myös usein kehittää useampi henkilö, jolloin ymmärrettävyysongelmat koodissa kertautuvat kun jokainen ohjelmoija joutuu käyttää aikaa saman koodin ymmärtämiseen.

Tässä diplomityössä tarkasteltavan ohjelmiston ei siis ole välttämättä tarkoituksenmukaista tuottaa PHPMetricsin tuottamissa mittauksissa oletusraja-arvoilla pelkästään vihreitä ympyröitä. Jos koodin laatua halutaan jatkossakin mitata ylläpidettävyysindeksillä,

voisi tarkasteltavalle projektille olla tarkoituksenmukaisempaa asettaa matalammat tavoitteet.

Taulukko 3. *Ylläpidettävyysindeksiä kuvaavien värien kalibrointi yhdenmukaisemmaksi tavoitteiden kanssa*

Ylläpidettävyysindeksi	Oletusraja	Uusi raja
Vihreä ympyrä	> 85	> 70
Keltainen ympyrä	65-84	55-69
Punainen ympyrä	< 65	< 55

Taulukossa 3 on määritelty rajat, joilla PHPMetrics tuottaa nyt kuvaajia ja millaiseksi rajat olisi tarkoituksenmukaisempaa kalibroida projektin ongelmakohtien tunnistamiseksi. Tässä diplomityössä kuvaajien generoinnissa käytetään PHPMetricsin oletusrajoja, jotta tulokset ovat vertailtavissa samanaikaisesti keskenään ja avoimen lähdekoodin projektien kanssa.

3.3.2 Vertailu avoimen lähdekoodin projekteihin

Edellä esiteltyjä mittauksia voidaan verrata muihin projekteihin konkreettisemman käsityksen saavuttamiseksi tarkasteltavan järjestelmän laadusta. Tässä diplomityössä mitattiin tunnettuja avoimen lähdekoodin projekteja, koska avoimen lähdekoodin projektien lähdekoodit on helposti löydettävissä.

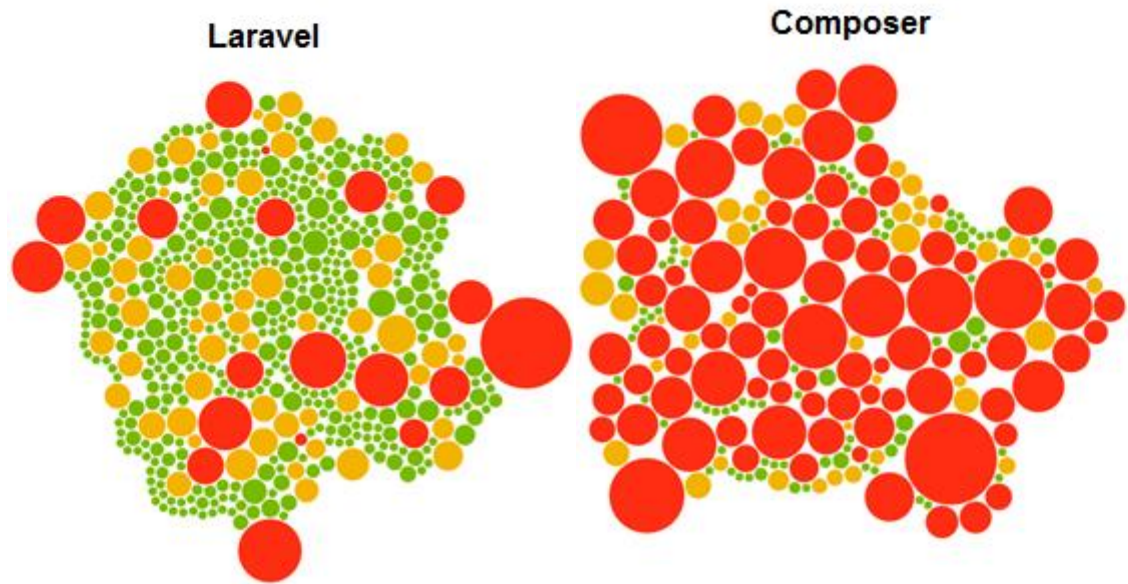
Vertailua varten mitattaviksi avoimen lähdekoodin projekteiksi valikoitui Laravel ja Composer. Mittauksessa mitattiin Laravel-kehyksen version 5.2 Github-hakemiston tiedostoja [32] ja Composerin Github-päähakemistoa [33].

Laravel [34] on suosittu ohjelmistokehys PHP-ohjelmistokehitykseen. Composer [35] on sovellus PHP-kirjastojen projektikohtaiseen paketinhallintaan.

Taulukko 4. *PHPMetrics-työkalun antamat pisteet Laravelin ja Composerin laadusta pisteytettynä välillä 0-100*

	Laravel	Composer
Ylläpidettävyys	86	40
Helppokäyttöisyys	79	2
Algoritmien yksinkertaisuus	30	0
Volyymi	42	0
Bugien todennäköisyyden vähäisyys	54	0

Taulukossa 4 on esitelty PHPMetricsin pisteytykset Laravel- ja Composer-projektien laadusta. Laravel on saanut kokonaisuutena kohtuullisen hyvät pisteet ja Composer heikot. Laravelin vahvuudet ovat erityisesti ylläpidettävyydessä ja helppokäyttöisyydessä, Composer saa kohtuulliset pisteet ainoastaan ylläpidettävyydestä.



Kuva 4. *PHPMetricsin generoima kuvaaja Laravel- ja Composer-projektin lähdekooditiedostojen laadusta*

Kuvassa 4 on havainnollistettu PHPMetricsin generoimilla kuvaajilla Laravel-projektin lähdekooditiedostojen syklomaattista kompleksisuutta ja ylläpidettävyysindeksiä. Kuvasta havaitaan, että Laravel-projektissa lähdekooditiedostot ovat keskimäärin ylläpidettävyysindeksiltään merkittävästi paremmalla tasolla kuin Composer-projektin lähdekooditiedostot.

PHPMetrics-työkalun generoimat taulukoissa 2 ja 4 esitellyt mittarit ovat tarkastelluista järjestelmistä parhaalla tasolla Laravelissa. Composer ja tässä diplomityössä tarkasteltavassa järjestelmässä mittarit ovat keskenään melko vastaavalla, eli heikolla, tasolla.

Tässä diplomityössä tarkasteltava järjestelmä on heikommalla tasolla keskimääräisen syklomaattisen kompleksisuuden ja ylläpidettävyysindeksin osalta kuin kumpikaan tarkastelluista avoimen lähdekoodin projekteista. Laravel on tarkastelluista järjestelmistä näissä selkeästi parhaimmalla tasolla.

Lähtökohtaisesti avoimen lähdekoodin projekteissa koodin ylläpidettävyys ja ymmärrettävyys ovat vieläkin keskeisemmässä asemassa kuin kaupallisissa järjestelmissä. Onkin mielenkiintoista kuinka niinkin yleisesti käytössä oleva ohjelmisto kuin Composer on päästetty koodin laadussa mittauksissa havaitulle tasolle.

Mittaukset havainnollistavat myös hyvin sitä, että PHPMetricsissä on asetettu rajat hyvinkin tiukoiksi. Kumpikaan mitatuista avoimen lähdekoodin projekteista ei pääse lähellekään täysiä pisteitä koodin laadussa. Ei välttämättä ole siis aivan tarkoituksenmukaistakaan tavoitella diplomityössä tarkasteltavassa järjestelmässäkään sitä, että PHPMetrics antaisi jokaisesta mittarista täydet pisteet ja kaikkien tiedostojen ylläpidettävyysindeksi näkyisi generoiduissa kuvaajissa vihreinä ympyröinä.

PHPMetricsiä kannattaa käyttää ennemminkin ongelmallisimpien tiedostojen tunnistamiseen ja järjestelmän kokonaislaadun kartoittamiseen. PHPMetrics ei myöskään korvaa ihmisen päättelykykyä, vaan joskus voi olla tarkoituksenmukaista luoda tiedosto heikollakin ylläpidettävyysindeksillä tai syklomaattisella kompleksisuudella muiden hyötyjen saavuttamiseksi.

4. POTENTIAALISESTI HYÖDYNNETTÄVÄT SUUNNITTELUMALLIT JA REFAKTOROINNIT

Tässä luvussa käsitellään suunnittelumalleja ja hajuja jotka voivat olla oleellisia työssä käsiteltävän ohjelman tapauksessa. Lisäksi tässä luvussa selvitetään trendejä web-ohjelmistojen arkkitehtuureissa, joita käsiteltävässä ohjelmassa voitaisiin hyödyntää.

4.1 Suunnittelumallit

Rakentaja. Rakentaja-suunnittelumallia hyödynnetään erityisesti tilanteissa, joissa luokan rakentaja on muodostumassa turhan monimutkaiseksi. Joskus onkin järkevää jakaa rakentajan logiikkaa selkeämmiksi ja ymmärrettävimmiksi kokonaisuuksiksi useampaan funktioon niin että olion rakennus tapahtuu vaiheittain ennen kuin syntynyt olio voidaan varsinaisesti ottaa käyttöön. Tyypillisesti rakentaja-suunnittelumallissa olion rakentamisen eri vaiheet ovat vaihtoehtoisia yhden tai useamman muiden vaiheiden kanssa, josta johtuen oliota luodessa kutsutaan vain osaa luokan rakentamisfunktioista. Tällaisissa tapauksissa kaikkien rakentamisfunktioiden kutsuminen aiheuttaisi tyypillisesti virhetilanteita. [1]

Adapteri. Adapteri on rakenteellinen suunnittelumalli, joka sovittaa tietyssä muodossa olevan tiedon sellaiseen muotoon, että toisenlaista muotoa vastaanottava rajapinta pystyy käsittelemään sitä [18]. Adapteri-suunnittelumallia voi hyödyntää monenlaisissa eri ongelmissa. Yksi hyvä esimerkki on tilanne, jossa olion käyttötapa on muuttunut merkittävästi alkuperäisestä, mutta olion rakenteen muuttaminen aiheuttaa ongelmia [1]. Adapteri on hyödyllinen myös esimerkiksi kun lähetetään vastaavaa tietoa moneen eri rajapintaan, jotka vastaanottavat tiedon eri muodossa.

Decorator. Decorator on suunnittelumalli, jolla pyritään vastaamaan tilanteeseen, jossa tietyn luokan täytyy toimia hieman poikkeavalla tavalla sen yleisimmästä käyttötarkoituksesta. Decorator-suunnittelumallilla pyritään välttämään perintähierarkiaan liittyvää monimutkaisuutta. Decorator-suunnittelumallia voidaan hyödyntää esimerkiksi tilanteessa, jossa erilaisia ominaisuusyhdistelmiä on useita, eikä haluta luoda omaa luokkaa jokaiselle yhdistelmälle. Käytännössä decorator-suunnittelumallissa luodaan uusi luokka, jonka jäsenmuuttujaksi asetetaan luokka, jonka tilaa halutaan muokata. Decorator-luokat toteuttavat saman rajapinnan kuin luokka, jonka ominaisuuksia decoratorilla pyritään laajentamaan. Saman rajapinnan toteuttaminen on siksi tarpeellista, että useita decorator-luokkia voidaan ketjuttaa ja kutsua vastaavaa metodia ketjun seuraavalta tietämättä tarkalleen, että mikä decorator-luokka on kyseessä. Muokattavan luokan muokattavat jäsenmuuttujat ovat julkisia, jotta decorator pääsee muokkaamaan niitä. [1]

Delegoija. Delegoija-suunnittelumallilla pyritään yksinkertaistamaan vaihtoehtoisia suorituspolkuja delegoimalla luokan päätöksentekoa muille luokille. Delegoija-suunnittelumalli toimii käytännössä niin, että luokalle annetaan parametrina olio, joka tietää miten määritellyssä tilanteessa tulee toimia. Delegoijaluokasta voidaan vain kutsua varsinaisen suorituksen toteuttavan luokan metodia, joka suorittaa tilanteeseen sopivan koodin. Delegoijaluokan ei siis tarvitse varsinaisesti tietää mitä kutsuttava luokka tekee. Delegoija-suunnittelumallia voidaan käyttää esimerkiksi tilanteessa, jossa luokan täytyy palauttaa vastaavaa tietoa erilaisessa muodossa. Luokka, jolle työtä delegoidaan, toteuttaa tiedon järjestämisen oikeassa muodossa ja delegoiva luokka voi vain keskittyä ydinlogiikan suorittamiseen. [1]

Julkisivu. Julkisivu on suunnittelumalli, jolla yksinkertaistetaan monimutkaisen luokan tai monimutkaisten luokkien rajapintoja sopivalle abstraktiotasolle tiettyyn tarpeeseen. Julkisivu-suunnittelumalli helpottaa koodin käsittelyä. Käytännössä Julkisivu-suunnittelumalli toteutetaan luomalla luokka, jonka metodit kutsuvat joukkoa toisten rajapintojen julkisia metodeja. Julkisivu-luokan julkisten metodien kautta kutsutaan tyypillisesti useita abstrahoitavien luokkien metodeita. Se myös saattaa kutsua vain osaa niiden luokkien metodeista, joita se abstrahoi. [18]

4.2 Ohjelman hajujen refaktorointi

Suuri luokka. Riittämätön modularisointi on modularisointihaju, joka aiheutuu liian suurista luokista. Luokan liiallinen suuruus voi johtua liiallisesta funktioiden lukumäärästä tai funktioiden koosta. Luokan koko vaikuttaa negatiivisesti koodin ymmärrettävyyteen, muokattavuuteen ja laajennettavuuteen. Riittämätön modularisointi -hajun saa korjattua tapauksen mukaan jakamalla luokkaa osiin [26]. Joskus luokkiin jako kannattaa toteuttaa hyödyntämällä perintää ja joskus uudet toiminnallisuudet kannattaa siirtää täysin omaan luokkaansa. Luokan metodeita voi tarvittaessa myös siirtää muihin olemassaoleviin luokkiin.

Pitkä metodi. Pitkä metodi on yksinkertaisesti haju, jossa yksittäisessä luokan metodissa on liikaa rivejä. Jo kymmenen rivin mittaisen metodin pituutta voi alkaa kyseenalaistamaan [36]. Liian pitkä metodi syntyy usein kun olemassaolevaan metodiin lisätään ajan mittaan hiljalleen uusia ominaisuuksia, eikä metodia jaeta osiin. Pitkästä metodista täytyy tunnistaa selkeitä kokonaisuuksia, jotka voi jakaa omiin metodeihinsa. Toiminnallisuutta kannattaa tilanteen mukaan siirrellä toisiin luokkiinkin, jotta luotua koodia voi uudelleenkäyttää tehokkaammin.

Monta vastuualuetta sisältävät luokat. Monikasvoinen abstraktio (multifaceted abstraction) on abstrahointihaju, joka syntyy kun luokalla on useita selkeitä vastuualueita. Lähtökohtaisesti yksittäisellä luokalla pitäisi olla vain yksi vastuualue ymmärrettävyyden helpottamiseksi. Kun luokalla on useita vastuualueita, muutosten teko luokkaan on mo-

nimutkaisempaa, joka johtaa helposti entistä suurempaan tekniseen velkaan. Monikasvoinen abstraktio -hajun saa korjattua jakamalla luokan kahteen tai useampaan osaan niin että jokainen vastuualue on omassa luokassaan. Useita vastuualueita sisältävien luokkien havaitsemisen apuna voidaan metriikkatyökalujen Lcom-mittaria. [26]

Pitkä parametrilista. Funktiokutsut pitkillä parametrilistoilla eivät ole riittävän luettavia. Pitkän parametrilistan sisältävässä funktiokutsussa on haastavaa lukea nopeasti, että mitä tietoa funktiolle annetaan ja mikä tieto vastaa mitäkin parametria funktion esittelyssä. Pitkiä parametrilistoja voidaan selkeyttää esimerkiksi parametriluokilla. Jos kaikki tai osa parametreista liittyvät vahvasti toisiinsa, ne voidaan joskus niputtaa yhteen luokkaan ja antaa kyseisen luokan instanssi parametrina. Joskus funktiokutsuja voidaan yksinkertaistaa antamalla se olio parametrina, josta funktiota kutsutaan. [36]

Alkeistyyppien käytön pakkomielle. Alkeistyyppien käytön pakkomielle on koodin haju, jossa yksinkertaisia muuttujia, joihin liittyy suoraan jotain toiminnallisuutta, käytetään sellaisenaan yksinkertaisen luokan sijaan. Alkeistyypit voivat olla esimerkiksi numeroita ja merkkijonoja. Alkeistyyppien käytön pakkomielteeseen liittyy myös vakioiden käyttö tiedon koodaamiseen. Alkeistyyppien käytön pakkomielle syntyy helposti nopeampana ratkaisuna luokan luontiin verrattuna. Esimerkiksi päivämääräväli tehdään helposti kahtena muuttujana luokkaan, vaikka niihin liittyisi suoraan esimerkiksi toiminnallisuus, jolla haetaan päivämäärävälin pituus. [36]

Keskitetty modularisointi. Keskitetty modularisointi (hub-like modularization) on modularisointihaju, joka aiheutuu kun luokka riippuu monesta muusta luokasta ja moni muu luokka riippuu samaisesta luokasta. Riippuvuuksien määrä aiheuttaa haasteita erityisesti muutosten teossa ja ymmärrettävyydessä, koska muutos voi vaikuttaa monella tavalla muokattavasta luokasta riippuviin luokkiin. Lisäksi riippuvuuksien määrä vaikuttaa luokan laajennettavuuteen, uudelleenkäytettävyyteen ja luotettavuuteen. [26]

Syklinen riippuvuus. Syklisesti riippuvainen modularisointi (cyclically-dependent modularization) on modularisointihaju, joka aiheutuu kun luokka on jonkin riippuvuusketjun kautta riippuvainen itsestään. Poikkeustapaus tässä hajussa on tarkoituksenmukaiset rekursiivisuudet. Kun luokka riippuu itsestään, todennäköisyys yllättävien bugien syntymiselle muutoksien yhteydessä kasvaa. Lisäksi syklinen riippuvaisuus aiheuttaa ongelmia koodin ymmärrettävyydessä ja uudelleenkäytettävyydessä. Isoissa ohjelmissa on haastavaa havaita itsestään riippuvaisia moduuleita koska luokissa on paljon erilaisia riippuvuuksia. [26]

4.3 Web-sovellusten arkkitehtuurisuunnittelun trendit

Tässä kohdassa käsitellään tarkasteltavan järjestelmän kannalta mielenkiintoisia ohjelmistokehityksen trendejä. Tarkoituksena on perehtyä teknologioihin alustavasti niin, että

voidaan arvioida mitä kyseinen teknologia mahdollistaa ja sen perusteella mahdollisesti suositella kyseisen teknologian käyttöönottoa.

Tarkasteltavat teknologiat ovat valikoituneet listaan sen perusteella, että ne ovat tulleet järjestelmää kehittäessä vastaan, kun erilaisiin haasteisiin on pyritty löytämään ratkaisua. Kyseessä on trendejä, joita ei tällä hetkellä käytetä tarkasteltavassa järjestelmässä, mutta niitä voitaisiin mahdollisesti tulevaisuudessa käyttää.

Teknologioita käsitellään tässä kohdassa mielenkiintoisuusjärjestyksessä. Tarkasteltavan järjestelmän kannalta mielenkiintoisimpina teknologioina ensimmäisenä käsitellään Mikroservicejä ja Dockereita, jonka jälkeen DevoOpsia. Lopuksi käsitellään Big Dataa, jota on hyvä pitää tulevaisuudessa silmällä järjestelmän tehostamiseksi.

4.3.1 Mikroservicet

Mikroservicet ovat pieniä sovelluksia, joita voidaan ottaa käyttöön, skaalata ja testata itsenäisesti. Mikroservicellä on tyypillisesti yksi selkeä vastuualuekokonaisuus. [37]

Mikroserviceillä pyritään ratkaisemaan suurten monoliittisten ohjelmistojen haasteita. Suurta monoliittista ohjelmaa saattaa olla mielekästä muokata vielä siinä vaiheessa kun se on viety tuotantokäyttöön, mutta vuosien ylläpidon jälkeen monoliittiseen järjestelmään tehtävien muutosten monimutkaisuus voi kasvaa sietämättömän suureksi. Tätä varten monoliittista sovellusta voidaan haluta jakaa toisensa kanssa kommunikoiviin osiin, jolloin saavutetaan helpommin hallittavissa olevampia ja dynaamisempia sovelluskomponentteja. [37]

Mikroservicejen pieni koko tekee koko komponentin korvaamisen toisella tarvittaessa huomattavasti yksinkertaisemmaksi kuin vastaavan toiminnallisuuskokonaisuuden korvaamisen täysin uudella koodilla monoliittisessa järjestelmässä [37]. Mikroservicet helpottavat siis esimerkiksi kokonaisen sovelluskomponentin tekemisen täysin uusiksi toisella ohjelmointikielellä niin, että uusi mikroservice-sovellus toteuttaa samat rajapinnat kuin vanha mikroservice-sovellus.

Mikroservicejen käyttäminen on mahdollista sekä täysin uudessa sovelluksessa, että olemassa olevissa monoliittisissa järjestelmissä. Käytännössä suurin osa mikroservicejä hyödyntävistä sovelluksista ovatkin olleet alun perin monoliittisia järjestelmiä, joiden ongelmia on lähdetty ratkaisemaan mikroservicejen käytöllä. [37]

4.3.2 Docker

Docker on avoimen lähdekoodin työkalu Linux-käyttöjärjestelmän pystytyksen ja siihen tehtävien asennusten automatisointiin [38]. Käytännössä Docker on kevyt käyttöjärjestel-

mäsäiliöiden virtualisointiteknologia [38]. Mikroservice-arkkitehtuurilla luotujen ohjelmien hallinnassa lähes standardiksi muodostunut Docker on suunniteltu kevytkäyttöiseksi ja suoritusteholtaan skaalattavaksi [38]. Skaalautuvuus on mahdollista sekä sen osalta, että samalle palvelimelle voi pystyttää useita Docker-säiliöitä ja toisaalta yksi Docker-säiliö voi käyttää usean palvelimen resursseja.

Docker on myös hyvä ja suosittu teknologia jatkuvan integraation ja jatkuvan käyttöönoton tukemiseen. Koska Dockerin säiliöitä on helppoa kopioida ja siirrellä, on mahdollista monistaa jokin ohjelman asiakasympäristöistä ja testailla siinä koodiin tehtyjä muutoksia käytännöstä saadulla testidatalla. [38]

Verrattuna perinteisiin virtuaalikoneisiin, Dockerin säiliöt ovat myös 10–15 prosenttia tehokkaampia. Docker-säiliöiden avulla on mahdollista saada enemmän palvelimen resursseista ohjelmiston käyttöön ja samalla jakaa palvelimen resurssit dynaamisemmin. [38]

4.3.3 DevOps

DevOps terminä tulee sanoista “development” ja “operations”, jolla kuvataan DevOpsin pyrkimystä tiivistää ohjelmistokehitystä tekevän henkilöstön kuten ohjelmistokehittäjien, testaajien ja laadunhallintahenkilöstön, sekä tuotantohenkilöstön kuten palvelimien, tietoverkkojen ja tietokantojen ylläpidon yhteistyötä. [39]

DevOps on joukko tekniikoita ja hyviä ohjelmistokehitystapoja, joilla pyritään tehostamaan ohjelmistokehitystä. Yksi DevOpsin tavoitteista on pyrkiä kehittämään oppimista palautteen toimivammalla viennillä tuotannosta ohjelmistokehitykseen. Samoin DevOpsilla pyritään kehittämään iteraatioiden läpivientiaikoja samalla parantaen ohjelmiston laatua. DevOpsissa kiinnitetään huomiota esimerkiksi yrityskulttuuriin, automaatioon ja ohjelmistojen laadun mittaamiseen. [39]

Yksi keskeisistä haasteista on ohjelmiston jatkokehityksen ja tuotannon välillä olevista tavoite-eroista. Ohjelmistokehittäjät pyrkivät mahdollisimman nopeaan muutokseen, kun taas tuotannon henkilöstö pyrkii tuotannossa olevien sovellusten vakauteen, josta syntyy muutosvastaisuutta tuotantokäytössä olevien sovellusten muutoksille. Näitä eroja pyritään DevOpsissa ratkaisemaan ottamalla myös tuotanto mukaan ketterään ohjelmistokehitysmenetelmään. [39]

DevOps-tiimin täytyy siis olla moniosaavaa ja tiimin sisällä olevan keskustelun avointa. Yksittäinen tiimi hoitaa sekä ohjelmistokehityksen, testaamisen, laadunvarmistuksen, dokumentoinnin, että tuotannon. Tällä saavutetaan kehityksen ja tuotannon välille me-henki siiloutuneiden ryhmien vastakkainasettelun sijaan. [39]

4.3.4 Big data

Tässä diplomityössä käsiteltävässä järjestelmässä on jo nyt joitakin suurista tietomääristä aiheutuvia haasteita. Erityisesti ongelmia aiheuttaa suurien tietomäärien generoiminen raportteihin, mikä saattaa tietyissä tapauksissa kestää minuutteja. Joskus joissakin tapauksissa raporttien generointiajat ovat olleet jopa lähellä kymmentä minuuttia.

Tähän mennessä eniten generointiaikaa vieneet prosessit on saatu ongelman havaitsemisen jälkeen pääasiassa algoritmeja ja Doctrine-ORM:in käyttöä optimoimalla murto-osaan suoritusajasta. Tulevaisuudessa tietomäärien jatkuvasti kasvaessa ja mahdollisesti yhä isompien yritysten tullessa yrityksen asiakkaiksi on hyvä tiedostaa myös big dataan liittyvät teoriat ja teknologiat, jotta niitä voidaan tarvittaessa hyödyntää.

Big datalla tarkoitetaan tietoa, jota on hyvin paljon, muuttuu hyvin nopeasti ja on hyvin vaihtelevaa [7]. Kyseessä on siis tietoa, jonka käsittely perinteisillä tiedonkäsittelytekniikoilla ja palvelimien tiedonkäsittelykyvyillä riittävän nopeasti on hyvin haastavaa, kallista tai jopa mahdotonta. Big datan käsittelytekniikoilla saadaan mahdollistettua big datan kerääminen, säilöminen, hallinta ja manipulointi riittävillä suoritusajoilla ja sopivilla suoritushetkillä [7].

Kuva 5. *Tiedonjalostusprosessi Big datalla [7]*

Kuten kuvassa 5 on havainnollistettu, tiedonjalostusprosessi big datassa on syklistä. Tiedon jalostus alkaa luonnollisesti datan keräämisestä (Capture), josta se jatkuu datan järjestelyyn (Organize) ja yhdistelyyn (Integrate). Tämän jälkeen dataa tulkitaan (Analyze), jotta kootusta datasta saadaan oikeanlaista tietoa. Kootun tiedon perusteella voidaan tehdä päätöksiä (Act) ja tarkentaa sitä, että millaista dataa halutaan kerätä tulevaisuudessa tarkempien tulosten saavuttamiseksi. [7]

Yksi keskeisistä teknologioista big datan hallinnassa ovat tietovarastot. Tietovarastoihin talletetaan valikoitua rakenteellista tietoa myöhempää käsittelyä varten. Tyypillisesti tietovarastoihin säilöttävä tieto on transaktioista syntyvää dataa. Tietovarastot on eriytetty suorituskykysyistä varsinaisesta järjestelmien päätöksentekoon käytetyistä tietokannoista. [7]

5. KÄYTÄNNÖN KOKEILUT

Tässä luvussa käsitellään sitä, miten diplomityön eteen tehtiin käytännön ohjelmointityönä. Ensimmäinen osa käytäntöä on ongelmallisen luokan refaktorointi. Toisessa osassa kehitettiin kokonaan uusi ominaisuus järjestelmään pyrkien samalla perusteltuihin arkkitehtuuriratkaisuihin.

Tämän luvun teksteissä käytetään me-muotoa, mutta käytännön työhön liittyvät päätökset on tehnyt vain minä jos tekstissä ei ole muuten mainittu.

5.1 FormModel.php:n refaktorointi

FormModel.php ei varsinaisesti ole edes aivan kriittisimmin refaktorointia vaativien tiedostojen joukossa, mutta se on luokka, jonka refaktorointi edesauttaa myöhempää refaktorointia. FormModel.php:hen ja siitä eriytettyihin tiedostoihin täytyy purkaa toiminnallisuutta muista refaktorointia vaativista luokista, eikä se ole helppoa jos FormModel.php on nykyisessä muodossaan.

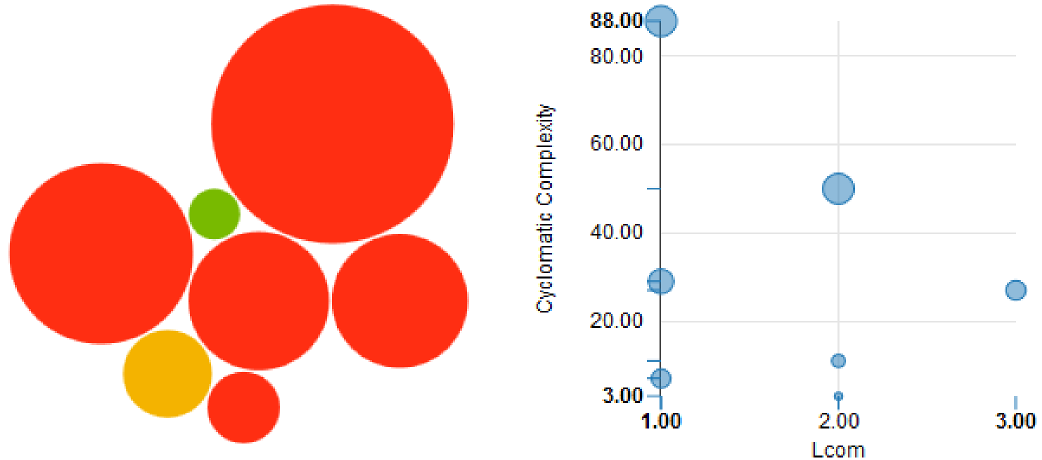
Koska FormModel.php on vielä kohtuullisesti hallittavassa muodossa, on se myös hyvä kohde opetella refaktorointia. Muutokset eivät ole liian monimutkaisia ja luokkaa refaktoroidessa voidaan kokeilla erilaisia ratkaisuja.

PHPMetrics-työkalun perusteella FormModel.php:n lähtötilanteessa syklomaattinen kompleksisuus oli 202 ja Lcom 2. PHPMetrics-työkalun määrittämäksi ylläpidettävyysindeksiksi määräytyi 25.

5.1.1 Ensimmäinen refaktorointi-iteraatio

FormModel.php:ta lähdettiin refaktoroimaan tavoitteena erityisesti luokkien ja metodien jakaminen mielessä niin, että ne ensinnäkin noudattavat määriteltyjä tavoitepituuksia. Lisäksi pyrittiin madaltamaan syklomaattista kompleksisuutta ja panostettiin metodien nimeämiseen niin, että kommenttien käyttö voidaan minimoida. Erityisen todennäköisesti kasvavissa malleissa pyrittiin pitämään luokan pituus runsaasti tavoitetasoja alempana.

Ensimmäisen refaktorointi-iteraation jälkeen luokka oli jaettu seitsemään eri luokkaan. Refaktoroinnin tulosta mitattiin ensimmäisen refaktorointi-iteraation jälkeen uudelleen, kun iteraatiolle asetetut tavoitteet luokkien ja metodien pituuksissa oltiin saavutettu. Metodien nimeämistä ei ole helppoa mitata, joten sen riittävyys täytyi hyväksyä subjektiivisin perustein.



Kuva 6. *PHPMetrics-työkalun generoimat kuvaajat ensimmäisen refaktroiti-iteraation tuloksena syntyneiden luokkien syklomaattisesta kompleksisuudesta, ylläpidettävyysindeksistä ja Lcom-mittarista*

Kuvassa 6 kuvataan ensimmäisestä refaktorointi-iteraatiosta syntyneitä luokkia. Vasemmanpuolimmaisessa kuvaajassa vain yksi ympyrä on vihreä ja yksi keltainen. Muiden ympyröiden väri on punainen tarkoittaen, että niiden ylläpidettävyysindeksi on alle 65. Osa ympyröistä on huomattavasti isompia kuin toiset tarkoittaen, että niiden välillä on huomattavia eroja syklomaattisessa kompleksisuudessa.

Kuvan 6 oikeanpuolimmaisessa osassa x-akselilla kuvataan luokkien Lcom-mittaria, y-akselilla syklomaattista kompleksisuutta ja ympyröiden koolla tiedostojen rivimääriä. Kuvaajasta voidaan havaita, että luokat ovat mitatuilta ominaisuuksiltaan hyvin erilaisia ja vaativat erilaisia jatkorefaktorointeja.

Erityisesti kuvan 6 oikeanpuolimmaisen kuvaajan x-akselilla kuvattava Lcom-mittari herätti pohtimaan, että mikä refaktoroinnin tuloksena syntyneissä luokissa on ongelmana. Luokkien vastuualueita oltiin refaktoroidessa jaettu pääosin vastaavasti kuin muuallakin järjestelmässä niin, että samaan aiheeseen liittyviä toiminnallisuuksia oltiin koottu yhteiseen tiedostoon.

Lcom-mittari havainnollistaa sitä, montako toisiinsa riippumatonta tietovirtaa luokasta löytyy. Jos tietovirtoja löytyy useampi kuin yksi, luokka tulisi jakaa osiin, sillä muuten luokka ei noudata olio-ohjelmointiparadigmaa kovinkaan hyvin. Useita tietovirtoja sisältävät luokat myös paisuvat hallitsemattomiksi erityisen helposti, ja toisiinsa liittymättömättömät funktiot vaikeuttavat koodin luettavuutta.

Luokan sisältämät useat tietovirrat rikkovat myös SOLID-suunnitteluperiaatteen ensimmäistä kohtaa, yhden vastuualueen periaatetta. Järjestelmää suunnitellessa kyseistä periaatetta ei ole sisäistetty riittävän hyvin, vaan on käsitetty todennäköisesti niin, että tietyn selkeän aihealueen käsittely luokassa riittäisi periaatteen riittäväksi noudattamiseksi.

Samaan aiheeseen liittyvien toiminnallisuuksien koonti yhteisiin tiedostoihin ei siis ole hyvä idea. Tämä päätelmä on erityisen tärkeä refaktoroinnin edistämiseksi järjestelmässä. Kaikkien yrityksessä työskentelevien ohjelmistosuunnittelijoiden on ymmärrettävä mihin se perustuu, jotta jatkossa kehitettävässä koodissa asia olisi kunnossa ja vanha koodi kehittyisi hiljalleen oikeaan suuntaan järjestelmällisen refaktoroinnin tuloksena.

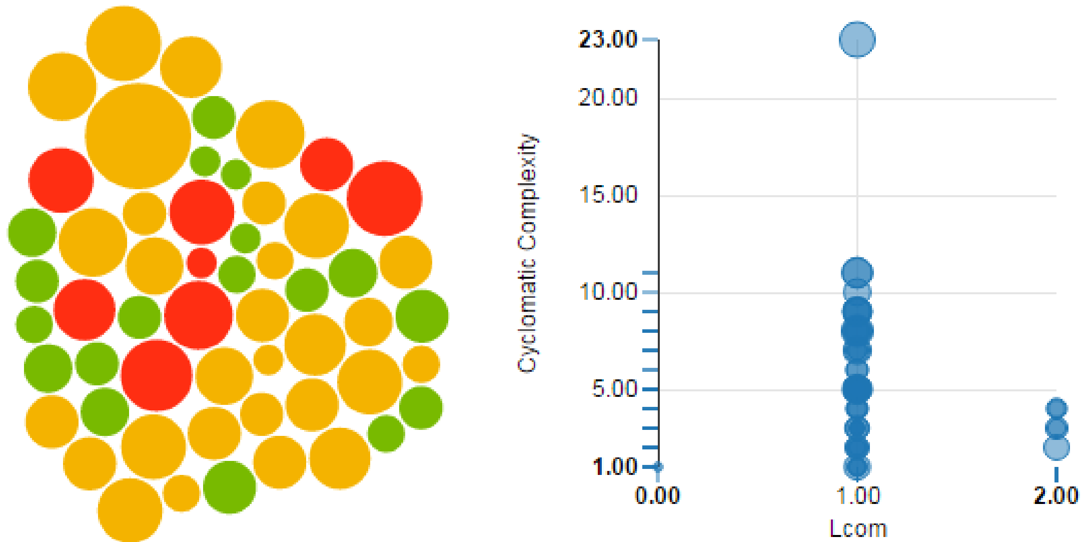
5.1.2 Toinen refaktorointi-iteraatio

Toista refaktorointi-iteraatiota lähdettiin tekemään niin, että kaikkien refaktorointiin liittyvien tiedostojen ylläpidettävyysindeksi olisi PHPMetrics-työkalulla mitattuna parempi kuin 65. Muutoksia ei haluttu tehdä pelkästään mittaustulosten parantamiseksi, vaan haluttiin myös ymmärtää miksi refaktorointitoimet ovat tarpeellisia. Rafaktorointia tehdessä haluttiin esimerkiksi jakaa luokkia selkeämmin niin, että jokaisella olisi selkeä vastuualue.

Refaktoroinnin yhteydessä muutama monimutkaisempi luokka oli haastavaa jakaa perustellusti niin pieniin osiin, että PHPMetrics-työkalun generoima ylläpidettävyysindeksi olisi alittanut tason 65 rajan. Minimirajaa ylläpidettävyysindeksille päätettiin tästä syystä laskea tasolle 60, jottei monimutkaisuutta lisätty turhaan tarpeettomalla luokkien jakamisella.

Toisen refaktorointi-iteraation tuloksena tiedostojen määrä kasvoi 55 tiedostoon. Tiedostojen keskimääräinen pituudeksi muodostui 49 riviä ja metodien keskimääräinen pituudeksi 17 riviä.

PHPMetrics-työkalu generoi toiseen refaktorointi-iteraatioon liittyvistä 55 tiedostosta kuvan 7 mukaiset kuvaajat. Jäljelle jäi kahdeksan tiedostoa, jotka näkyvät kuvaajassa punaisina ympyröinä, eli joiden ylläpidettävyysindeksi alitti 65:n. Keltaisia ympyröitä kuvaajassa on 29 ja vihreitä 18. Ylipäätään refaktoroitu osa järjestelmää on nyt siis ylläpidettävyysindeksinsä osalta kohtuullisessa kunnossa jopa avoimen lähdekoodin standardeillakin.



Kuva 7. *PhpMetrics-työkalun generoimat kuvaajat toisen refaktrointi-iteraation tuloksena syntyneiden luokkien syklomaattisesta kompleksisuudesta, ylläpidettävyysindeksistä ja Lcom-mittarista*

Refaktoroinnin yhteydessä saatiin kehitettyä myös luokkien Lcom-arvoa, eli käytännössä supistettua luokkien vastuualueiden määrää. Viiden tiedoston Lcom-arvo on vielä kaksi ja yhden nolla. Muiden luokkien Lcom-arvoksi saatiin yksi. Lcom-arvoa pyrittiin saamaan tasolle yksi, mutta muutaman luokan tapauksessa se jäi melko luonnollisella tavalla tasolle kaksi.

5.1.3 Refaktoroinnin lopputulos

Laaja refaktorointityö, jota tämän diplomityön yhteydessä tehtiin, oli henkilökohtaisesti hyvin opettavaista ja auttoi kehittymään paremmaksi ohjelmoijaksi. Refaktoroinnin aikana tehdyt ratkaisut toimivat tarkastellulle järjestelmälle hyvänä pohjana tulevalle refaktorointityölle.

Taulukko 5. *Rivimäärän kasvu refaktoroinnin tuloksena*

Vaihe	Luokkien lukumäärä	Metodien lukumäärä	Rivien lukumäärä
Lähtötilanne	1	35	1273
1. refaktorointi	7	88	1661
2. refaktorointi	55	156	2703

Taulukossa 5 on havainnollistettu luokkien, metodien ja rivimäärien kasvua refaktoroidessa. Taulukossa on erityisen mielenkiintoista se, kuinka voimakkaasti rivien lukumäärä kasvoi luokkien ja metodien lukumäärän kasvaessa. Uusien luokkien ja metodien esittelyt lisäsivät runsaasti rivejä, mutta rivien lukumäärä kasvoi myös koodin jäsentelyn väljentämisen tuloksena.

Metodien ja rivien lukumäärien kasvussa täytyy ottaa huomioon myös se, että samaan aikaan kun järjestelmää refaktoroitiin, niin sitä myös jatkokehitettiin. Myös refaktoroitavaan osaan tehtiin joitakin lisäominaisuuksia, jotka vääristävät hieman tilastoja. Käytännössä uudet ominaisuudet ovat kasvattaneet koodirivien määrää noin 100 koodirivillä alkutilanteeseen verrattuna.

Kokonaisuudessaan rivien lukumäärä yli tuplaantui refaktoroinnin tuloksena. Koodin refaktorointi abstraktimmaksi ja väljemmäksi helpottaa koodaajien ajattelutyötä, sekä nopeuttaa koodin ymmärtämistä.

Jos järjestelmää lähdetään refaktoroimaan järjestelmällisesti tämän diplomityön teon yhteydessä opittujen asioiden pohjalta, tulee järjestelmän tiedostojen määrä kasvamaan räjähdysmäisesti. Järjestelmässä on useita tiedostoja joiden koko on yli tuplasti suurempi kuin tässä kohdassa refaktoroitu tiedosto. Kyseiset tiedostot jakaantuvat todennäköisesti siis jopa yli sataan osaan.

Tässä kohdassa toteutettu refaktorointi oli huomattavasti työläämpää kuin etukäteen odotettiin. Eri osat refaktoroitavaa tiedostoa myös vaativat hyvinkin voimakkaasti toisistaan poikkeavia määriä refaktorointia.

5.2 REST-rajapinnan toteuttaminen ulkoisten palveluiden käytettäväksi

Useampi asiakas on toivonut erilaisia toiminnallisuuksia, jotka vaativat tarkasteltavan järjestelmän kommunikointia ulkoisen palvelun kanssa. Varsinkin tapauksissa, joissa tietoa tarjotaan järjestelmästä ulospäin, rajapinnan tarjoaminen pitäisi olla erityisen standardisoitua ja sen perustana olevat arkkitehtuuripäätökset perusteltuja, sekä tarkkaan harkittuja.

Tarkasteltavassa järjestelmässä on jo joitakin toiminnallisuuksia, joissa haetaan muualta tietoa. Järjestelmässä on ollut pitkään esimerkiksi autentikoitumiseen tarkoitettu AD-integraatio, jonka avulla haetaan käyttäjätunnukset asiakkaan keskitetystä käyttäjätunnusten hallinnasta. AD-integraatiolla saadaan automatisoitua käyttäjätunnusten luontia, sekä oikeuksienhallintaa.

Osa asiakkaista käyttää järjestelmää omassa sisäisessä verkossaan tai IP-rajoitetussa ympäristössä, joista kumpaankaan voi olla vaikeaa päästä mobiililaitteilla käytännön työympäristöissä. Eräs asiakas halusi tähän ongelmaan ratkaisun ja lähdimme toteuttamaan sitä idealla, jossa asiakkaan käyttöön tarjottaisiin julkisessa verkossa oleva ympäristö, josta asiakkaan oma ympäristö pystyy hakemaan tietoa. Näin saadaan aikaan tilanne, jossa järjestelmän loppukäyttäjät voivat luoda uutta tietoa julkisessa verkossa, mutta eivät pääse lukemaan sitä.

Kahden erillisen järjestelmän välinen kommunikointi on hyvä tapa opetella tekemään rajapintaa, jota muutkin, jonkin muun yrityksen toteuttamat järjestelmät voivat käyttää. Lopputuloksena syntyvää rajapintaa voidaan mahdollisesti käyttää jo sellaisenaan tarjoamaan tietoa muiden ohjelmistojen hyödynnettäväksi ja se on myös hyvä pohja tulevalle laajemmalle rajapinnalle.

Toistaiseksi virheet ja muutokset rajapinnan määrittelyyn ovat myös sallittavampia, koska rajapintaa käytetään vain itse. Rajapinnan toimivuutta saadaan siis testailtua käytännössä ja vielä hiottua tarpeen mukaan ennemmin kuin siitä tarjotaan tietoa muille järjestelmille.

Luotavaa rajapintaa voidaan hyödyntää myös yksittäisen järjestelmän sisäisesti. Järjestelmän kannattaisi tulevaisuudessa suorittaa logiikkaa entistä enemmän selainpäässä. Tähän hyvänä lähtökohtana on standardoitu ja hyvin dokumentoitu rajapinta.

5.2.1 Arkkitehtuuripäätöksiä

Rajapintaa päätettiin yrityksen ohjelmistokehittäjien kesken käydyn keskustelun jälkeen ruveta toteuttamaan REST-arkkitehtuurin mukaisesti. Ainoana realistisena vaihtoehtona oli SOAP-rajapinta, joka lopulta päätettiin hylätä, jotta luotava rajapinta tarjoaisi paremman yhteensopivuuden, ymmärrettävyyden ja tehokkuustason selainpään kanssa kommunikoinnille.

REST-rajapinnan toteuttamiseksi halusimme tueksi jonkin valmiin kirjaston kutsujen reitityksen. Reitityksen toteuttaminen itse olisi vaatinut työtä ja arkkitehtuuripäätöksiä, joten valmiin ratkaisun hyödyntäminen koettiin hyväksi ideaksi.

Verkosta löytyi joitakin itsenäisiä PHP-reitityskirjastoja, vakuuttavin ehkäpä klein.php [40]. Itsenäisten reitityskirjastojen kehittäminen ei kuitenkaan ole ollut viime vuosina enään kovinkaan aktiivista. Päädyimme siis ohjelmistokehitystiimin yhteispäätöksellä ottamaan rajapintaa varten käyttöön PHP-frameworkiksi kohtuullisen kevyen ja suositun PHP-ohjelmistokehyksen, Laravelin. Laravel on moderni ohjelmistokehys, jonka kehityksessä on kiinnitetty huomiota rajapintojen vaatimiin toiminnallisuuksiin.

Laravelin käyttöönotto oli myös vaikea päätös: vaikka Laravel on ohjelmistokehykseksi kevyt, tulee sen mukana paljon ylimääräistä toiminnallisuutta. Huolta herätti myös se, miten saamme Laravelin toimimaan järjestelmän toimintaa tarvittavilta osin tukevana ohjelmistokehyksenä sen sijaan, että se olisi koko järjestelmää ohjaava ohjelmistokehys.

Laravelin käyttöönotto enemmänkin kirjastona kuin ohjelmistokehyksenä oli lopulta yllättävän helppoa. Se jousti hyvin myös siinä, että tiedostojen kansiorakenne oli hyvin erilainen Laravelin oletuskansiorakenteeseen verrattuna.

Laravel aiheuttaa haasteita myös PHP-versioiden tukemisessa. Laravelin version 5.2 käyttöönotto vaatii vähintään PHP:n version 5.5.9, joka on uudempi kuin muutamat asiakasympäristöissä käytössä olevat PHP-versiot.

REST-rajapinnan käyttöönotto jossakin yleisemmin käytössä olevassa ominaisuudessa vaatii siis toimenpiteitä asiakasympäristöihin, joissa on liian vanha PHP-versio. PHP-version päivitys uudempaan versioon on muutenkin hyvä idea, erityisesti uudempien versioiden paremman tietoturvallisuuden ansiosta.

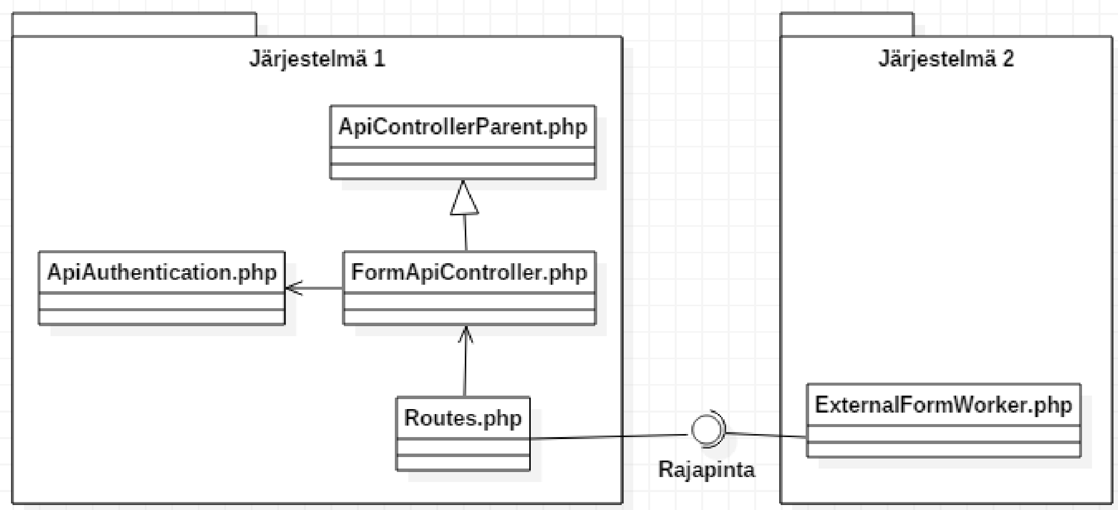
5.2.2 Syntynyt komponentti: rajapinta

Kehitettyyn REST-rajapintaan pystyy autentikoimaan kahdella tavalla: keksien sisältämän tunnisteen perusteella tai tunnuksen ja salasanan yhdistelmällä. Keksit soveltuvat tilanteeseen, jossa REST-rajapintaa käytetään selaimesta tarkasteltavan järjestelmän käyttöliittymän kautta.

Kun REST-rajapintaa käytetään ulkoisesta järjestelmästä, on luontevampaa että tunnus ja salasana annetaan järjestelmään joka kutsun yhteydessä keksien säilömisen sijaan. Tunnus ja salasana annetaan järjestelmälle kutsun headereissa HTTP Basic Authentication -standardin mukaisesti.

Kehitettyyn rajapintaan tuli toistaiseksi vain kaksi kutsua: tietyn kohteen kaikkien tietueiden haku järjestelmästä ja yksittäisen tietueen poisto järjestelmästä. Toinen järjestelmä hakee ensin kaikki kohteen tietueet, lisää ne yksitellen järjestelmään vastaavaan kohteeseen tietokantaan konfiguroidun tiedon mukaisesti. Lopuksi järjestelmä poistaa siirretyt lomakkeet yksitellen järjestelmästä, josta lomakkeet haettiin.

Toisesta järjestelmästä tietoa hakeva järjestelmä kutsuu rajapintaa minuutin välein tarkistaakseen onko toiseen järjestelmään tullut uusia tietueita. Tiedon haussa käytetään paljon olemassa olevia toiminnallisuuksia, joten rajapinnan asiakkaana toimivaan järjestelmään tuli suhteellisen vähän lisäyksiä: uusi taulu tietokantaan, johon on kirjattu rajapintayhteyden tarvitsemat tiedot, rajapintakutsun teko ja rajapinnan palauttamien tietojen parsiminen, sekä oikeassa muodossa olevien tietojen ohjaaminen sopiville kontrollereille.



Kuva 8, *UML-kaavio toteutetusta ominaisuudesta*

Tiedostorakenteen osalta REST-rajapintaan liittyvät reititys- ja kontrolleritiedostot lisättiin kontrollerien kansion alle omaan api-kansioonsa. Kansioon luotiin rajapinnan toiminnallisuuden mahdollistamiseen liittyvät tiedostot Routes.php, ApiControllerParent.php, ApiAuthentication.php ja FormApiController.php kuvan 8 mukaisesti.

Routes.php on järjestelmän ainoa tiedosto, joka hyödyntää Laravel-kehyksen ominaisuuksia. Routes.php:ssa ensin tarkistetaan, että käyttäjä on autentikoitunut, käyttäen ApiAuthentication.php:n toiminnallisuuksia. Tämän jälkeen käytetään Laravelin reititysominaisuuksia, jotta pyyntö saadaan johdettua oikean kontrollerin käsiteltäväksi.

Palvelinosa, jonka rajapintaa asiakkaiden ympäristöissä voidaan käyttää, asetettiin toimimaan omaan, pelkästään rajapintakäyttöön tarkoitettuun ympäristöönsä julkisessa verkossa olevalle palvelimelle. Samaan ympäristöön aiotaan tulevaisuudessa laittaa myös muiden asiakkaiden moduuleita vastaavaan käyttöön.

Luotu toiminnallisuus laitettiin nopeasti asiakkaalle hotfixinä tuotantokäyttöön, koska kyse oli melko kiireellisestä tilaustyöstä. Toiminnallisuudesta on tullut positiivista palautetta ja asiakas oli toiminnallisuuden toteutukseen tyytyväinen.

5.2.3 REST-rajapinnan jatkohyödyntäminen

Järjestelmien väliseen kommunikaatioon luotua REST-rajapintaa on tarkoitus hyödyntää jatkossa merkittävästi enemmän tulevien ominaisuuksien luonnissa ja vanhojen ominaisuuksien jatkokehityksessä.

REST-rajapintaa alettiin hyödyntämään jo eräässä järjestelmän generoimassa kuvaajassa. Kyseinen kuvaaja on aikaisemmin generoitu kuvaksi, mutta siitä haluttiin saada modernimpi. Järjestelmässä käytetään yleisesti HTML5 canvas –pohjaisia kuvaajia, joihin verrattuna vanha ja vanhanaikaisella tavalla generoitava kuvaaja ei ollut enää mielekäs.

Koska viimeisetkin merkittävistä asiakkaista ovat hiljalleen siirtyneet Internet Explorer 8:aa uudempiin selainversioihin, voidaan kuvaajan generointiin käyttää modernimpia SVG-teknologiaa hyödyntäviä kirjastoja. Näistä kuvaajan toteutukseen valikoitui D3js-kirjasto, jonka avulla saa luotua helposti kustomoitavia, visuaalisesti näyttäviä kuvaajia. D3js:lle on olemassa myös paljon avoimen lähdekoodin esimerkkejä erilaisten kuvaajien luontiin.

Uusi kuvaaja hakee REST-rajapinnan kautta dataa, jonka avulla kuvaajan datat generoituvat. Toistaiseksi kuvaajan uusiminen toi uutena ominaisuutena vain sen, että kun sivulla oleva tieto päivittyy, saadaan myös kuvaajan datat päivitettyä vastaamaan sitä.

Kuvaajan saa D3js-kirjaston avulla vietyä tulevaisuudessa uudelle tasolle, sillä SVG-muotoiseen kuvaajaan saa luotua interaktiivisuutta käyttäjän kanssa. D3js-kirjaston SVG-kuvaajiin saa helposti esimerkiksi toiminnon, jossa tiettyä kohtaa klikkaamalla tapahtuu jotain.

6. JÄRJESTELMÄN ARKKITEHTUURIN TULEVAISUUS

Tässä kappaleessa käsitellään ensin sitä, miten aikaisemmin esiteltyjä ongelmakohtia tulisi kehittää. Jokainen kohdan 3.2 alikohta käsitellään kohdassa 6.1 samassa järjestyksessä vaiheittain, jonka lisäksi kyseisen kohdan lopuksi käsitellään hajujen refaktorointia. Tämän jälkeen jatketaan siihen, kuinka suunnittelumalleja tulisi hyödyntää järjestelmässä. Lopuksi kohdassa 6.3 käsitellään erilaisia arkkitehtuuriratkaisuvaihtoehtoja järjestelmän tulevaisuuteen.

6.1 Arkkitehtuurin laadun jatkokehittäminen

Jokaisella järjestelmää kehittävällä ohjelmistosuunnittelijalla on iso vastuu ohjelman arkkitehtuurista. Yhteisiä päätöksiä arkkitehtuurista tehdään hyvin abstraktilla tasolla ja tarkemmat päätökset jäävät kulloinkin kyseessä olevan toiminnallisuuden kehittäjän vastuulle. Tämä on järkevää nykyisessä tilanteessa, jossa yrityksen ohjelmistosuunnittelijoiden kyvyt ohjelmistosuunnittelussa eivät poikkea merkittävästi toisistaan.

Nykyisestä arkkitehtuurisuunnittelumallista ei kannata siirtyä kohti mikromanagerointia. Mikromanageroinnin sijaan kannattaa panostaa ohjelmistosuunnittelijoiden huolellisuuden ja osaamisen kehittämiseen. Ohjelmistosuunnittelussa pitää olla selkeitä sääntöjä, niistä on pidettävä kiinni ja koodin selkeyteen on panostettava aikaisempaa enemmän.

Ohjelmoijien täytyy jakaa osaamista avoimesti ja olla kykeneviä antamaan sekä vastaanottamaan vastaan palautetta. Ohjelmoijien oma mielenkiinto koodin laadun ylläpitämiseen ja uusien asioiden opiskeluun on merkittävässä asemassa ja toimintamalleja on pyrittävä kehittämään jatkuvasti.

6.1.1 Ylläpidettävyys

Järjestelmän jokaisen ohjelmistokehittäjän pitäisi tehdä entistä enemmän refaktorointia normaalin ohjelmistokehityksen rinnalla. Kun ohjelmistosuunnitteluprosessiin otetaan refaktorointi tehokkaammin mukaan, täytyy ohjelmistosuunnittelijoiden opetella samalla uusia asioita. Tällöin koodikatselmoinnit olisi hyvä ottaa ainakin aluksi mukaan prosessiin, jotta osaamista saadaan jaettua tehokkaammin ohjelmistosuunnittelijoiden välillä ja saadaan käsiteltyä erilaisia ratkaisuvaihtoehtoja.

Nyt koodista löytyy runsaasti luokkia, joiden pituus on tuhat tai tuhansia rivejä ja metodeita, joiden pituus on sata tai satoja rivejä koodia. Jatkossa luokkien maksimipituus saisi olla 300 varsinaista koodiriviä ja metodien maksimipituus 30 varsinaista koodiriviä.

Kommentteja ja tyhjiä rivejä ei siis lasketa näihin koodin pituusrajoihin. Lähtökohtaisesti varsinkin luokkien pituuksissa tulisi pyrkiä huomattavasti lyhempiin pituuksiin, vaikka maksimiraja onkin suhteellisen korkealla.

Usein luokkien ja metodien pituutta suositellaan vieläkin lyhyemmiksi, mutta rajoja ei kuitenkaan välttämättä kannata asettaa liian nopeasti hyvin tiukoiksi. Yleisesti kannattaa tavoitella luokkien pituudeksi reilusti alle sataakin riviä riittävän abstraktion saavuttamiseksi.

Pitkiä luokkia ja metodeita on vaikeaa ylläpitää. Pitkän metodin koodiin tehdystä muutoksesta on hyvin vaikeaa tunnistaa, että mihin kaikkialle muutos voisi vaikuttaa. Pitkiin luokkiin kerääntyy myös helposti useita vastuualueita, mikä rikkoo SOLID-suunnitteluperiaatteiden mukaista yhden vastuualueen periaatetta.

Jatkossa refaktoroidessa täytyy kynnys useita vastuualueita sisältävän olion jakamiseen olla paljon pienempi. Useita vastuualueita sisältävien luokkien tunnistamista voidaan tukea koodin metriikoilla, erityisesti tarkkailemalla luokkien Lcom-mittaria.

Yksi osa, jolla järjestelmän ymmärrettävyyttä lähdettiin käytännössä parantamaan, oli luomalla ohjelmoijille yhteinen ohjelmointityyliopas. Ohjelmointityylioppaaseen kirjattiin maksimipituudet luokille ja metodeille, joilla pyritään jatkossa edesauttamaan sitä, etteivät niiden pituudet jatkossa venyisi hallitsemattomaksi. Lisäksi ohjelmointityylioppaassa otetaan kantaa sisennyksiin, kommentteihin, koodin jaksottamiseen tyhjillä riveillä ja sulkeiden sijoittamiseen.

Tulevaisuudessa pitäisi pitää tavoitteena, ettei sisennysten määrä ylitä viittä. Viiteen sisennykseen lasketaan luokan ja metodin esittelyyn käytetyt sisennykset, joten varsinaisia ehtolauseiden tai silmukoiden aiheuttamia sisennyksiä tulisi olla kolme tai vähemmän.

Haasteita ymmärrettävyydelle syntyy myös näkymäosan Smarty-templateissa, joissa on työlästä selvittää, että mitä muuttujia missäkin templatessa on määriteltynä. Lisäksi eroavaisuudet vastaavien muuttujien nimeämisessä eri tiedostoissa aiheuttaa ymmärrettävyysongelmia. Ongelmaa voisi ymmärrettävyyden kannalta dokumentoida käytettävissä olevia muuttujia kommenttina tiedostojen ylälaitaan, mutta toisaalta muuttujien dokumentointi saattaa olla tarpeettoman työlästä.

6.1.2 Suorituskyky

Tietokantahakujen tehostamiseksi voisi olla järkevää tutkia tarkemmin pystytäänkö joissakin tapauksissa hyödyntämään eräajoja tietojen prosessointiin taustalla niin, että käyttäjä saa nopeasti eukäteen prosessoitua tietoa. Toistaiseksi tähän ei kannata kuitenkaan käyttää tuotekehitykseen käytössä olevia rajallisia resursseja, sillä prosessointiajat ovat vielä toistaiseksi olleet jokseenkin hyväksyttävällä tasolla.

Järjestelmän ajamisesta versiosta 6.1 lähtien tulisi vaatia vähintään PHP 5.6:ta ja vuoden kuluttua julkaistavasta versiosta 7.1 lähtien PHP 7:ää. PHP 5.6 tarvitaan välttämättömästi ainakin Laravel-ohjelmistokehityksen ajamiseen, mutta myös tietoturvasyistä. Nykyisin useissa ympäristöissä ajettavaan PHP 5.4:ään ei enää julkaista korjauksia tietoturva-aukkoihin.

PHP 7 kehittäisi suurella todennäköisyydellä huomattavasti järjestelmän suorituskykyä. Useiden eri PHP-ohjelmistojen, -ohjelmistokehysten ja -kirjastojen suorituskyky paranee PHP 7:n myötä kymmeniä prosentteja [41]. Tarkasteltava järjestelmä tietokantakeskeisenä järjestelmänä ei välttämättä muutu nopeammaksi yhtä voimakkaasti kuin suurin osa PHP 7:n tehokkuutta todistaneet ohjelmat.

6.1.3 Helppokäyttöisyys

Tarkasteltavan järjestelmän helppokäyttöisyydessä on paljon edistettävää. Yksi tapa, jolla järjestelmän käytöstä saisi sulavampaa, olisi AJAX-tekniikan tehokkaampi hyödyntäminen. Nyt AJAX:ia käytetään joissakin, pääasiassa kriittisimmissä ominaisuuksissa, kuten lomakkeen tallennukseen liittyvässä lomakelistan latauksessa ilman sivunpäivitystä. Monissa muissa toiminnoissa, erityisesti järjestelmäylläpitonäkymässä, sivu latautuu yksinkertaisten toimintojen yhteydessä turhaan uudelleen.

Joskus latautuva sivu voi sisältää paljon generoituvaa tietoa ja yksinkertaisten toimintojen toistaminen muutaman kerran vie runsaasti aikaa. AJAX-kutsujen tehokkaampi hyödyntäminen parantaisi helppokäyttöisyyden lisäksi myös suorituskykyä, kun samoja raskaita kutsuja ei tarvitse toistaa turhaan.

6.1.4 Hajujen refaktorointi

Koodin hajuja järjestelmässä on haastavaa tunnistaa ohjelmistollisesti, sillä hajujen tunnusmerkit eivät ole välttämättä absoluuttisia, vaan koodin kontekstiin liittyvää. Koodin hajujen refaktorointi saadaan siis käyntiin laajentamalla järjestelmän ohjelmoijien kykyä tunnistaa hajuja.

Ei riitä, että ohjelmoijat vain tietävät koodin hajujen perustunnusmerkit, vaan ohjelmoijien pitää myös ymmärtää miksi koodin haju on koodin haju. Motivaatio välttää koodin hajuja ei ole riittävä ilman riittävää ymmärrystä.

Tässä diplomityössä on pyritty tunnistamaan järjestelmässä toistuvia koodin hajuja, jotta järjestelmän kannalta oleellisimpien koodin hajujen oppimisprosessi saadaan käyntiin. Koodin laadun tarkastelu hajujen näkökulmasta on ollut opettavaista ja auttanut tuottamaan ymmärrettävämpää koodia.

Hajujen tunnistamista opetellessa on välillä hyvä keskustella tiimin kesken, onko tunnistettu tilanne oikeasti haju ja miksi se on haju. Keskustelu tuntuu auttavan ymmärtämään koodin hajuja syvällisemmin. Kun tehdään koodikatselmointia, kannattaa myös ottaa koodin haju-näkökulma mukaan koodin tarkasteluun.

Tarkasteltavassa järjestelmässä merkittävimmät ylläpidettävyyden ongelmien juurisyyt liittyvät luokkien ja metodien pituuteen. Nämä ovat myös yksinkertaisimpia hajuja ymmärtää ja mitata, koska niitä voidaan tarkastella melko konkreettisella tasolla. Se miten pitkien luokkien ja metodien refaktorointi tulisi toteuttaa, onkin haastavampaa. Pelkästään luokkien ja metodien pituusrajoitusten asettaminen ei ole riittävä ratkaisutapa koodin kokonaisvaltaiseen laadun kehittämiseen.

Kun tavoitellaan lyhyitä luokkia ja metodeita, täytyy kiinnittää samalla myös muihin ymmärrettävyyteen liittyviin tyyliseikkoihin. Koodin pituutta voidaan vähentää esimerkiksi laittamalla samalle riville enemmän toimintalogiikkaa, joka ei varsinaisesti helpota koodin ymmärrettävyyttä. Vääränlaisella koodin lyhentämisellä myös koodin jaksottaminen muuttuu helposti tiiviimmäksi ja sitä kautta vaikeammaksi ymmärtää.

Liiallinen yksittäisiin koodin hajuihin keskittyminen saattaa siis aiheuttaa muiden koodin hajujen syntymisen järjestelmään. Koodin hajuja pitääkin tunnistaa laajemmin, jotta järjestelmästä saadaan kokonaisvaltaisesti laadukkaampi.

Yksi koodin haju, joka voi esiintyä itsenäisenäkin, mutta liittyy usein liian pitkiin luokkiin, on monta vastuualuetta sisältävät luokat. Tarkasteltavassa järjestelmässä on monia luokkia, joissa kootaan tiettyyn aihealueeseen liittyviä toiminnallisuuksia samaan valtavaan luokkaan. Tämä aiheuttaa taas pitkien metodien syntymisen, jotta tietyn vastuualueen toiminnallisuudet saadaan saman metodin sisälle. Käytännössä tarkasteltavassa järjestelmässä tämänkaltaiset luokat täytyisi jakaa jokaisen julkisen metodin osalta omaksi luokakseen tai jopa useiksi luokiksi.

Koodissa esiintyy useita metodeita, joilla on turhan pitkiä parametrilistoja. Tämäkin haju liittyy tarkasteltavassa järjestelmässä usein erityisesti metodien pituudessa olevaan ongelmaan: metodien pitäisi olla omia luokkiaan, jolloin myös luokkia instantioidessa suurempien tietomäärien alustaminen luokille olisi luonnollisempaa.

Tarkasteltavassa järjestelmässä esiintyy myös vähemmissä määrin hajua “Alkeistyyppien käytön pakkomielle”. Yksinkertaisia ja mahdollisesti ryhmiteltäviä tietueita, joihin liittyy suoraan kevyttä toiminnallisuutta, saatetaan käsitellä luokassa suoraan sellaisenaan. Koodia pitäisi olla valmiimpi jakamaan pieniinkin luokkiin koodin ymmärrettävyyden parantamiseksi.

Järjestelmässä esiintyy jokseenkin myös keskitettyä modularisointia. Käytännössä se johtuu liian pitkistä luokista, joita käytetään useista luokista ja jotka pituutensa vuoksi riippuvat monista muista luokista.

Järjestelmässä ei ole ainakaan toistaiseksi tunnistettuja ongelmia syklisen riippuvuuden kanssa, mutta se on mahdollinen suurien luokkien ja metodien aiheuttama sivuvaikutus. Syklinen riippuvuus pitääkin pystyä tunnistamaan herkemmin, sillä siitä voi aiheutua poikkeuksellisen hämmentäviä ja vaikeasti tunnistettavia ongelmia.

6.1.5 Refaktorointitaktiikka

Refaktoroinnille täytyisi varata huomattavasti nykyistä enemmän aikaa. Refaktorointia täytyisi lähitulevaisuudessa tehdä erityisen paljon, jotta tässä työssä havaitut ongelmat saataisiin korjattua mahdollisimman nopeasti. Mitä aikaisemmin mittavat refaktorointityöt tehdään, sitä vähemmän tunnistetut koodissa olevat ongelmat tulevat aiheuttamaan ongelmia tulevaisuudessa.

Käytännössä refaktorointia ei voida tehdä niin paljoa kuin järjestelmä tällä hetkellä vaatisi. Järjestelmään täytyy jatkuvasti kehittää uusia ominaisuuksia liiketoiminnallisista syistä ja asiakkaiden toiminnallisten vaatimusten tyydyttämiseksi. Refaktorointia voidaan kuitenkin tehdä paljon nykyistä enemmän ja sillä saavutetaan pitkällä tähtäimellä hyötyjä asiakastyytyväisyyteenkin laadun parantuessa.

Järjestelmää kehittäessä kannattaisi tehdä refaktorointia sekä täysin omana prosessinaan, että uusien ominaisuuksien yhteydessä. Niitä tehdään toki jo nytkin, mutta aivan liian vähän.

Aikaa pelkästään refaktoroinnille kannattaa varata erityisen paljon varsinkin järjestelmän suurimpien refaktorointitöiden alussa, josta erillisen refaktoroinnin määrää voidaan hiljalleen laskea isoimpien ongelmien korjaamisen jälkeen. Erikseen refaktoroinnille varattua aikaa pitäisi olla aina runsaasti mukana ohjelmistokehityksessä.

Usein helpoin ja järkevin hetki tehdä refaktorointia on uuden toiminnallisuuden toteutuksen yhdessä. H. Liun, Y. Liun, G. Xuen ja Y. Gaon toteuttaman tutkimuksen [27] mukaan käytännössä jopa 86 % ohjelmistoprojekteissa tehtävästä refaktoroinnista tehdään toiminnallisten muutosten yhteydessä. Myös tarkasteltavassa järjestelmässä kannattaa panostaa refaktoroinnin tuomiseen mukaan jokaisen järjestelmän ohjelmoijan päivittäiseen ohjelmointityöhön.

Joskus erikseen refaktoroinnille säästettyä aikaa voidaan hyödyntää myös uusien toiminnallisuuksien tuontiin loppukäyttäjille nopeasti. Jos ohjelmistokehityksessä on riittävä luottamus siihen, että refaktoroinnille varataan aikaa, voidaan toteuttaa ominaisuus loppukäyttäjälle käyttöön nopeasti ja refaktoroida koodi riittävälle laatutasolle myöhemmin. Tämä tuo myös joustavuutta ominaisuuksien kokeiluille. [27]

6.2 Suunnittelumallien hyödyntäminen

Suunnittelumallien käytön opettelu on haastavaa. Erityisen haasteellista suunnittelumallien käytön opettelussa on se, että miten tunnistaa milloin käyttää suunnittelumallia, jota on käyttänyt joko harvoin tai ei ikinä. Suunnittelumallin käytöstä voi olla enemmän haittaa kuin hyötyä, jos sitä erehtyy hyödyntämään väärässä paikassa tai väärällä tavalla.

Suunnittelumallien sisäistämiseen tarvitaankin teoriapohjan lisäksi käytännön kokemusta ja niissä toimii hyvin myös virheiden kautta oppiminen. Virheiden tekeminen kaupallisessa ohjelmistossa ei ole mieluisaa, joten kynnys suunnittelumallien ensimmäisiin käyttökertoihin on korkea.

Toisaalta suunnittelumalleja kannattaisi opetella käyttämään matalan prioriteetin koodilohkoissa, joissa muutokset ovat epätodennäköisiä ja koodin laatuun liittyvät riskit pieniä. Toisaalta taas suunnittelumallien yksi vahvuuksista on joustavuus muutoksien yhteydessä, jonka vuoksi tällöin ei saavutettaisi kaikkia suunnittelumallin tuomia hyötyjä ja tehtäisiin mahdollisesti koodilohkosta tarpeettoman monimutkainen.

Rakentaja-suunnittelumallia on jo käytetty järjestelmässä muutamissa osissa järjestelmän koodia. Rakentaja-suunnittelumallin käyttö on toiminut hyvin esimerkiksi monista asioista riippuvaisten monimutkaisten SQL-kyselyiden rakentamiseen. Jatkossakin sitä kannattaa käyttää vastaavissa tilanteissa.

Adapteri-suunnittelumallille ei toistaiseksi ole erityisen selkeitä käyttökohteita. Lähtökohtaisesti järjestelmän sisällä kannattaa ylläpidettävyyssyystä muuttaa koko luokka vastaanottamaan standardinmukaista tietoa sen sijaan, että tehtäisiin adaptereita vanhentuneen luokan kanssa kommunikointiin. Sen sijaan tulevaisuudessa integraatiotarpeiden kasvaessa tulee todennäköisesti tarvetta käyttää adapteri-suunnittelumallia ulkoisten järjestelmien kanssa kommunikoidessa.

Decorator-suunnittelumallia voisi todennäköisesti käyttää useammassakin kohtaa järjestelmässä, mutta kyseisen suunnittelumallin tapaukset ovat niin yksilötapauksia, että niistä on haastavampaa tehdä minkäänlaisia selkeitä laajempia linjauksia tai ehdotuksia siitä, missä niitä pitäisi käyttää. Sen sijaan ohjelmoijien pitäisi hallita decorator-suunnittelumalli ja käyttää sitä itsenäisesti sopivan tilanteen tullessa vastaan.

Delegoija-suunnittelumalli olisi mielenkiintoinen ainakin tilanteissa, joissa saatetaan haluta metodin palauttavan Doctrine-olioita tai taulukon riippuen tilanteen suorituskykyvaatimuksista. Doctrine-oliot ovat lähtökohtaisesti helppokäyttöisempiä ja ymmärrettävämpiä monimutkaisten tietojen käsittelyyn, mutta massiivisempien tietomäärien kanssa taulukkomuotoon haettavat tiedot ovat prosessoinnin osalta kevyempiä käsitellä.

Julkisivu-suunnittelumallia tulisi käyttää yleisesti järjestelmässä useissa paikoissa. Koodia pitäisi saada hallittavammaksi ja ymmärrettävämmäksi, mitä Julkisivu-suunnittelumallin hyödyntäminen edistäisi.

6.3 Tulevaisuudessa hyödynnettävät teknologiat ja toimintatavat

Tarkasteltavassa järjestelmässä on jätetty vielä hyödyntämättä monia moderneja web-sovelluskehityksen tekniikoita, teknologioita ja trendejä. Niitä toki kannattaakin hyödyntää vain, jos niistä koetaan saavutettavan riittävästi hyötyä, sillä niiden opetteluun saattaa mennä runsaasti aikaa. Myös varovaisuus on hyvästä, eikä kaikkiin trendeihin kannata lähteä mukaan, sillä kaikki trendit eivät ole pysyviä ja saatetaan havaita pitkässä juoksussa jopa haitallisiksi.

Tällä hetkellä web-sovelluskehityksen trendeistä järjestelmän tulevaisuuden tavoitteita vaikuttaisi tukevan parhaiten microservicet ja Docker. Myös DevOpsiin liittyviä hyväksi havaittuja tapoja kannattaisi mahdollisesti ottaa mukaan prosessiin. Myöhemmin sovelluksessa saatetaan tarvita big dataan liittyviä tekniikoita.

Microserviceillä pystytään kehittämään järjestelmän ylläpidettävyyttä ja skaalautuvuutta. Microservice-komponenttien pystytyksen täytyy onnistua automaattisesti, sillä jo nyt ympäristöjen pystytys ja päivitys on työlästä ja Microservicejen käyttöönotto voisi moninkertaistaa niihin vaaditun työmäärän. Tähän ratkaisuna toimii Dockerit, joilla asiakasympäristöjen automatisointi ja virtualisointi saataisiin uudelle tasolle.

Jo nytkin asiakasympäristöjen pystyttämiseen ja ylläpitoon kuluvaa aikaa saataisiin pienennettyä huomattavasti Dockerin avulla. Docker auttaisi myös palvelinresurssien nykyistä dynaamisemmassa hallinnassa. Palvelinresurssit saataisiin tehokkaammin käyttöön, kun asiakasympäristöt voivat käyttää minkä tahansa palvelimen resursseja juuri niin paljon kuin sille on tarvetta.

Yksi tässä diplomityössä käsiteltävän järjestelmän tulevaisuudensuunnitelmista on järjestelmän myyminen asiakkaille helposti suoraan internetistä. Tähän Docker olisi erinomainen työkalu, sillä ympäristön pystytys saataisiin sen avulla automatisoitua ja varattua ympäristölle palvelimelta sen tarvitsemat resurssit.

DevOpsin periaatteiden ottaminen mukaan ohjelmistokehitysprosessiin ainakin osittain voisi mahdollisesti olla hyvä idea. Jo nykyisellään DevOpsin periaatetta moniosaavalle tiimille noudatetaan pienen tiimikoon vuoksi. Kaikki tuotekehitystiimissä olevat kolme henkilöä tekevät tuotekehitystä ja kaksi henkilöä tiimistä tekevät myös tuotantoon liittyviä tehtäviä. Tuotekehitystiimiin kuuluva henkilöstö tekee laaja-alaisesti erilaisia tehtäviä. Painotukset siinä, että mitä kukakin tekee, poikkeaa kuitenkin toisistaan.

Järjestelmässä käytetään jo nyt automaatiotestausta, mutta sitä kannattaisi hyödyntää DevOpsin mukaisesti nykyistäkin vahvemmin perustoiminnallisuuksien toiminnan varmistukseen.

Sovelluksen tehokkuusvaatimusten kasvaessa merkittäväksi teknologiaksi saattaa nousta myös big data. Ainakin toistaiseksi tehokkuutta saadaan kuitenkin kehitettyä riittävästi koodia ja tietokantakyselyjä optimoimalla. Ensimmäisiä kohteita big datan hyödyntämiselle tulee todennäköisesti olemaan järjestelmän raportointiominaisuudet, jossa tietoa saattaa joutua käsittelemään suuria määriä jokaisen sivulatauksen yhteydessä.

7. YHTEENVETO JA JOHTOPÄÄTÖKSET

Tämän diplomityön toteuttamisen aikana saatiin kehitettyä järjestelmän osien laatua, lisättyä ohjelmistokehitystiimin osaamista ja luotua uutta toiminnallisuutta. Diplomityön teko on ollut hyvin opettavainen arkkitehtuurin suunnittelun, ohjelmistosuunnittelun ja ohjelmointityylin osalta. Kirjallisuudesta oppi paljon uutta teoriaa ja ohjelmistoalan perusasiatkin jäivät kirjallisuudesta entistä vahvemmin mieleen. Erityisesti refaktorointia sisältänyt käytännön osuus oli opettavainen, sillä sitä tehdessä havaitsi käytännössä kuinka tiettyjen perinteisten olio-ohjelmoinnin suunnitteluperiaatteiden heikko noudattaminen vaikuttaa ohjelman laatuun.

Suunnittelumalleja tutkiessa havaittiin, että suunnittelumallien tietoinen hyödyntäminen on järjestelmässä nykyisin hyvin vähäistä. Suunnittelumalleja tulisi käyttää nykyistä enemmän, jotta saadaan toteutettua ratkaisuja yleisesti hyväksi todetuilla tavoilla, sekä saadaan lisättyä sanastoa ohjelmoijien välille. Suunnittelumallien tehokkaampi hyödyntäminen vaatii ohjelmoijilta itsenäistä opiskelua ja suunnittelumallien osaamisen jakamista tiimin kesken.

Tarkasteltavassa järjestelmässä esiintyy useita koodien hajuja, joiden vähentämiseen täytyy käyttää entistä enemmän työpanosta. Erityisen ongelmallisia koodin hajuja järjestelmässä on pitkät luokat, pitkät metodit ja useita vastuualueita sisältävät luokat. Järjestelmään on kerääntynyt paljon teknistä velkaa ja lähtökohtaisestikin refaktorointi saisi olla keskeisempi osa ohjelmistokehitystyötä, jottei teknisen velan määrä ainakaan kasvaisi. Refaktorointia tulisi tehdä enemmän sekä normaalin ohjelmointityön ohessa, että itsenäisenä työnä tiettyjen ongelmallisten osa-alueiden kehittämiseksi. Ainakin refaktoroinnin määrän kasvatuksen alkuvaiheissa olisi hyvä ottaa prosessiin mukaan koodikatselmoinnit, jolla saataisiin tuettua refaktoroinnin opettelua vertaispalautteella.

Järjestelmän tulevaisuudensuunnitelmien osalta kannattaa pitää mielessä erityisesti Microservicet ja Dockerit. Microserviceillä saa jaettua järjestelmää useaan, itsenäisesti toimivaan osaan, joita on helpompi hallita. Microservicejen avulla järjestelmän eri osia voidaan tarvittaessa toteuttaa eri teknologioilla. Microservicejen käyttöönotto voi tapahtua hiljalleen niin, että ensin eriytetään järjestelmästä vain yksi pieni osa omaksi komponentikseen. Microservicejen käyttöönotto tekisi järjestelmien pystyttämisestä ja päivittämisestä aikaisempaa työläämpää. Järjestelmien pystytystä ja päivittämistä saataisiin kuitenkin automatisoitua Docker-työkalun avulla, joka on työkalu palvelinympäristöjen virtualisointiin ja automatisointiin.

LÄHTEET

[1] A. Saray, Professional PHP Design Patterns, Wiley Publishing Inc, Indianapolis, Indiana, USA, 2009, 288 p.

[2] P. Vora, Web Application Design Patterns, Elsevier Inc., Massachusetts, USA, 2009, 448 p.

[3] K. Jamsa, Introduction to Web Development Using HTML 5, Jones & Bartlett Learning, Boston, Massachusetts, USA, 2014, 590 p.

[4] Microsoft Edge, verkkosivu. Saatavissa (viitattu 9.4.2016): https://www.microsoft.com/en-us/windows/microsoft-edge

[5] D. Powers, PHP Solutions - Dynamic Web Design Made Easy, Third Edition, Apress, New York, New York, 2014, 528 p.

[6] B. Brinzarea-Iamandi, C. Darie, A. Hendrix, AJAX and PHP: Building modern web applications (toinen painos), Packt Publishing Ltd, Birmingham, UK, 2009, 294 p.

[7] J. Hurwitz, A. Nugent, F. Halper, M. Kaufman, Big Data For Dummies, John Wiley & Sons, Inc., Hoboken, New Jersey, USA, 2013, 336 p.

[8] jQuery, verkkosivu. Saatavissa (viitattu 5.3.2016): https://jquery.com/

[9] AngularJS, verkkosivu. Saatavissa (viitattu 5.3.2016): https://angularjs.org/

[10] O. Campesato, K. Nilson, Web 2.0 Fundamentals: With AJAX, Development Tools, and Mobile Platforms, Jones and Bartlett publishers, Sudbury, Massachusetts, USA, 2011, 791 p.

[11] K. Dunglas, Persistence in PHP with the Doctrine ORM, Packt Publishing Ltd, Birmingham, UK, 2013, 103 p.

[12] S. Ambler, The Elements of UML 2.0 Style, Cambridge University Press, New York City, New York, USA, 2005, 202 p.

[13] S. Datta, Metrics-Driven Enterprise Software Development, J. Ross Publishing, Fort Lauderdale, Florida, USA, 2007, 304 p.

[14] L. Westfall, The Certified Sofware Quality Engineer Handbook, ASQ Quality Press, Milwaukee, Wisconsin, USA, 2009, 132 p.

[15] F. Tsui, O. Karam, B. Bernal, Essentials of Software Engineering (kolmas painos), Jones & Bartlett Learning, Burlington, Massachusetts, USA, 2014, 345 p.

[16] L. Laird, M. Brennan, Software Measurement and Estimation: A Practical Approach, A John Wiley & Sons, Inc., Publication, Hoboken, New Jersey, USA, 2006, 274 p.

[17] J. Governor, D. Hinchcliffe, D. Nickull, Web 2.0 Architectures, O'Reilly Media Inc., Sebastopol, California, USA, 2009, 250 p.

[18] S. Holzner, Design Patterns for Dummies, Wiley Publishing Inc., Indianapolis, Indiana, USA, 2006, 308 p.

[19] D. Odell, Pro JavaScript Development – Coding, Capabilities, and Tooling, Apress, New York City, New York, USA, 2014, 454 p.

[20] W. Brown, R. Malveau, H. McCormick III, T. Mowbray, AntiPatterns: Refactoring Software, Architectures and Projects in Crisis, John Wiley & Sons, Inc., New York City, New York, USA, 1998, 309 p.

[21] C. Pitt, Pro PHP MVC, Apress, New York City, New York, USA, 2012, 500 p.

[22] S. Abeysinghe, RESTful PHP Web Services, Packt Publishing Ltd., Birmingham, UK, 2008, 220 p.

[23] E. Guiness, Ace the Programming Interview – 160 Questions and Answers for Success, John Wiley & Sons, Inc., Indianapolis, Indiana, USA, 2013, 445 p.

[24] P. Tripathy, K. Naik, Software Evolution and Maintenance: A Practitioner's Approach, John Wiley & Sons Inc., Hoboken, New Jersey, USA, 2015, 416 p.

[25] S. Jarzabek, Effective Software Maintenance and Evolution - A Reuse Based Approach, Taylor & Francis Group, Boca Raton, Florida, USA, 2007, 424 p.

[26] G. Suryanarayana, G. Samarthyam, T Sharma, Refactoring for Software Design Smells: Managing Technical Debt, Elsevier Inc, Massachusetts, USA, 2015, 258 p.

[27] H. Liu, Y. Liu, G. Xue, Y. Gao, Case study on software refactoring tactics, IET Software, Vol. 8, No. 1, 2014, pp. 1-11.

[28] Apache, verkkosivu. Saatavissa (viitattu 5.3.2016): http://www.apache.org/

[29] Doctrine-project, verkkosivu. Saatavissa (viitattu 5.3.2016): http://www.doctrine-project.org/

[30] Smarty, verkkosivu. Saatavissa (viitattu 5.3.2016): http://www.smarty.net/, viitattu 5.3.2016

[31] PHPMetrics, verkkosivu. Saatavissa (viitattu 5.3.2016): http://www.phpmetrics.org/

[32] Laravel-kehyksen GitHub-sivu, verkkosivu. Saatavissa (viitattu 3.4.2016): https://github.com/laravel/framework

[33] Composer-työkalun GitHub-sivu, verkkosivu. Saatavissa (viitattu 3.4.2016): https://github.com/composer/composer

[34] Laravel, verkkosivu. Saatavissa (viitattu 5.3.2016): https://laravel.com/

[35] Composer, verkkosivu. Saatavissa (viitattu 3.4.2016): https://getcomposer.org/

[36] Refactoring, verkkosivu. Saatavissa (viitattu 16.1.2016): https://refactoring.guru/

[37] J. Thönes, Microservices, IEEE Software, Vol 32, No. 1, 2015, pp. 113-116

[38] C. Anderson, Docker, IEEE Software, Vol. 32, No. 3, 2015, pp. 102–104.

[39] M. Hüttermann, DevOps for Developers, Apress, New York City, New York, USA, 2012, 184 p.

[40] Klein.php:n GitHub-sivu, verkkosivu. Saatavissa (viitattu 5.3.2016): https://github.com/chriso/klein.php

[41] PHP News Archive, verkkosivu. Saatavissa (viitattu 11.4.2016): http://php.net/archive/2015.php