Panu Leppäniemi

Ohjelmistoprojektin lähdekoodin visualisointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

6.4.2017

Metropolia

Tekijä	Panu Leppäniemi
Otsikko	Ohjelmistoprojektin lähdekoodin visualisointi
Sivumäärä	32 sivua + 1 liite
Aika	6.4.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja	Lehtori Olli Hämäläinen

Insinöörityön tarkoituksena oli tarkastella ohjelmistoprojektin laajuuden ja monimutkaisuuden hahmottamista eri sidosryhmien näkökulmasta sekä kokeilla, voiko lähdekoodia visualisoimalla helpottaa kokonaiskuvan muodostumista ohjelmistoprojektista. Työn motivaationa oli havainto, ettei jo 1960- ja 1970-luvulla esitettyjä ohjelmointiin liittyviä edistyksellisiä ajatuksia (esimerkiksi visualisoinnin hyödyntäminen osana ohjelmointia) näy vieläkään arkipäivässä.

Projekti tehtiin itsenäisenä toteutuksena, mutta sen tuloksena syntynyttä prototyyppisovellusta testattiin oikeiden asiakasprojektien lähdekoodilla. Insinöörityöprosessi jakautui taustatutkimukseen, toteutukseen ja jatkokehitysideoiden pohtimiseen.

Taustatutkimuksessa käytiin läpi erilaisia ohjelmistojen visualisointiin painottuvia ohjelmistoja. Lisäksi pohdittiin ohjelmistojen käyttötarkoituksia, hyötyjä ja mahdollisia puutteita insinöörityön tavoitteiden näkökulmasta. Ohjelmistoista saatiin ideoita prototyyppisovelluksen käyttöliittymää varten.

Insinöörityössä toteutettiin prototyyppisovellus, jonka avulla voitiin todeta, onko ohjelmistokoodin sisäisten riippuvuuksien lukeminen ylipäätään mahdollista Scala-ohjelmointikielellä ja minkälaisia hyötyjä toteutuksesta saattaa olla. Prototyyppiin osa-alueisiin kuuluu ohjelma, jolla Scala-ohjelmointikielellä toteutetun ohjelmistoprojektin sisäiset riippuvuudet voidaan lukea. Lisäksi toteutettiin selainpohjainen käyttöliittymä, joka visualisoi nämä riippuvuudet.

Insinöörityö antoi perspektiiviä siihen, miten ohjelmistokehitys on edennyt teknologian osalta vuosien mittaan. Konkreettinen tulos oli ohjelmistoprojektin lähdekoodin visualisointisovelluksen prototyyppi. Yksi sovelluksen tunnistettu hyöty oli subjektiivinen kokemus kohdeprojektin lähdekoodin kokonaisuuksien hahmottamisen helppoudesta. Lisäksi erilaisten lähestymistapojen mahdollisuuksien vertailu suhteessa orgaanisesti ajan myötä syntyneisiin ja vakiintuneisiin ohjelmoinnin perusmekanismeihin tuotti ideoita siitä, miten jatkossa voidaan paremmin tukea lähdekoodin rakenteen hahmottamista ja suunnittelua. Esimerkiksi uusia paradigmoja, kuten virtuaalista tai laajennettua todellisuutta, hyödyntämällä voitaisiin tukea ohjelmointia sekä ohjelmistojen laajuuden ja monimutkaisuuden hahmottamista.

Avainsanat	visualisointi, ohjelmistoprojekti, lähdekoodi, Scala

Author	Panu Leppäniemi
Title	Source code visualisation in software projects
Number of Pages	32 pages + 1 appendix
Date	Thursday 6th April, 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor	Olli Hämäläinen, Senior Lecturer

Programming has come far since its inception. Nowadays software is used as a foundation for numerous businesses. Yet its inner workings are presented as linear text and visible only to programmers. Therefore the purpose of this thesis was to explore different visual approaches to perceiving software projects' source code.

The process of the thesis consisted of background research, implementation of a software prototype and gathering of new ideas. During the research a number of different computer applications which visualise software or source code were analysed. The analysis focused on use cases, benefits and shortcomings of these applications. This resulted in ideas for the prototype's user interface. A working visualisation software prototype was implemented. The prototype is capable of deducing internal dependencies of software implemented in the Scala programming language. It has a web-based user interface which visualises the dependencies. It was tested with actual software projects.

The implemented visualisation software proved to be useful by making it easier to perceive an overall view of a project's source code. It also inspired new ideas. For example by adopting new paradigms such as virtual or augmented reality one could support the understanding of software complexity.

Keywords	visualization, software project, source code, Scala

Sisällys

1 Johdanto 1

2 Lähdekoodin visualisointiin painottuvia ohjelmistoja 2

2.1 CodeCity – lähdekoodin visualisointiohjelmisto 2
2.2 Gource – versionhallinnan visualisointiohjelmisto 3
2.3 MySQL Workbench – tietokantasuunnitteluohjelmisto 4
2.4 Flowhub – visualisaatiopainotteinen ohjelmointiympäristö 5

3 Lähdekoodin visualisoinnin hyödyt ja sovellusalueet 7

3.1 Kokonaisuuksien ja riippuvuuksien hahmottaminen 7
3.2 Kompleksisuuden ja teknisen velan hallitseminen 8
3.3 Koodikatselmointi 9
3.4 Arkkitehtuurin dokumentointi 10
3.5 Kiireelliset ongelmanselvitystilanteet 11
3.6 Läpinäkyvyys muille sidosryhmille 11

4 Työkalu lähdekoodin visualisointiin 13

4.1 Teknologiavalinnat 13
4.2 Scala-kääntäjän vaiheet 16
4.3 Prototyypin arkkitehtuuri 17
4.4 Käyttöliittymä 18

5 Tulokset, ideat ja nykyisen toteutuksen puutteet 22

5.1 Ideat ja uudet sovelluskohteet 23
5.1.1 Visualisaation yhdistäminen versionhallintaan 23
5.1.2 Ohjelmistokoodin tuottaminen visualisaation pohjalta 24
5.1.3 Laajemman kuvan tarkastelu ja tekniset yksityiskohdat 25
5.1.4 Lisätyn todellisuuden tai virtuaalitodellisuuden hyödyntäminen 27
5.2 Nykyisen toteutuksen tiedostetut puutteet 28

6 Yhteenveto 30

Lähteet 31

Liitteet

Liite *Scalac*-ohjelman komentorivitulosteen mukaiset kääntövaiheet

1 Johdanto

Insinöörityön tarkoituksena on perehtyä ohjelmistoprojektin lähdekoodin visualisointiin. Syyt aiheen valintaan ovat moninaisia, mutta omalla mielenkiinnolla on suurin vaikutus. Kun tutkii jo 1960- ja 1970-luvulla esitettyjä ohjelmointiin liittyviä ajatuksia ja toteutuksia, voi vain harmitella, ettei ole päästy nykypäivän tilannetta pidemmälle siinä, miten lähdekoodi ylipäätään nähdään ja miten sitä esitetään ja käsitellään.

Työn tavoitteena on helpottaa ohjelmistoprojektin laajuuden ja monimutkaisuuden hahmottamista. Ohjelmistot ja niiden lähdekoodi ovat nykyään tärkeä osa yritysten liiketoimintaa, elleivät jopa itse liiketoiminta. Tästä syystä on järkevää laajentaa ymmärrystä projektin lähdekoodista ja sen tilasta muillekin kuin ohjelmistokehittäjille: avata uusi näkymä projektin (ei-teknisille) tuoteomistajille.

Haaste on yksittäistä insinöörityötä isompi, mutta toivottavasti työ avaa ajatuksia ja raivaa tilaa tuleville ratkaisuille. Työssä tehdään prototyyppi, jonka avulla voidaan alustavasti kokeilla, onko visualisointi ylipäätään mahdollista toteuttaa ja päästä siihen pisteeseen, että toteutuksesta alkaa olla jonkinlaista hyötyä. Prototyyppiä on tarkoitus testata tosielämän asiakasprojektin lähdekoodilla.

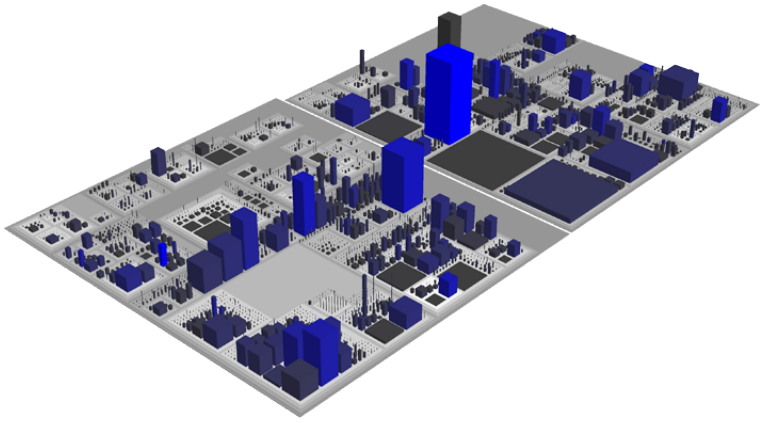
Insinöörityön prototyypin soveltuvuus on rajattu staattisesti tyypitetyllä Scala-ohjelmointikielellä toteutettuihin projekteihin. Tästä huolimatta varsinaisesta visualisoinnista vastuussa oleva käyttöliittymätoteutus ei ole sidottu yksittäiseen ohjelmointikieleen, joten sitä voisi tulevaisuudessa hyödyntää erilaisilla teknologioilla rakennetuissa projekteissa.

2 Lähdekoodin visualisointiin painottuvia ohjelmistoja

Insinöörityön tausta-ajatuksena on lähestyä ohjelmistoprojektin lähdekoodia perinteisestä lineaarisesta tekstipohjaisesta esityksestä poikkeavalla tavalla; tässä tapauksessa visualisoimalla. Tällaisia ajatuksia on esitetty jo pitkään. Tässä luvussa perehdytään muutamaan visualisointipainotteiseen työkaluun. Lisäksi pohditaan työkalujen käyttötarkoituksia ja omakohtaisesti todettuja hyötyjä ja puutteita.

2.1 CodeCity – lähdekoodin visualisointiohjelmisto

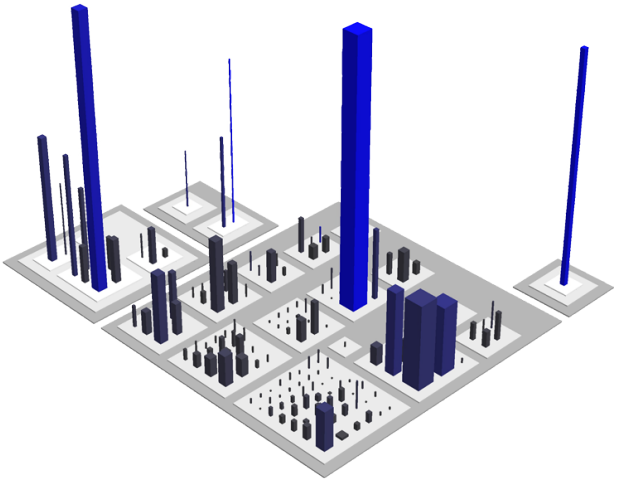
CodeCity on työkalu, joka visualisoi ohjelmiston interaktiivisena, navigoitavana ja kolmiulotteisena kaupunkina (kuva 1). Se on kehitetty akateemisena tutkimusprojektina, eikä sitä voi käyttää kaupallisissa sovelluksissa [1].



Kuva 1: CodeCity-visualisaatio: Java Development Kit (JDK, versio 1.5) [1].

Lähestymistapa on mielenkiintoinen. Varsinkin kokonaan erillisten projektien vertailu paljastaa nopeasti eroavaisuuksia koodirakenteissa (kuva 2): projekti voi sisältää lukumääräisesti harvoja, mutta koodimäärältään suuria luokkia tai funktioita, tai päinvastoin.

Ongelma on se, ettei visualisaatiossa korosteta riippuvuuksia, jotka synnyttävät kompleksisuutta. Vähiten koodia sisältävä komponentti voi olla kriittisin ja eniten muiden komponenttien käytössä. Huomionarvoista on myös, että asioiden absoluuttinen mitta ei käy mistään ilmi, vaan visuaalisaatio nojaa suhteellisuuteen.



Kuva 2: CodeCity-visualisaatio: CodeCity (versio 1.303) [1].

Vertaamalla CodeCityn luomaa kaupunkivisualisaatiota todelliseen kaupunkiin huomataan, että siitä puuttuu oleellinen elementti: tiet tai kadut. Teiden ja katujen tarkoituksena voisi olla riippuvuuksien esittäminen eri rakennusten (eli komponenttien) välillä. Visualisaatiosta ei selviä, mikä komponentti kutsuu mitäkin.

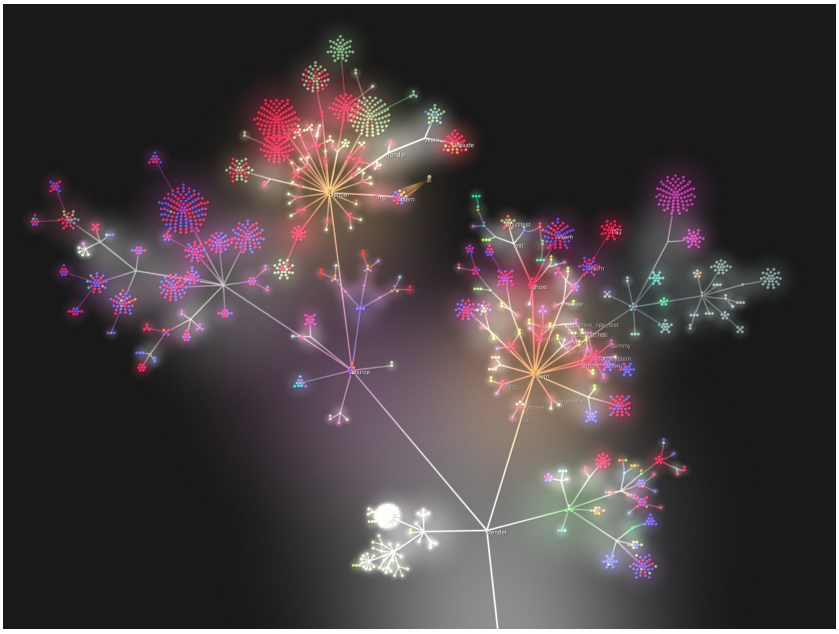
2.2 Gource – versionhallinnan visualisointiohjelmisto

Gource on OpenGL-pohjainen ohjelmistoprojektin versionhallinnan visualisointityökalu. Se on avointa lähdekoodia, ja sen alkuperäinen kehittäjä on Andrew Caudwell [2].

Gource visualisoi versionhallinnan seuraamat muutokset ohjelmistossa animaatioksi, jota voisi kuvata termillä uusinta. Se tarvitsee syötteeksi versionhallinnan (esim. Git) historiatiedot. Gource tukee useita versionhallintajärjestelmiä; tuettujen listalla ovat Git-versionhallintajärjestelmän lisäksi Mercurial, Bazaar ja SVN [3].

Animaatiossa näkyvät muutokset tiedostoihin sekä muutoksentekijät eli ohjelmistokehittäjät, jotka työskentelevät projektin lähdekoodin parissa. Esitystä voi kelata, hidastaa ja nopeuttaa. Gourcen esittämä puumainen rakenne muodostuu projektin kansio- ja tiedostohierarkian mukaan (kuva 3).

Gourcen visualisaatio kallistuu enemmän viihteelliseksi esitykseksi kuin hyödylliseksi tiedonvälittäjäksi. Mutta harvoin asiakas on innostunut koodista niin paljon kuin projektin edistysaskelia Gourcen avulla esitettäessä.



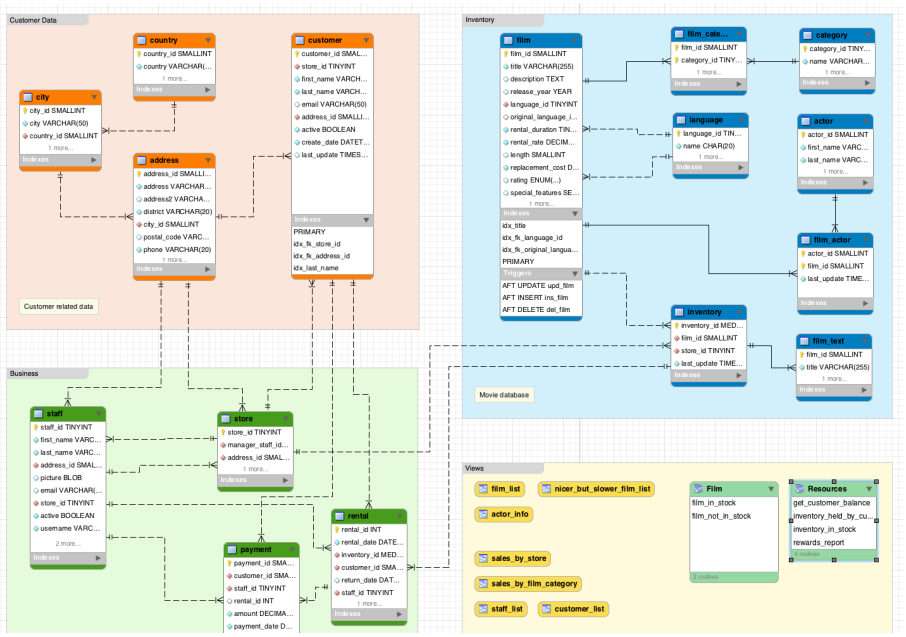
Kuva 3: Blender-grafiikkaohjelmiston versionhallinnan Gource-visualisaatio [3].

Kun versionhallinnan historian animointi on ajettu Gourcen läpi, ei uusia muutoksia enää ilmaannu näytölle. Tämä mahdollistaa ohjelmistokoodin rakenteen tarkastelun hallitummin (vaikkakin tiedosto- ja kansiolähtöisesti). Tarkastelimme asiakkaan kanssa projektin lähdekoodia animaation jälkeen. Staattisemman visualisaation avulla pystyin osoittamaan tiedostot, jotka sisälsivät päänvaivaa aiheuttavaa ohjelmistokoodia. Nämä ”tautipesäkkeet”olivat vain hyvin pieni osa lähdekoodista. Silti niiden vaikutus koko projektiin oli huomattava, koska moni muu asia oli riippuvainen niistä. Tämä ei Gourcen esityksestä näkynyt, mutta asiakas ymmärsi tilanteen.

2.3 MySQL Workbench – tietokantasuunnitteluohjelmisto

MySQL Workbench on ohjelma, jolla voidaan suunnitella ja toteuttaa relaatiopohjaisia tietokantoja. Se sisältää esimerkiksi seuraavia työkaluja: tietokannan graafinen suunnittelu (ER-notaatioon pohjautuen), SQL-editori ja performanssioptimointia avustavat visualisaatiot [4].

Monet tietokantatyökalut nojaavat visualisointiin (kuten MySQL Workbench) ja tarjoavat luontevan tavan suunnitella ja rakentaa tietomalleja (kuva 4). Vaikka taulukkojen attribuuttien määrittelyt ovatkin teknisiä, käyttäjän ei tarvitse kirjoittaa lopullisia SQL-lausekkeita, vaan ne voidaan generoida graafisen suunnitelman pohjalta.



Kuva 4: MySQL Workbenchin esimerkkiprojekti [4].

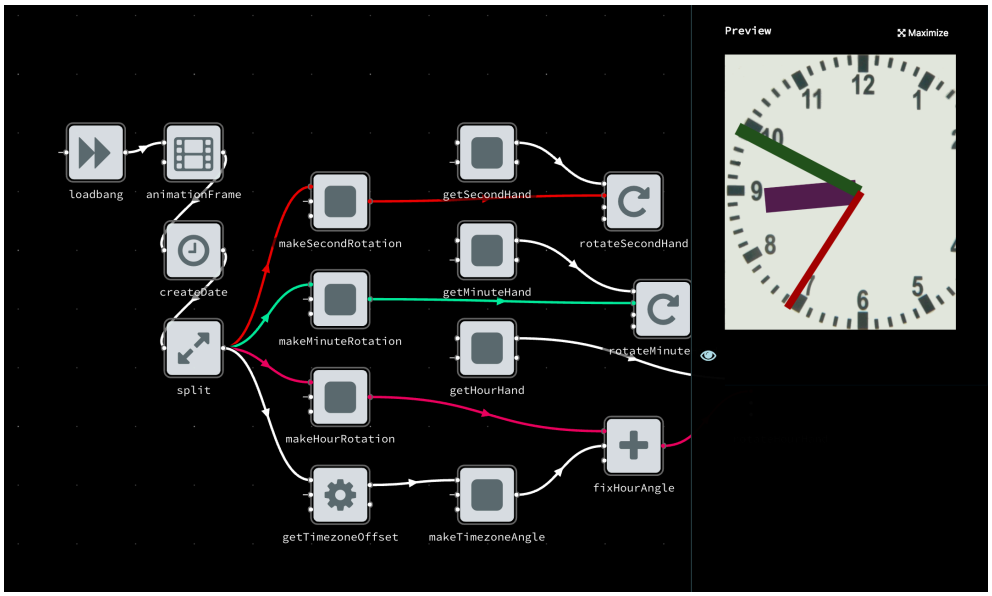
Tietokannan rakenteen tutkiminen on usein ensimmäisiä asioita, mitä tehdään, kun halutaan hahmottaa jo olemassa olevaa projektia. Säilytettävän tiedon tarve ja siten sen rakenne vaikuttaisivat muuttuvan harvemmin kuin koodi. Poikkeuksena ovat dokumenttipohjaiset tietokannat (NoSQL), joissa rakenteita ei tarvitse määritellä etukäteen tietokantatasolla, vaan rakenteet muodostuvat tallennetun datan perusteella.

2.4 Flowhub – visualisaatiopainotteinen ohjelmointiympäristö

Flowhub on tämän joukon erilaisin ja siten ehkäpä mielenkiintoisin tapaus. Se tarjoaa tavan rakentaa ohjelmistoja visuaalisesti. Se tukee esimerkiksi JavaScript-kielellä toteutettua NoFlo-kirjastoa. NoFlo lainaa elementtejä John Paul Morrisonin vuonna 1969 esittelemästä ohjelmointikonseptista nimeltä Flow-Based Programming [5].

Flow-Based Programming, FBP, on ohjelmointiparadigma, joka nojaa vahvasti graafisuuteen (vaikkakaan ei pakota siihen). FBP määrittää ohjelmistot mustien laatikkojen verkostoina, jotka kommunikoivat keskenään välittämällä dataa esimääriteltyjen yhteyksien kautta [6]. Mustalla laatikolla tarkoitetaan sitä, ettei yhden komponentin tarvitse tietää toisen komponentin sisäisestä toteutuksesta, vaan kommunikaatio perustuu komponenttien syötteisiin ja ulostuloihin (joita voi olla useita komponenttia kohden [7]). Ohjelmistot voidaan rakentaa yksittäinen pala kerrallaan, ja palat voidaan yhdistää toisiinsa graafissa

(kuva 5).



Kuva 5: NoFlo-esimerkkiprojekti Flowhub-palvelussa [8].

Sen sijaan, että Flowhub-palvelua voisi hyödyntää perinteisemmän ohjelmointiparadigman kanssa, se vaatii, että lähdekoodi kirjoitetaan tietyllä tavalla, tietynlaisia mekanismeja käyttäen. Tämä saattaa vaikeuttaa sen käyttöönottoa esimerkiksi jo olemassa olevissa projekteissa sekä lisätä kynnystä sen käyttöönottoon uusissa projekteissa.

3 Lähdekoodin visualisoinnin hyödyt ja sovellusalueet

Tässä luvussa kuvataan lähdekoodin visualisoinnin sovelluskohteita ja sillä saavutettavia hypoteettisia hyötyjä. Hyötyjen objektiivinen mittaaminen on rajattu tämän työn ulkopuolelle. Ohjelmistoihin liittyvät työkalut tehdään yleensä ensisijaisesti ohjelmistokehittäjien tarpeisiin. Lähestymistapa on lähes aina tekninen. Lähdekoodin visualisointia ja sen hyötyjä tarkastellaan myös liiketoiminnallisesta näkökulmasta.

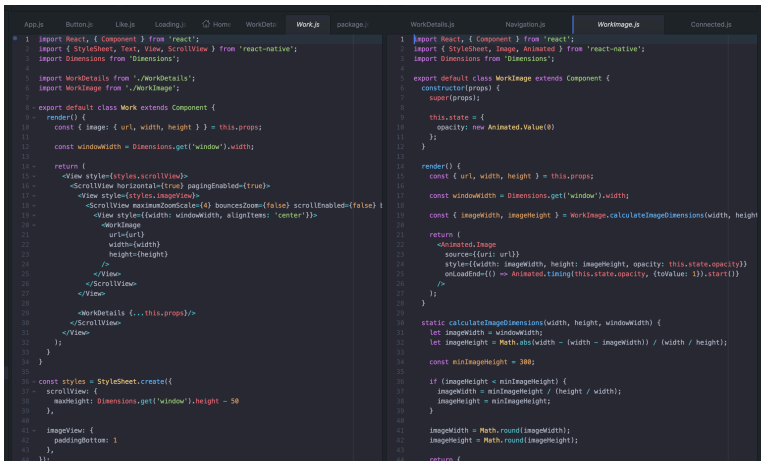
3.1 Kokonaisuuksien ja riippuvuuksien hahmottaminen

Ohjelmistot ovat kompleksisia kokonaisuuksia. Ohjelmistoprojekti voi koostua tuhansista riveistä lähdekoodia. Lisäksi lähdekoodissa esiintyvien osien riippuvuuksien määrä kasvaa nopeasti koodipohjan mukana: luokka A viittaa luokkaan B, joka vuorostaan kutsuu C- ja D-luokkia ja niin edelleen.

Lähdekoodi on jaettu tiedostoihin ja ilmaistaan lineaarisena tekstinä. Lineaarinen esitysmuoto on historian painolastia, niin sanottujen reikänauhojen ja konekielen ajalta [9]. Koodieditoreissa tiedostojen tai tiedostossa olevan lähdekoodin välisiä riippuvuuksia ei tuoda esille, vaan riippuvuuksien ymmärtämiseksi lähdekoodi on luettava läpi.

Ihmisen työmuistiin mahtuu tutkimusten mukaan neljä ”mieltämisyksikköä”kerrallaan [10]. Lähdekoodin todellisia riippuvuuksia voi olla hankala hahmottaa turvautumalla pelkästään lukemiseen ja muistiin. Tältä kannalta on myös mielenkiintoista, miten lähdekoodi esitetään: pieniä osia kerrallaan mahdollisesti isoistakin kokonaisuuksista (kuva 6).

Projektin riippuvuuksien ja kokonaisuuksien hahmottamisesta on hyötyä esimerkiksi työmäärien arvioinnissa. Arviointi voi olla helpompaa, kun tiedetään, mihin kaikkiin ohjelmiston osa-alueisiin muutos saattaa vaikuttaa. Kun tämä ymmärretään, saatetaan myös pystyä minimoimaan muutoksesta aiheutuvan ohjelmistovirheen riskiä.



Kuva 6: Atom-koodieditori.

3.2 Kompleksisuuden ja teknisen velan hallitseminen

Voidakseen ohjelmoida tuottavasti on ymmärrettävä lähdekoodia, jonka parissa työskennellään [11, s. 15]. Ohjelmistojen kompleksisuus synnyttää kustannuksia niin ohjelmistovirheiden kuin hidastuneen kehityksen vuoksi. Kompleksisuus muodostuu ajan myötä ohjelmiston kehittyessä orgaanisesti. Sitä on hallittava samalla tavalla kuin ohjelmiston teknistä velkaa. Teknisellä velalla tarkoitetaan tilannetta, joka voi syntyä, jos laiminlyödään kolmea periaatetta:

1. samaa asiaa tekevän tai saman koodin poistaminen
2. lähdekoodin yksinkertaistaminen
3. lähdekoodin tarkoituksen selventäminen (esim. kommentointi tai selkeyttäminen uudelleennimeämällä) [12, s. 15].

Tarpeetonta kompleksisuutta saattaa muodostua vahingossa, esimerkiksi uusina riippuvuuksina. Kompleksisuuden määritelmä voi olla esimerkiksi seuraavanlainen: kuinka vaikea komponentin tai järjestelmän määrittely ja/tai sisäinen rakenne on ymmärtää, ylläpitää ja todentaa [13]. Kompleksisuuden mittaamiseen on työkaluja, mutta tuloksena saatetaan esittää pelkästään numeerinen arvo. Erilaisia mittareita ovat esimerkiksi syklomaattinen kompleksisuus tai oliometriikat (kuten yhteenkuuluvuus, sidonnat tai luokkahierarkiaan liittyvät metriikat [14]).

Tarkastellaan edellä mainituista esimerkiksi syklomaattista kompleksisuutta. Syklomaattinen kompleksisuus kuvaa ohjelmassa esiintyvien lineaaristen kontrollirakenteiden mää-

rää [13; 15]. Syklomaattinen kompleksisuus M muodostetaan seuraavalla kaavalla:

$$M = E - N + 2P \tag{1}$$

E on linkkien/reunojen lukumäärä graafissa

N on solmujen lukumäärä graafissa

P on yhteydessä olevien (mutta ulkopuolisten) komponenttien määrä.

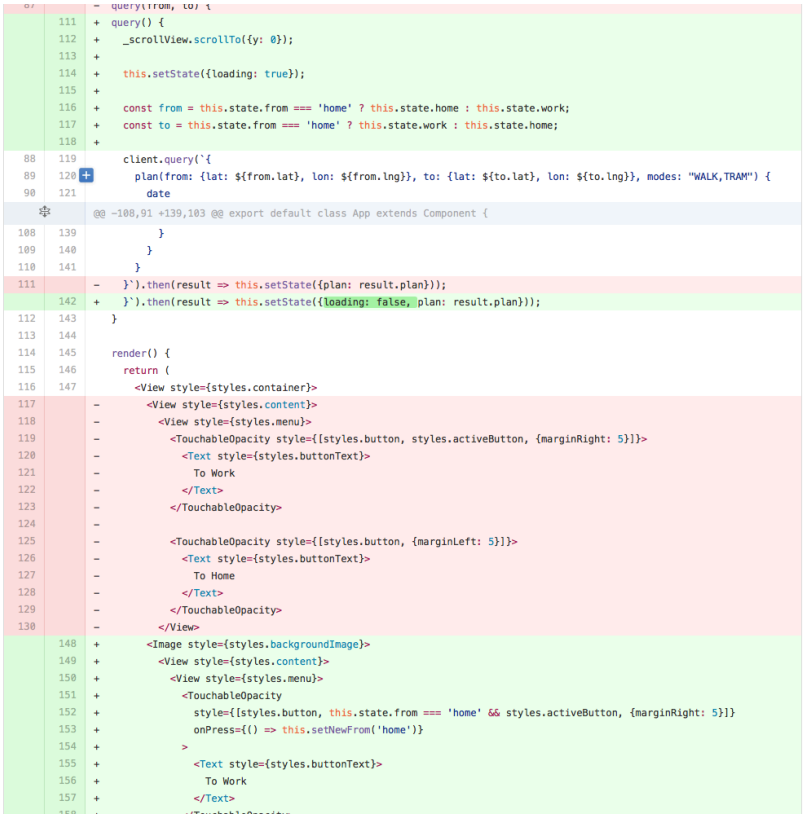
Kaavan 1 voi todeta olevan objektiivinen, mitattava näkemys projektista ja sen lähdekoodin tilasta. Silti yksittäisesta luvusta ja sen muutoksesta saattaa olla vaikea hahmottaa todellista vaikutusta projektin todelliseen, vaikkakin subjektiiviseen, laatuun. Tällaista numeerista mittaria ei kannata pitää pääasiallisena ohjaavana tekijänä ohjelmiston laadulle.

3.3 Koodikatselmointi

Kokemuksien perusteella voidaan todeta, että koodikatselmointi on hyödyllistä [16]. Hyödyllisyys ei kuitenkaan ole vakio. Siihen vaikuttavat muun muassa seuraavat asiat: tiimin yhteistyökyky, jäsenten taitojen vaihtelu, projektin pituus ja katselmoitavan kokonaisuuden laajuus.

Koodikatselmoinnin yhteydessä esitettyjä kommentteja luettaessa on havaittu, että iso osa kommenteista liittyy konventioihin [16]. Projektin konventioiden (esim. koodin muotoiluseikat) seuraaminen on tärkeää, mutta samalla yksinkertaista ja järkevää automatisoida; tietokone tekee tämänkaltaista mekaanista työtä paljon paremmin kuin ihminen. Näin jää aikaa ja energiaa keskittyä tärkeämpiin asioihin, kuten eri osa-alueiden riippuvuuksiin ja isompaan kuvaan. Tämä on kuitenkin haastavaa, varsinkin jos katselmoitava kokonaisuus on laaja.

Esimerkiksi Git-versionhallintaohjelmistoa hyödyntävä GitHub-palvelu esittää lähdekoodiin kohdistuneet muutokset tekstitasolla (kuva 7). Tekstimuotoisessa katselmoinnissa, jossa muutoksia usein tarkastellaan yksittäisten tiedostojen kautta, implisiittiset riippuvuudet ja vaikuttavuudet voivat helposti jäädä piiloon.



Kuva 7: Versionhallinnassa olevan lähdekoodin muutosten vertailu GitHub-palvelussa.

3.4 Arkkitehtuurin dokumentointi

Ohjelmistokoodin arkkitehtuurin kuvaileminen tai kommentointi tekstipohjaisesti on todella raskasta niin lukijalle kuin kirjoittajallekin. Tekstillä voidaan viestiä yksityiskohtia, esimerkiksi sitä, minkälaisen syötteen rajapinta haluaa, mutta arkkitehtuurin kuvailuun siitä ei ole.

UML-kaaviot (Unified Modeling Language), tai UML:n kaltaiset kaaviot, täyttävät tämänkaltaisen korkean tason kuvailutarpeen paremmin. Ne voivat toimia hyvin nopean suunnittelun ja suunnitelman kommunikoinnin välineenä (esim. luokkakaavion muodossa). Jatkuvasti kehittyvän sovelluksen UML-mallien ylläpitäminen ei välttämättä ole hyödyllistä tai kustannustehokasta. UML on harvoin todellinen kuva varsinaisesta järjestelmästä vaan ainoastaan mielikuva tai hahmotelma, joka voi olla virheellinen (huom. poikkeuksena lähdekoodista generoidut UML-kaaviot). Varsinaisen ongelman ratkaisemisen sijaan UML-kaavioilla yritetään usein hoitaa oireita [17]. Olisiko mahdollista löytää jokin käytännönläheisempi visualisaatio, joka soveltuu ja joustaa jatkuvan muutoksen kontekstissa avustaen kokonaisuuksien hahmottamista mutta menemättä liian tekniselle tasolle?

3.5 Kiireelliset ongelmanselvitystilanteet

Valitettavasti ohjelmistovirheet selviävät joskus vasta tuotantoympäristössä, oikeiden käyttäjien löytäminä. Jos ohjelmistovirhe on kriittinen, sen aiheuttamat kustannukset voivat olla suuriakin (tilausta ei pääse tekemään, laskutoimitukset menevät väärin, tieto ei tallennu tietokantaan). Tämänkaltaisessa tilanteessa on välttämätöntä korjata ohjelmistovirhe nopeasti. Vaikka usein ongelman suurpiirteisen syyn saattaakin pystyä arvaamaan, varsinaisen ohjelmistovirheen paikantaminen voi kuitenkin olla hidasta.

Koodia muutettaessa vaaran tunne ei ole aina ilmiselvää. Voivatko nopealiikkeiset muutokset tai korjaukset aiheuttaa uusia ohjelmistovirheitä tai regressioita? Linkkien näkeminen eri osa-alueiden välillä voi helpottaa perimmäisen syyn paikantamista ja auttaa säätelemään tarvittavan varovaisuuden tasoa.

3.6 Läpinäkyvyys muille sidosryhmille

Ohjelmistoprojektin moottori eli lähdekoodi on usein ainoastaan ohjelmistokehitystiimin nähtävissä ja ymmärrettävissä. Tämä ei välttämättä ole huono asia, mutta pahimmassa tapauksessa se voi estää järkevän liiketoiminnan.

Käytännössä valitettavan usein todetaan asiakkaan rakennuttaneen tai ostaneen sovelluksen, joka päällisin puolin näyttää tavanomaiselta, mutta pinnan alla onkin ongelmia: sekavaa, ohjelmistosuunnittelun periaatteita noudattamatonta koodia, jonka keskinäiset riippuvuudet muodostavat umpisolmuja ja haurasta spagettikoodia.

Ongelmat saattavat nousta pinnan alta esille hidastuneena kehityksenä, kasvavina kustannuksina tai tiheämmin ilmenevinä ohjelmistovirheinä. Pahimmillaan heikkolaatuinen koodipohja jarruttaa tai jopa estää liiketoiminnallisia muutoksia. Miten siis tuoda projektin teknistä tilaa keskeisille sidosryhmille näkyviin, jotta mahdollisesti vaadittavat korjausliikkeet ovat vielä hallittavissa?

Yksi sidosryhmistä voi olla myös toinen tekninen toimittaja. Vaikka se pystyykin hahmottamaan projektin tilaa pelkästään lähdekoodista käsin, ohjelmiston nykytilan visualisointi korkealla tasolla saattaisi helpottaa projektin haltuunottovaihetta.

Koodin logiikan kommunikointiin on yritetty löytää erilaisia tapoja. Yksi niistä, Domain-driven design (termi, jonka Eric Evans toi esiin teoksessaan [18]), pyrkii tuomaan liiketoimintaa ja sen perusteella syntyvää ohjelmistokoodia lähemmäksi toisiaan. Yksinkertaistettuna tämä tapahtuu mallintamalla liiketoimintaa ja sen termistöä ohjelmistokoodissa hyödyntämällä tietynkaltaisia rakennuspalikoita (kuten *Entity*, *Aggregate*, *Domain Event* tai *Service*). Liiketoiminnan erilaiset osa-alueet myös ryhmitellään. Tämä korkeamman tason ryhmittely asettaa kontekstin kulloinkin käytävälle keskustelulle ja sen yhteydessä esiintyvälle termistölle; mitä tämä käsite tarkoittaa tässä kontekstissa.

Lisäksi tärkeä osa Domain-driven design -ideologiaa on yhteisen kielen (englanniksi ubiquitous language) muodostaminen [18, s. 306]. Ajatuksena on, että liiketoiminnan asiantuntijat voivat keskustella sujuvasti teknisen toteutustiimin kanssa ilman, että liiketoimintaan liittyvä logiikka muuttaa täysin muotoaan toteutusvaiheessa tai ohjelmistokehittäjät puhuvat pelkästään teknistä jargonia.

Visualisoimalla näiden periaatteiden mukaisesti rakennettua koodia voidaan periaatteessa päästä tilanteeseen, jossa koko tiimi (kaikki keskeiset sidosryhmät mukaan lukien) voi ymmärtää varsinaisen toteutuksen pohjalta muodostettua esitystä.

4 Työkalu lähdekoodin visualisointiin

4.1 Teknologiavalinnat

Työn sovelluskohteeksi valittiin Scala-ohjelmointikieli, jota suoritetaan Java Virtual Machine -alustan päällä. Scalan kehitys alkoi vuonna 2001, ja sen ensimmäinen versio julkaistiin vuonna 2003. Kielen isä on Martin Odersky, joka toimii professorina Lausannen teknisessä yliopistossa (EPFL) [19]. Nimi Scala tulee englannin kielen sanoista *scalable* and *language*, mutta lisäksi se tarkoittaa italiaksi portaikkoa. Kielen logo kuvastaakin EPFL:n ikonisia portaita (kuva 8).

Kuva 8: Lausannen yliopiston portaikko ja Scala-logo [11].

Scala on staattisesti tyypitetty kieli, mikä tarkoittaa, että muuttujien tyypitykset ovat tiedossa jo lähdekoodin käännösaikana. Kääntäjä huomauttaa esimerkiksi lähdekoodissa esiintyvistä tyyppivirheistä. Vastakohtana staattisesti tyypitetylle kielelle ovat dynaamisesti tyypitetyt kielet, joissa muuttujille tai arvoille ei eksplisiittisesti määritetä tyyppejä ja tyyppi voi muuttua suorituksen aikana. Näissä kielissä virheet nousevat tyypillisesti esiin vasta suorituksen aikana. Tyypitetyssä kielessä tyyppimääritykset toimivat ajantasaisena dokumentaationa, josta on hyötyä niin koodia lukevalle henkilölle kuin kääntäjällekin [20, s. 5].

Scala on suunniteltu niin, että se yhdistää kaksi toisistaan poikkeavaa ohjelmointiparadig-

maa: funktionaalisen ohjelmoinnin ja olio-ohjelmoinnin. Scalassa kaikki arvot rakentuvat olioista [21, s. 18].

Tärkeä kysymys työn kannalta oli löytää tapa muodostaa abstrakti syntaksipuu. Ohjelmoijan kirjoittamaa ja lukemaa koodia nimitetään konkreettiseksi syntaksiksi. Abstrakti syntaksi on yksinkertaisempi, tietokoneen ymmärtämä puista muodostuva kuvaus ohjelmasta, ja sitä kutsutaan nimellä abstrakti syntaksipuu (AST) [20, s. 53]. Syntaksipuun muodostamiseksi Scalan ekosysteemi tarjoaa vaihtoehtoja:

- Scala-kääntäjän lisäosa
- Scala.meta-metaohjelmointikirjasto
- Scalan reflektiokirjaston hyödyntäminen.

Scala.meta olisi voitu valita tähän projektiin, mutta toistaiseksi sillä ei voi saavuttaa haluttua lopputulosta. Tarvittavat ominaisuudet (kuten nimiavaruuksia sisältävien nimien päättely) ovat suunnitteilla vasta seuraaviin versioihin [22]. Scala.meta -metaohjelmointityökalun tarkoituksena on korvata scala.reflect-kirjasto, joka on ollut de facto -standardi Scalan metaohjelmoinnissa.

Prototyyppi päädyttiin rakentamaan Scala-kääntäjän lisäosaksi. Toteutus on suoraviivainen, mutta ei välttämättä käyttäjäystävällisin.

Prototyyppi vaatii, että kohdeprojekti käyttää SBT-nimistä avoimen lähdekoodin koontityökalua. Prototyyppi eli lisäosa otetaan käyttöön konfiguroimalla kohdeprojektin koontityökalun asetuksia *build.sbt*-tiedostoa muokkaamalla (koodiesimerkki 1). Lisäosan konfiguroinnissa määritetään projektin nimiavaruus, jotta voidaan poimia projektin kannalta olennaiset viittaukset talteen. Näin yksinkertaistetaan visualisaatiota, koska viittaukset Scalan sisäisiin tyyppimäärityksiin jäävät pois.

```
1 scalacOptions += "-Xplugin:/Visualizer-assembly-1.0.jar"
2 scalacOptions += "-P:visualizer:namespace:com.example"
```

Koodiesimerkki 1: Lisäosan konfigurointi.

Tavallisesti Scalan kääntäjä kääntää vain muuttuneet osat ja muutoksen vaikutuksen alaiset osat vähentääkseen kääntämiseen kuluvaa aikaa. Tätä prosessia kutsutaan englanniksi termillä *incremental recompilation*. Tällä hetkellä prototyyppi kuitenkin olettaa, että

projekti käännetään kokonaisuudessaan, joten aikaisempien käännösten tuotokset pitää ensiksi poistaa (koodiesimerkki 2).

```
1 sbt
2 clean
3 compile
```

Koodiesimerkki 2: Lisäosan suorittaminen kohdeprojektissa komentoriviltä.

Visualisointiin keskittyvä käyttöliittymätoteutus tehtiin JavaScript-ohjelmointikielellä, seuraavia kirjastoja hyödyntäen:

- React
- D3.js
- Babel
- Webpack.

JavaScript itsessään on viime vuosina edennyt niin teknologiana kuin suosionkin osalta. Se on vallannut alaa perinteisimmiltä kieliltä ja kehittynyt selaimen yksinkertaisesta skriptauskielestä kunnolliseksi ammattityökaluksi jopa palvelinpuolen toteutuksia varten. Voisi sanoa, että JavaScript on saavuttamassa sen aseman, johon Java-ohjelmointikieli alun perin pyrki; ympäristö olisi asennettu niin tietokoneeseen kuin pölynimuriin [23]. Nykyään lähes jokainen laite, jossa on selain, kykenee JavaScriptin ajamiseen. JavaScriptin suorittamisen vaatimus ei varsinaisesti ole selain, vaan JavaScript-suoritusympäristö, kuten Chrome V8 [24].

React on Facebookin kehittämä avoimen lähdekoodin JavaScript-kirjasto. Se määrittää, miten sovelluksen käyttöliittymä ja siihen liittyvä logiikka rakennetaan. Sen kantavana ajatuksena on komponenttipohjaisuus [25]. Tämä ohjaa ohjelmistokehittäjää kirjoittamaan uudelleenkäytettävää, kompositioon perustuvaa koodia.

D3.js on JavaScript-kirjasto, joka on tarkoitettu dokumenttien (kuten HTML-sivun) muotoiluun datan perusteella [26]. Se soveltuu hyvin erilaisten interaktiivisten graafien ja visualisaatioiden tuottamiseen. D3.js noudattaa deklaratiivista ohjelmointiparadigmaa. Deklaratiivisuus on imperatiivisen ohjelmointiparadigman vastakohta: sen sijaan, että komennetaan tietokoneelle vaihe vaiheelta, miten jokin asia tehdään, deklaratiivisessa paradigmassa ilmaistaan, mitä halutaan saavuttaa [27].

Babel, vaikka ei suoraan liitykään Reactiin, on yleinen osa sen ympärille rakentuvaa ekosysteemiä. Babel on JavaScript-kääntäjä. Se mahdollistaa uusien ominaisuuksien (kuten ECMAScript 2017:n tai Reactin JSX-syntaksin) hyödyntämisen lähdekoodissa, vaikka selaimet eivät tukisikaan kyseisiä ominaisuuksia. Babel kääntää kirjoitetun lähdekoodin selainten tai muiden suoritusympäristöjen yleisesti tukemaksi JavaScriptiksi. Tämä tarkoittaa sitä, että JavaScript voi kehittyä kielenä huomattavasti nopeammin kuin aikaisemmin, koska enää ei tarvitse odottaa uusien ominaisuuksien tukea erikseen jokaiselta selainvalmistajalta.

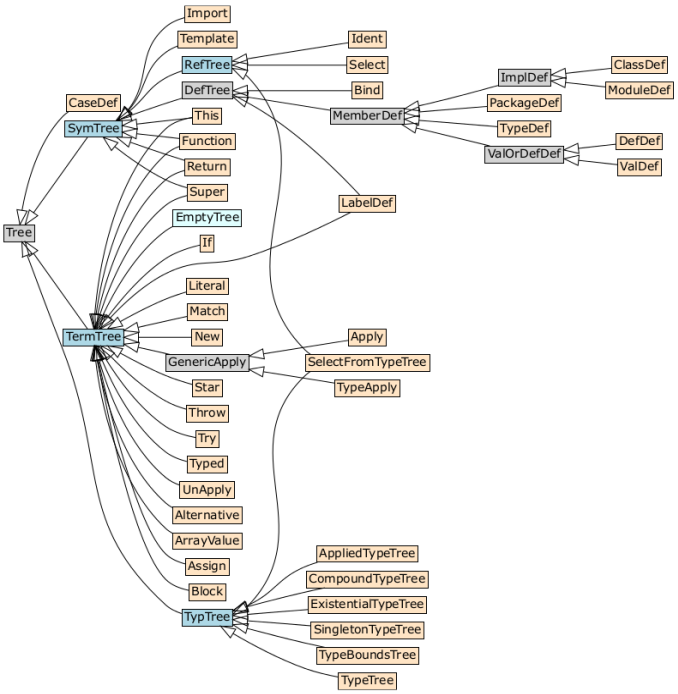
4.2 Scala-kääntäjän vaiheet

Scala-kääntäjä, *scalac*, on olennaisessa osassa insinöörityötä. Kääntäjä on ohjelma, joka muuntaa ohjelmointikielellä kirjoitetun ohjelmistokoodin (lähdekoodin) toiseksi tietokoneen ymmärtämäksi kieleksi.

Kääntäjä rakentaa lähdekoodista abstraktin syntaksipuun. Abstrakti syntaksipuu koostuu erilaisista elementeistä (kuva 9).

Scalac-ohjelma osaa kertoa kääntämisvaiheensa ja sen mitä niissä tapahtuu, kun annetaan komento *scala -Xshow-phases* (liite 1). Jokaisella kääntämisvaiheella luetaan edellisen vaiheen rakentama puu ja muodostetaan uusi, muokattu puu [28]. Työn kannalta oli tärkeää löytää vaihe, jossa on mahdollisimman paljon tyyppitietoa eikä mitään oleellista ole vielä purettu pois. Lähinnä kokeilujen kautta parhaaksi kääntäjävaiheeksi osoittautui referenssien tarkistusvaihe (*refchecks*), joka suorittaa vielä muutaman tyyppitarkistuksen [28].

Tutkimalla erilaisia elementtejä syntaksipuussa selvisi, että parhaiten työssä tarvittua tietoa tarjoavat *TypeTree*-tyyppiset elementit. Sitä kautta saadaan tarvittava määrä tietoa, joka sisältää riippuvuuden nimen nimiavaruuksineen. Kokonaisuudessaan nimi saattaa tosin sisältää Scalan sisäänrakennettujen tyyppien määrityksiä, kuten *Option* tai *List*, joten tämänkaltainen tieto suodatetaan, koska toistaiseksi tämän tiedon hyödyntämiselle visualisaatiossa ei ole käyttöä. Lisäksi mahdollista jatkokehitystä ajatellen visualisaatiota ei ole järkevää sitoa yksittäiseen ohjelmointikieleen.



Kuva 9: Scalan kääntäjän syntaksipuussa esiintyviä elementtejä [29, s. 95].

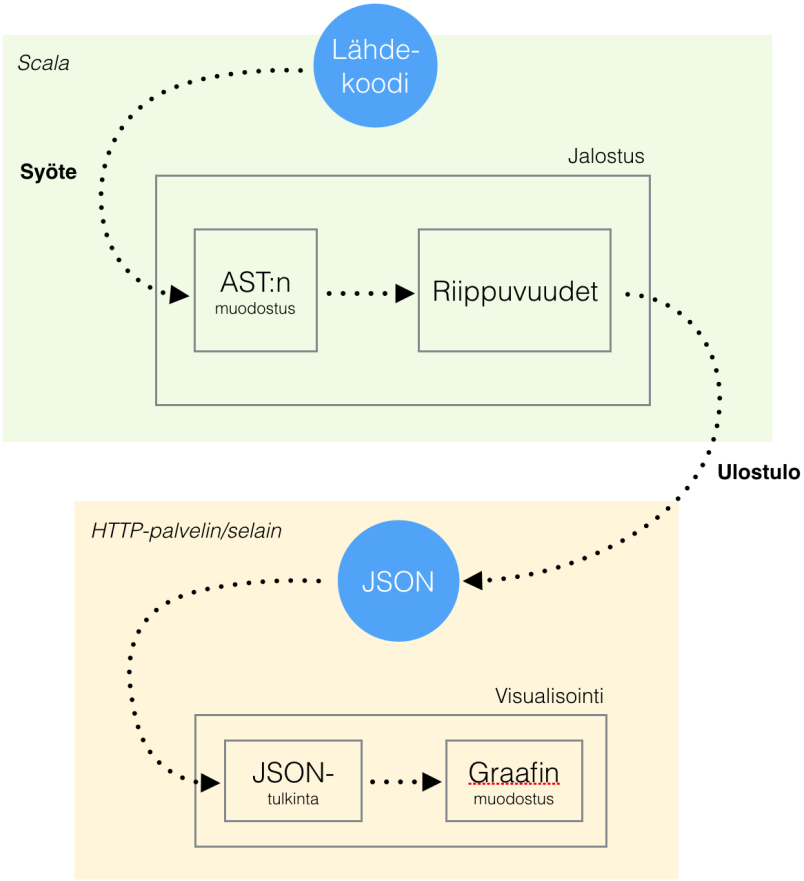
4.3 Prototyypin arkkitehtuuri

Insinöörityössä visualisointiohjelmiston toiminta koostuu muutamasta suoritusvaiheesta (kuva 10):

1. lähdekoodin lukeminen (kääntämisvaiheessa)
2. Abstract Syntax Tree -muodostus
3. riippuvuuksien tulkinta AST:sta
4. jalostaminen omaan JSON-formaattiin ja ulosvienti tiedostoon
5. JSON-tiedoston lataaminen ja visualisointi (graafimuodossa) selaimessa.

Lähdekoodin lukemisvaihe vaatii koko projektin kääntämisen. Toisin sanoen koko projektin lähdekoodi luetaan kerralla.

Kääntäjä muodostaa lähdekoodista abstraktin syntaksipuun. Syntaksipuuta tutkimalla voidaan päätellä eri osa-alueiden väliset riippuvuudet. Riippuvuudet, jotka eivät ole viittauksia projektin nimenomaisiin osa-alueisiin, suodatetaan pois. Tieto riippuvuuksista kerätään muistiin ja tallennetaan lopuksi JSON-tiedostoon.



Kuva 10: Prototyyppisovelluksen suorituksen eri vaiheet.

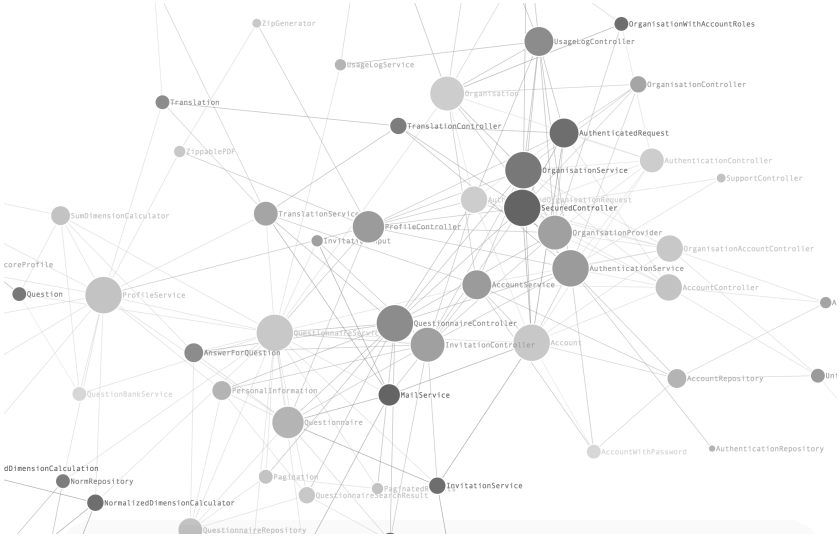
JSON-tiedosto kopioidaan visualisaatiokäyttöliittymän saataville ja tarjoillaan staattisena tiedostona HTTP-palvelimella. Näin selaimessa toimiva käyttöliittymä voi ladata tiedoston AJAX-pyynnöllä.

4.4 Käyttöliittymä

Prototyypin käyttöliittymän tärkein ominaisuus on yleinen visualisaatio lähdekoodista ja siinä esiintyvistä riippuvuuksista. Visualisaatiossa näkyvät pallot edustavat luokkia. Yksittäisen luokan esitysmuodon koko on sidottu riippuvuuksien määrään: mitä enemmän projektin eri osa-alueet ovat riippuvaisia kyseisestä luokasta, sitä suuremmalla ympyrällä sitä havainnollistetaan. Luokat yhdistetään toisiinsa kaareutuvilla viivoilla, joissa riippuvuuden suunnat ilmaistaan nuolilla. Visualisaatioalue on liikuteltava: sitä voi suurentaa tai pienentää, mikä on hyödyllinen ominaisuus, kun tarkastellaan laajaa projektia.

Ohjelmistoprojektien monimutkaisuus tekee visualisoinnista haasteellista, koska visuali-

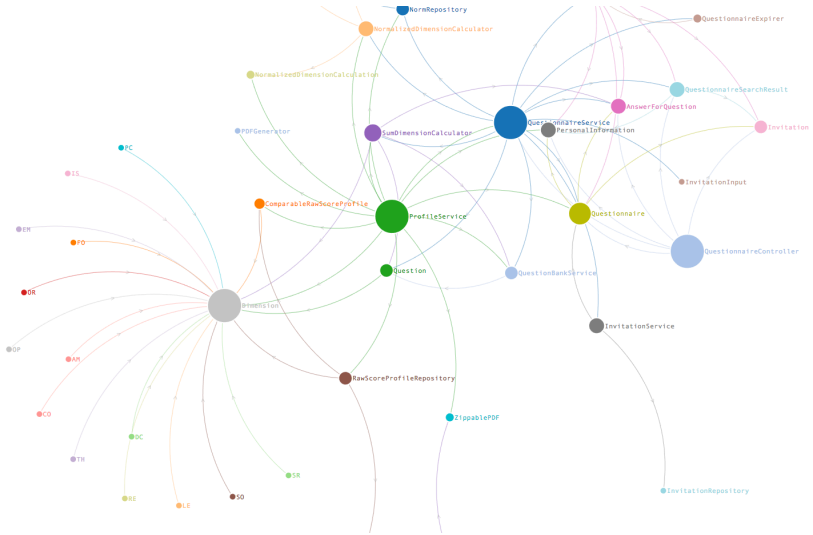
soitavia osia on lukumääräisesti paljon. Selkeästi tämä ilmenee esimerkiksi riippuvuuksien visualisoinnissa. Lopputulos on helposti sekava. Varsinkin riippuvuuksia ja niiden suuntaa on vaikea erottaa (kuva 11).



Kuva 11: Ensimmäinen visualisointi.

Visualisaatiota on yritetty selkeyttää seuraavin keinoin (kuva 12):

- Värikoodaus erottaa luokat toisistaan.
- Riippuvuus värikoodataan lähteen mukaan.
- Riippuvuuden suunta osoitetaan pienellä merkkinuolella.
- Riippuvuuksia osoittavat viivat kaareutuvat: vähemmän täysin samansuuntaisilta vaikuttavia, toisiinsa sekoittuvia viivoja.

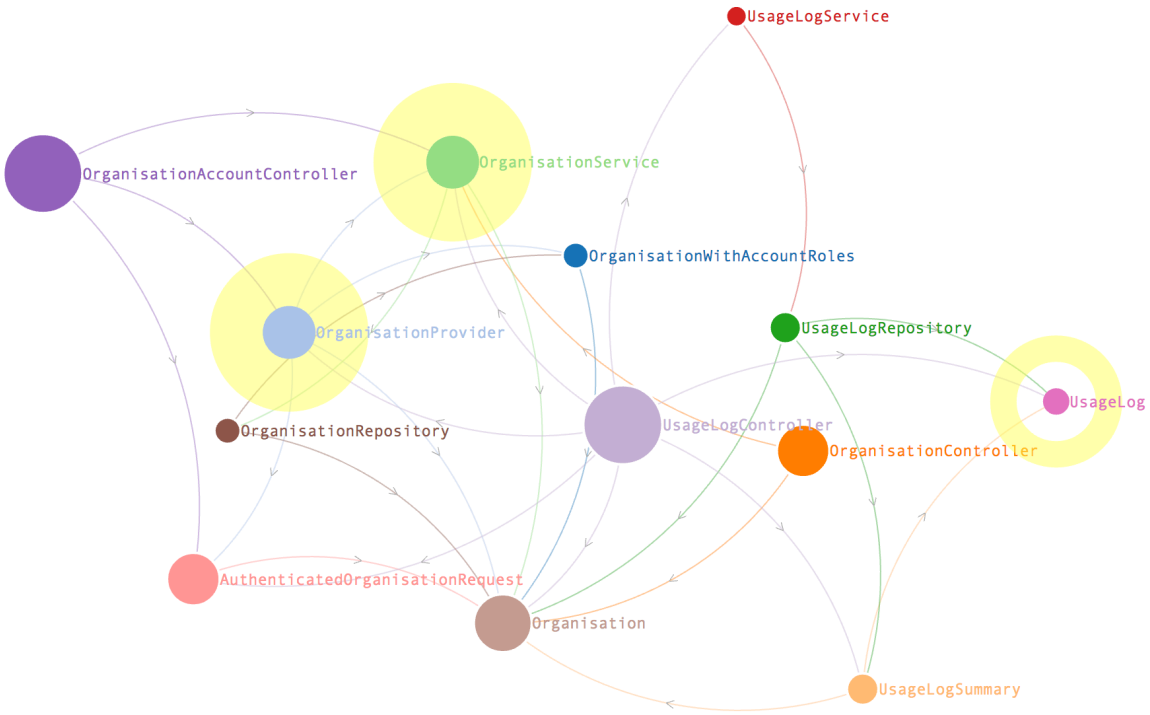


Kuva 12: Nykyinen toteutus.

Varsinaisen visualisoinnin lisäksi prototyyppi sisältää erilaisia toiminnallisuuksia. Toimintojen on tarkoitus helpottaa projektin visualisaation ja sen eri osa-alueiden tarkastelua.

Hiirellä osoitetun luokan nimiavaruuden sisältävä nimi ja riippuvuuksien lukumäärä tuodaan näkyville käyttöliittymän alareunaan (tieto näkyy kuvassa 16). Näin nähdään absoluuttinen riippuvuuksien määrä. Tieto nimestä kokonaisuudessaan helpottaa luokan etsimistä lähdekoodista.

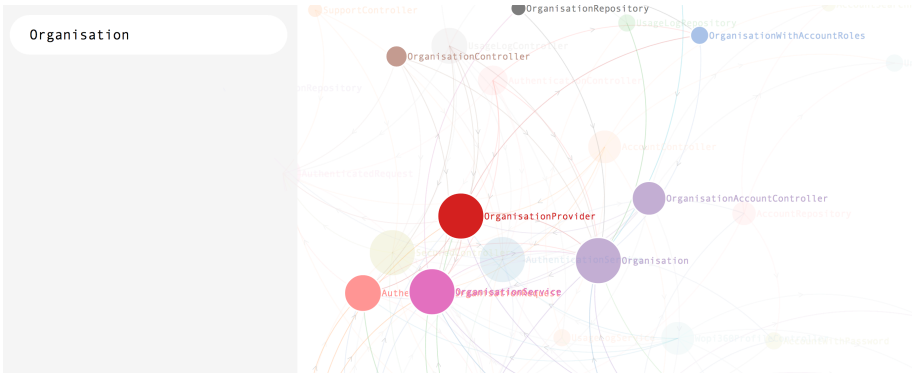
Luokkia voidaan väliaikaisesti korostaa kaksoisklikkaamalla. Tämä toimii keskusteluissa tai esittelyissä tukena, kun halutaan keskittyä visualisaation tiettyihin luokkiin (kuva 13).



Kuva 13: Korostettujen luokkien esitystapa.

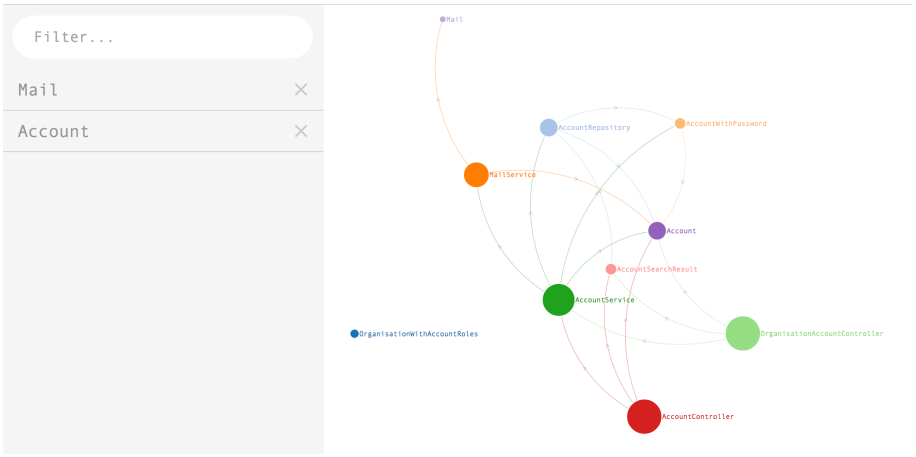
Visualisaatiosta voi myös hakea tai suodattaa luokkia nimellä (nimiavaruudet mukaan lukien). Käyttöliittymässä on kaksi erilaista hakutoiminnallisuutta: nopea suodatus ja hakusanalista.

Kirjoittamalla hakusanan hakukenttään hakusanasta ohimenevät luokat häivytetään taustalle; näin korostetaan täsmääviä luokkia (kuva 14). Suodatuksella nähdään nopeasti, kuinka isoon joukkoon haku kohdistuu.



Kuva 14: Hakusanan kirjoittamalla voi nopeasti nähdä, kuinka isoon joukkoon haku kohdistuu.

Enter-näppäimellä syötetty hakusana lisätään hakusanalistaan. Hakusanalista karsii kaikki muut paitsi hakuun täsmäävät luokat kokonaan pois näkyvistä. Hakusanoja voi syöttää useita, joten asiaankuuluvia kokonaisuuksia pystyy yhdistelemään ja näin muodostamaan mielenkiintoisia koonteja tarkasteltavaksi (kuva 15).



Kuva 15: Hakusanoja syöttämällä voidaan rakentaa ja tarkastella pienempiä kokonaisuuksia.

5 Tulokset, ideat ja nykyisen toteutuksen puutteet

Insinöörityössä tutkittiin visualisointiin painottuvia työkaluja. Erilaisia ratkaisuja on paljon. Moni niistä vaikuttaa olevan suunnattu lähinnä ohjelmistokehittäjille. Erilaiset toteutukset tuntuvat jäävän mielenkiintoisiksi kokeiluiksi sen sijaan, että niistä saisi arkipäivässä hyödyllisiä työvälineitä. Kysymykseksi jää, ratkaisevatko visualisoinnit varsinaisia kompleksisuuteen liittyviä ongelmia vai hoidetaanko niillä vain oireita.

Insinöörityössä toteutettiin visualisointisovelluksen prototyyppi. Prototyypin toteutus validoi insinöörityön hypoteesin: onko riippuvuuksien päättely ylipäätään mahdollista toteuttaa järkevästi.

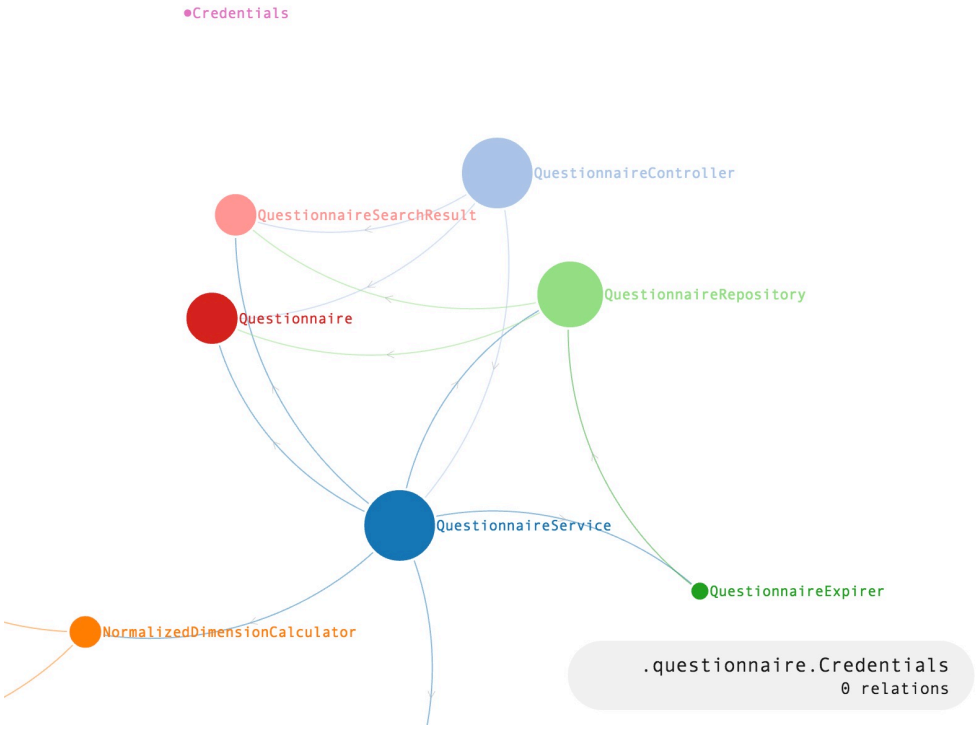
Prototyyppi koostuu kahdesta osa-alueesta. Ensimmäinen on Scala-kääntäjän lisäosa, jonka avulla voidaan päätellä projektissa esiintyvät luokat ja niiden väliset riippuvuudet sekä tallentaa tämä tieto JSON-tiedostoon. Toinen on selaimessa toimiva käyttöliittymä, joka esittää JSON-tiedostosta luetut riippuvuudet graafimuodossa.

Prototyyppi olisi paljon hyödyllisempi, jos se tukisi useita kieliä. Näin saataisiin prototyyppi helpommin ja laajemmin testikäyttöön. Samalla pystyttäisiin vertailemaan visualisaation soveltuvuutta täysin erilaisissa projekteissa.

Toteutuksen soveltuvuus voi vaihdella jo projektin koodaustyylin (esimerkiksi perinteistä olio-ohjelmointia tai funktionaalista ohjelmointia) perusteella. Ainakin jos ohjelmointityyli on olio- tai luokkapainotteinen, koodin osat saadaan visualisoitua selkeästi. Nykyinen toteutus ei tutki funktiokutsuja sen tarkemmin, joten jos ohjelmointityyli painottuu funktionaaliseen ohjelmointiin, todelliset riippuvuudet eivät välttämättä nouse esiin.

Käyttöliittymän toteutus oli mielenkiintoista, koska etukäteen ei voinut olla varma, miten erilaiset mekanismit toimisivat ja onko niistä hyötyä. Haku- ja suodatusominaisuudet osoittautuivat tärkeiksi ja luonnollisiksi osiksi hahmottamista. Lisäksi yksi huomionarvoinen hyöty on käyttämättömien osa-alueiden huomaaminen (kuva 16, jossa näkyy käyttämätön luokka *Credentials*). Kun huomataan turha koodi ja päästään siitä eroon, työ ei

myöhemmin keskeydy kysymyksellä koodin tarpeellisuudesta [30, s. 14].



Kuva 16: Käyttämätöntä koodia.

Työn tärkein tulos on kuitenkin erilaisten lähestymistapojen mahdollisuuksien vertailu orgaanisesti ajan myötä syntyneisiin ja vakiintuneisiin ohjelmoinnin perusmekanismien rajoituksiin. Nykyiset ohjelmointimenetelmät perustuvat askel askeleelta vanhan päälle rakennettuun, oli vanha pohja ollut hedelmällinen tai ei. Uudenlaisten ideoiden ja vanhojen tapojen kyseenalaistamisen tärkeys vain korostuu uusien paradigmojen, kuten tekoälyn ja lisätyn todellisuuden, yleistyessä.

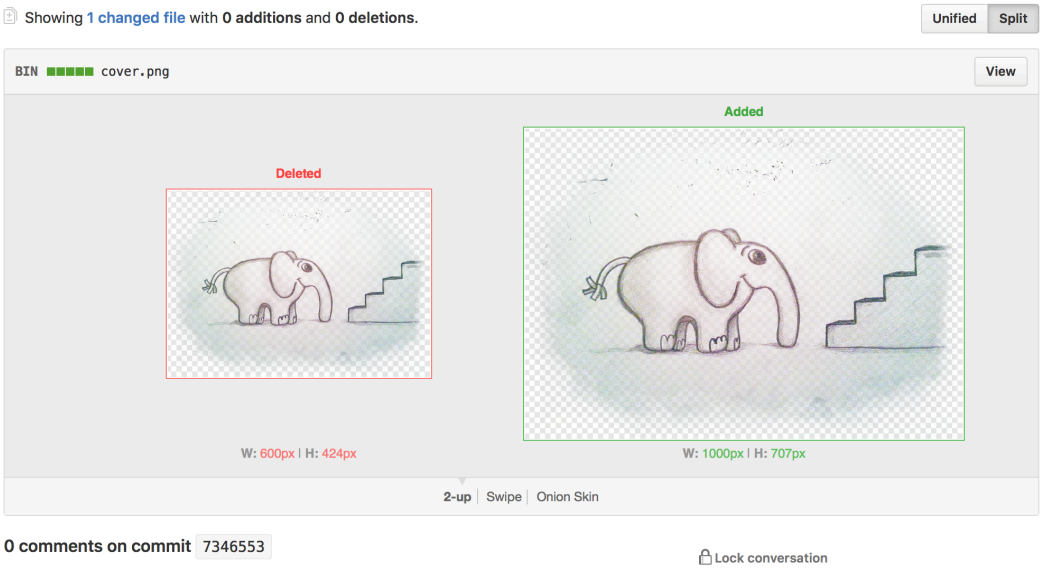
5.1 Ideat ja uudet sovelluskohteet

5.1.1 Visualisaation yhdistäminen versionhallintaan

Yhdistämällä visualisaatio versionhallintaan, vaikka osaksi GitHub-palvelua tai suoritettavaksi CI-palvelimella, pystyttäisiin esittämään kokonaisarkkitehtuurin muutokset esimerkiksi koodikatselmoinnin yhteydessä. Samalla saataisiin käsitys siitä, kuinka nopeasti so-

vellus kompleksisoituu ajan myötä. Myös refaktoroinnista toivottavasti seuraavat parannukset olisivat nähtävissä esimerkiksi riippuvuuksien vähentymisenä tai paremmin nimettyinä komponentteina.

Edellä kuvatun kaltainen ominaisuus olisi jo nyt mahdollista toteuttaa yksinkertaistetusti GitHub-palvelun kuvavertailuominaisuuden avulla (kuva 17), jos dynaamisen visualisoinnin lisäksi generoitaisiin staattinen kuva riippuvuuksista versionhallintaan.



Kuva 17: GitHub-palvelun kuvavertailuominaisuus.

Ainut rajoite nykyisessä toteutuksessa on generoinnin yhteydessä satunnaisesti määriteltävät värit ja sijainnit riippuvuuksille. Jotta vertailu olisi mielekkäämpää, pitäisi värejä ja sijainteja sitoa johonkin kiintopisteeseen. Näin uudet muutokset olisivat selkeämmin nähtävissä.

5.1.2 Ohjelmistokoodin tuottaminen visualisaation pohjalta

Ohjelmisto nykyversiossaan keskittyy jo rakennetun ohjelmiston visualisointiin. Seuraava askel voisi olla, että staattisen visualisoinnin sijaan ohjelman arkkitehtuuria voitaisiin rakentaa vastaavanlaisella työvälineellä. Tämä idea ei ole uusi, vaan jo 1960-luvulta, kuten T. O. Ellisin GRAIL-järjestelmän demo osoittaa [9]. Maailmalla on jo olemassa olevia toteutuksiakin, esimerkiksi NoFlo [8] ja Luna [17].

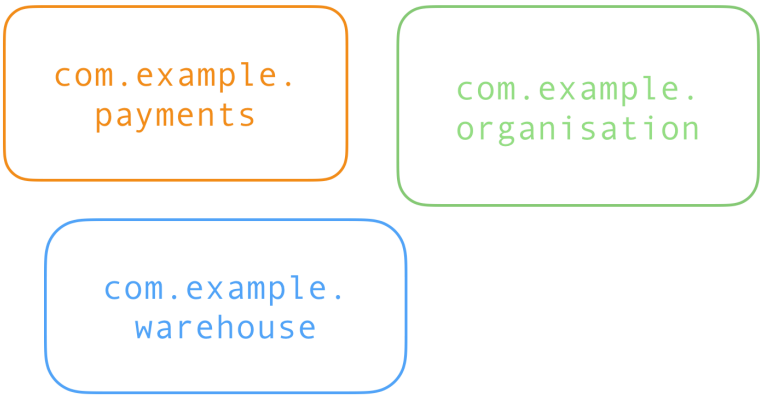
Vaikka teknologia kehittyykin nopeasti, ihmisten tavat ja tottumukset eivät [9]. Siksi kan-

nattaa suosia pientä kehitystä eteenpäin sen sijaan, että vaadittaisiin kokonaan uudenlaisen ohjelmointiparadigman käyttöä (kuten esimerkiksi NoFlon tapauksessa).

Lähdetään liikkeelle tilanteesta, jossa tuoteomistajalla, joka ei välttämättä osaa tuottaa ohjelmistokoodia, on liiketoimintaosaamisen lisäksi teknistä, jopa arkkitehtuurista ymmärrystä tuotteesta. Ensimmäinen askel on nykyisen koodipohjan visualisointi korkealla tasolla, jotta tuoteomistaja hahmottaa kokonaisuutta. Mutta tämän lisäksi tuoteomistaja voisi esimerkiksi lisätä uusien komponenttien rajapintamääritykset sovellukseen, jolloin ohjelmistokehittäjä voisi tehdä konkreettisen toteutuksen. Tuoteomistajalla olisi siis mahdollisuus yhdistellä kokonaiskuvaa valmiista tai vielä konkreettiselta toteutukseltaan olemattomista palasista.

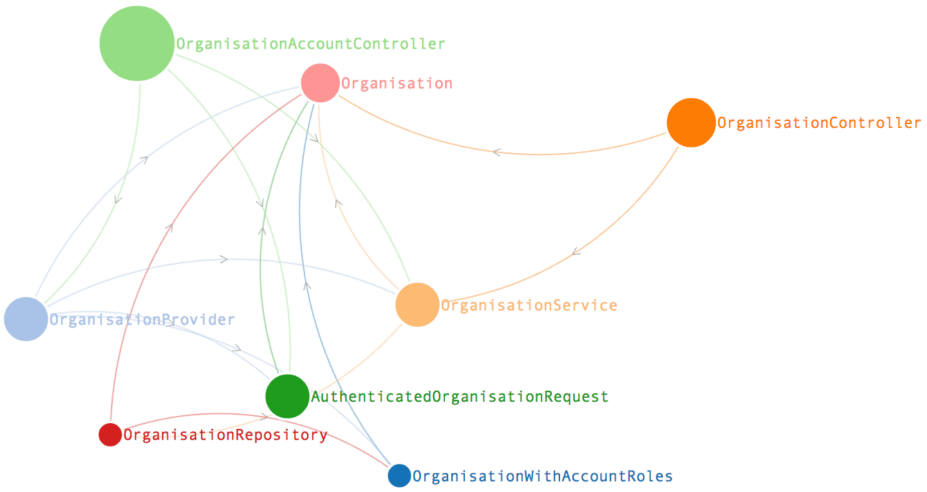
5.1.3 Laajemman kuvan tarkastelu ja tekniset yksityiskohdat

Lähdekoodin esitystarkkuuden toteutus prototyypissä on vielä pelkästään luokkatasolla (kuva 19). Tämä saattaa riittää varsinkin yksinkertaisiin ohjelmistoprojekteihin. Hyödyllistä kuitenkin olisi, jos esitystarkkuutta voisi säädellä käyttöliittymässä visuaalisaation suurennosta muuttamalla. Yksittäisten luokkien sijaan voisi tarkastella lähdekoodia nimiavaruuksien, pakettien tai moduulien tasolla. Nimiavaruudet saattavat sisältää liiketoiminnallisen kontekstin koodille, kuten laskutus tai rekisteröityminen (kuva 18).



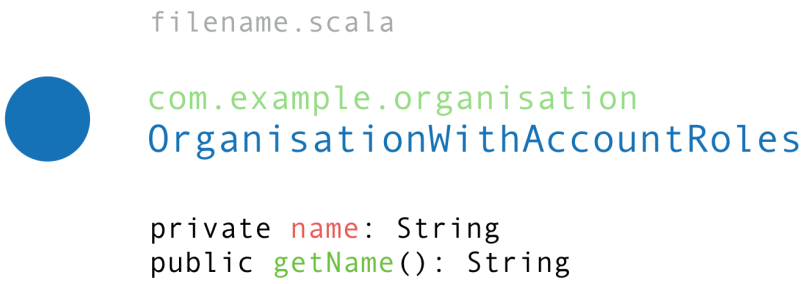
Kuva 18: Ylimmän tason visualisaatio, jossa näkyy vain lähdekoodin moduulit.

Voitaisiin siis korkealla tasolla valita laskutus ja syventyä tarkastelemaan siihen liittyviä luokkia ja niiden muodostamia rakenteita lähemmin (kuva 19).



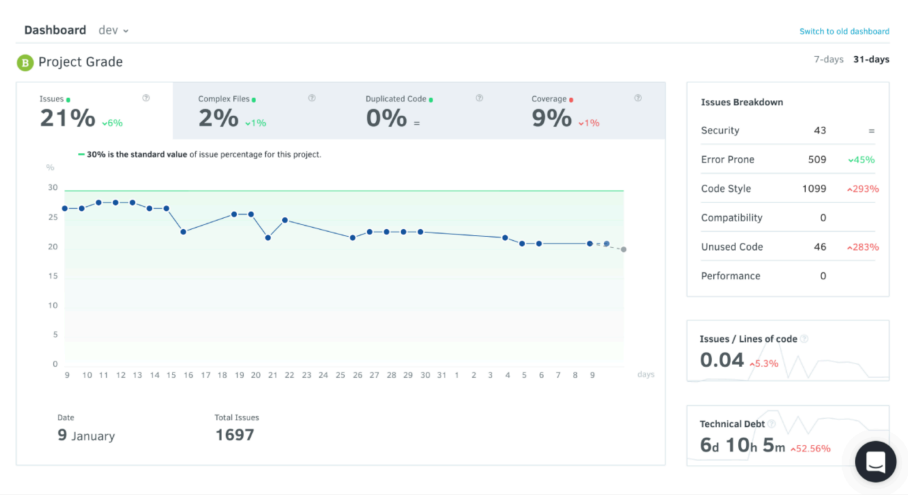
Kuva 19: Visualisaatio luokkatasolla.

Lisäksi luokkataso ei tällä hetkellä paljasta yksityiskohtia: luokan attribuutteja tai metodeja. Ne voisivat tulla näkyviin tarkastelemalla kohteita vielä lähempää (kuva 20). Viimeinen askel olisi navigointi konkreettiseen lähdekoodiin. Näin päästäisiin ohjelmistoprojektin lähdekoodin lineaarisesta esitysmuodosta lähemmäksi kokonaisvaltaisempaa spatiaalista hahmottamista.



Kuva 20: Visualisaatio tarkimmalla tasolla.

Laajempaa kuvaa voisi tarkastella myös tilastojen kautta. Tilastoja tai yksittäisiä lukuja ei kannata pitää ohjelmistoprojektin ainoina laatua määrittävinä tekijöinä. Tästä huolimatta ne voivat olla hyödyllisiä ja ohjata kehitystä parempaan suuntaan. Erilaisia ohjelmistoprojekteihin soveltuvia metriikoita ja tilastoja on kehitelty jo aikaisemmin. Siksi ei kannata keksiä pyörää uudelleen vaan integroitua johonkin olemassa olevaan toteutukseen tai palveluun, kuten https://www.codacy.com (kuva 21) tai https://codeclimate.com.



Kuva 21: Codacy-palvelu [31].

5.1.4 Lisätyn todellisuuden tai virtuaalitodellisuuden hyödyntäminen

Virtuaalitodellisuus (engl. Virtual Reality, lyh. vr tai VR) ja lisätty todellisuus (engl. Augmented Reality, lyh. ar tai AR) ovat yleistyneet akateemisista tutkimusprojekteista kaupallisesti varteenotettavaviksi ja tuotteistetuiksi ratkaisuiksi. Tästä teknologiasta voisi olla hyötyä tämänkin työn kontekstissa.

Työssä on käsitelty ohjelmistoprojektin visualisointia, mutta kaksiulotteisesti. Yksi mahdollisuus olisi tuoda visualisaatio kolmiulotteiseksi ja tilallisesti osaksi todellista maailmaa. Luokkia ja niiden sijaintia voitaisiin määrittää X- ja Y-akseleiden lisäksi Z-akselilla. Koodin tarkastelu ei olisi enää rajoitettu kaksiulotteiseen ikkunaan. Arkkitehtuurin suunnitteluun liittyvä keskustelu käytäisiin kirjaimellisesti koodin ympärillä.

Voidaan kuvitella leikkisä skenaario, jossa ohjelmistoprojektia varten olisi varattu oma huoneensa. Huoneeseen visualisoitaisiin projektin lähdekoodi kolmiulotteisesti (esimerkiksi lisätyn todellisuuden avulla). Komponentit kelluisivat esimerkiksi palloina ja riippuvuudet näiden välillä näkyisivät lankoina ilmassa. Projektin luonne olisi nähtävissä jo ovelta. Onko kyse pienestä vai laajasta järjestelmästä? Koostuuko järjestelmä paristä monoliittisesta lohkareesta vai lukuisista pienistä osista?

Lisäksi mielenkiintoinen kysymys. Voitaisiinko kyseisessä tilanteessa organisoida koodia kolmiulotteisesti fyysisessä maailmassa, puumaisten kansioiden ja tiedostojen sijaan? Ohjelmistoprojektin liiketoimintakriittisimmät osa-alueet voisivat esimerkiksi sijaita huo-

neen keskellä, kun taas harvoin huomiota vaativat koodikomponentit sijaitsisivat nurkissa. Siivouksen, eli refaktoroinnin, tarve saattaisi tulla selvemmin esille.

Vaikka edellä esitetyt ajatukset saattavat kuulostaa kaukaisilta, ei niiden toteutus nykyteknologialla ole mahdotonta. Lisätty todellisuus on lähitulevaisuudessa arkipäivää.

5.2 Nykyisen toteutuksen tiedostetut puutteet

Insinöörityössä Scala-projekteille tehty toteutus generoi riippuvuudet visualisointia varten Scalan kääntäjään nojaten. Riippuvuuksien päättely vaatii tällä hetkellä koko projektin lähdekoodin kääntämisen, eikä tavallisesti käytetty inkrementaalinen uudelleenkääntäminen, mikä on huomattavasti nopeampaa, ole mahdollista. Yksi vaihtoehto olisi parantaa toteutusta niin, että se osaisi ottaa huomioon vain muuttuneen lähdekoodin (kuten Scalan kääntäjä tekee) ja päivittää riippuvuudet tämän perusteella. Tämä muutos olisi kuitenkin vain pieni optimointi nykytoteutukseen. Vaihtoehtoinen ja huomattavasti käyttäjäystävällisempi ratkaisu olisi scala.meta-ohjelmiston hyödyntäminen. Scala.metan 2. versio mahdollistaisi ohjelmiston suorittamisen ja riippuvuuksien generoinnin [22] suoraan lähdekoodia lukemalla. Kohdeprojektia ei siis tarvitsisi kääntää kokonaisuudessaan, mikä nopeuttaisi prosessia huomattavasti. Lisäksi tämä mahdollistaisi sen, että riippuvuuksien generointi voisi olla täysin erillään kohdeprojektista: erillinen ohjelma, jolle projektin lähdekoodi vain annettaisiin syötteenä.

Visualisointitoteutuksen suorituskyky, varsinkin laajoissa projekteissa, joissa visualisoitavaa on paljon, voi huonontaa käyttökokemusta. Prototyyppiä rakennettaessa käyttöliittymän suorituskykykoptimointiin ei tarkoituksella kiinnitetty huomiota. Yksi optimoinneista, joka kuitenkin ulkoasun kustannuksella tehtiin, on läpinäkyvien elementtien vähentäminen. Lisäksi tällä hetkellä riippuvuuksien päättely vaatii projektin koko lähdekoodin kääntämistä alusta asti. Tämä voi olla hidasta laajoissa Scala-projekteissa.

Nykyinen toteutus ei ole käyttäjäystävällinen. Vaikka käyttöönotto onkin suhteellisen helppoa (toteutus SBT-työkalun lisäosana), itse käyttö voisi olla huomattavasti optimaalisempaa. Hidasteena on toistaiseksi se, että generoitava JSON-tiedosto (joka kuvaa riippuvuudet) pitää manuaalisesti kopioida samaan kansioon visualisointityökalun kanssa, jotta HTTP-palvelin pystyy siirtämään sen staattisena tiedostona selaimelle.

Ideaalitilanteessa tämä kaikki tapahtuisi automaattisesti. Visualisaatio voisi päivittyä muutosten yhteydessä automaattisesti verkkosivulle, jonne koko tiimillä on pääsy.

6 Yhteenveto

Insinöörityön tavoite oli tehdä karkea prototyyppiohjelmisto, jonka avulla todetaan, onko tällainen visualisaatio teknisesti mahdollista rakentaa. Lisäksi toisena tavoitteena oli mahdollisen subjektiivisen hyödyn toteaminen.

Tekninen toteutettavuus pystyttiin toteamaan hyvinkin nopeasti rakentamalla toimiva prototyyppi. Toteutettu prototyyppi, vaikka onkin nykytilassaan ominaisuuksiltaan karu, todistaa jo olevansa edes jollain tasolla hyödyllinen. Hyödyllisyyttä voi perustella subjektiivisella kokemuksella siitä, että visualisointia voi käyttää apuna ohjelmistoprojektin osa-alueiden ja riippuvuuksien hahmottamisessa. Prototyyppi suoriutui myös käytännön testistä. Asiakasprojektin lähdekoodin visualisaatiota käytettiin apuna uuden ohjelmistokehittäjän perehdyttämiseen. Lisäksi visualisaatiolla hahmotettiin uuden toiminnallisuuden toteutuksen vaikutusta sovelluksen arkkitehtuuriin.

Kokemuksen kautta visualisointia voi oppia hyödyntämään paremmin. Lisäksi rakentamalla oivalluksien kautta tulleita uusia ominaisuuksia prototyyppiä voi parantaa entisestään. Lopputulos ei kuitenkaan tuo varsinaista ratkaisua ohjelmistoprojektien kompleksisuuteen. Ohjelmistoprojektit ovat jatkossakin luonteeltaan monimutkaisia, eikä tätä voi kiertää. Mutta toivottavasti joskus päästään tilanteeseen, jossa erilaiset sidosryhmät pystyvät hahmottamaan ja ymmärtämään ohjelmistoissa esiintyvää kompleksisuutta ja sen seurauksia paremmin.

Insinöörityön tärkein tulos ei ole siitä syntynyt kevyt prototyyppi, vaan ajatukset ja keskustelut, jotka aihe on herättänyt. Toivottavasti tämä kipinä luo pohjaa uudenlaiselle ajattelulle sen suhteen, miten tulevaisuudessa hahmotetaan ja käsitellään ohjelmistojen lähdekoodia.

Lähteet

1 Wettel, Richard. 2016. CodeCity. Verkkodokumentti. <https://wettel.github.io/codecity.html>. Luettu 7.8.2016.

2 Caudwell, Andrew. 2009. Gource: software version control visualization. Verkkodokumentti. <https://github.com/acaudwell/Gource>. Luettu 3.3.2017.

3 Caudwell, Andrew. 2014. Gource - a software version control visualization tool. Verkkodokumentti. <http://gource.io>. Luettu 5.3.2017.

4 MySQL Workbench. 2017. Verkkodokumentti. Oracle. <https://www.mysql.com/products/workbench/>. Luettu 23.3.2017.

5 Bergius, Henri. 2013. Interview with J. Paul Morrison, the father of Flow-Based Programming. Verkkodokumentti. <http://bergie.iki.fi/blog/paul-morrison-interview/>. Luettu 7.3.2017.

6 Morrison, J. Paul. 2001. Flow-based Programming. Verkkodokumentti. <http://www.jpaulmorrison.com/fbp/>. Luettu 7.3.2017.

7 Morrison, J. Paul. 2014. Definition of Flow-based programming. Verkkodokumentti. <https://github.com/flowbased/flowbased.org/wiki/Definition>. Luettu 7.3.2017.

8 NoFlo, Flow-Based Programming for JavaScript. 2014. Verkkodokumentti. NoFlo. <https://noflojs.org>. Luettu 18.2.2017.

9 Victor, Bret. 2013. The Future of Programming (Dropbox's DBX conference). Verkkodokumentti. <https://vimeo.com/71278954>. Luettu 18.2.2017.

10 Gilakjani, Abbas Pourhossein. 2012. Analysis of Working Memory and Its Capacity Limit in Visual and Auditory Information. Verkkodokumentti. <http://www.academypublication.com/issues/past/tpls/vol02/02/26.pdf>. Luettu 20.11.2016.

11 Odersky, Martin; Spoon, Lex & Venners, Bill. 2011. Programming in Scala, Second Edition. Artima.

12 Kerievsky, Joshua. 2004. Refactoring to Patterns. Addison-Wesley.

13 ISTQB:n testaussanasto. 2014. Verkkodokumentti. ISTQB. <http://www.fistb.fi/sites/fistb/files/liitteet/istqb_sanasto_2015-04-30%202.3%20ENG-FI.pdf>. Luettu 6.2.2017.

14 Metriikat ja mittaaminen. 2011. Verkkodokumentti. Tampereen teknillinen yliopisto. <http://www.cs.tut.fi/~evo/kalvot/metriikat6.pdf>. Luettu 8.9.2016.

15 Cyclomatic complexity. 2016. Verkkodokumentti. Wikipedia. <https://en.wikipedia.org/wiki/Cyclomatic_complexity>. Luettu 20.11.2016.

16 Bird, Christian & Bachhelli, Alberto. 2013. Expectations, Outcomes, and Challenges Of Modern Code Review. Verkkodokumentti. <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ICSE202013-codereview.pdf>. Luettu 10.3.2017.

17 Luna, Visual and textual functional programming language. 2017. Verkkodokumentti. New Byte Order. <http://www.luna-lang.org>. Luettu 18.2.2017.

18 Evans, Eric. 2003. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

19 Odersky, Martin. 2016. A Brief History of Scala. Verkkodokumentti. <http://www.artima.com/weblogs/viewpost.jsp?thread=163733>. Luettu 4.3.2017.

20 Pierce, Benjamin C. 2002. Types and Programming Languages. The MIT Press.

21 Wampler, Dean & Payne, Alex. 2014. Programming Scala. O'Reilly Media.

22 Burmako, Eugene. 2016. Scala.meta- ja scala.macros -projektien perustaja, Lausannen tekninen yliopisto, Lausanne. Keskustelu 3.9.2016.

23 Java Timeline. 2015. Verkkodokumentti. Oracle. <http://oracle.com.edgesuite.net/timeline/java/>. Luettu 15.3.2017.

24 Chrome V8. 2012. Verkkodokumentti. Google. <https://developers.google.com/v8/>. Luettu 25.2.2017.

25 Forsström, Mikko. 2015. Paras ystäväni React. Verkkodokumentti. <https://fraktio.fi/blogi/paras-ystavani-react>. Luettu 26.9.2016.

26 Bostock, Mike. 2015. Data-Driven Documents. Verkkodokumentti. <https://d3js.org>. Luettu 25.2.2017.

27 Turner, Ada. 2016. Swift 3 and Declarative Programming. Verkkodokumentti. <https://possiblemobile.com/2016/09/swift-3-declarative-programming>. Luettu 26.2.2017.

28 Overview of Compiler Phases. 2011. Verkkodokumentti. Scala Group. <https://wiki.scala-lang.org/display/SIW/Overview+of+Compiler+Phases>. Luettu 11.3.2017.

29 Stocker, Mirko. 2012. Scala Refactoring. Master's Thesis. University of Applied Sciences Rapperswil.

30 Cheung, Ka Wai. 2012. The Developer's Code. The Pragmatic Programmers.

31 Automated code reviews & code analytics. 2016. Verkkodokumentti. Codacy. <https://www.codacy.com>. Luettu 23.3.2017.

Scalac-ohjelman komentorivitulosteen mukaiset kääntövaiheet

phase name	description
parser	parse source into ASTs, perform simple desugaring
namer	resolve names, attach symbols to named trees
packageobjects	load package objects
typer	the meat and potatoes: type the trees
patmat	translate match expressions
superaccessors	add super accessors in traits and nested classes
extmethods	add extension methods for inline classes
pickler	serialize symbol tables
refchecks	reference/override checking, translate nested objects
uncurry	uncurry, translate function values to anonymous classes
tailcalls	replace tail calls by jumps
specialize	@specialized-driven class and method specialization
explicitouter	this refs to outer pointers
erasure	erase types, add interfaces for traits
posterasure	clean up erased inline classes
lazyvals	allocate bitmaps, translate lazy vals into lazified defs
lambdalift	move nested functions to top level
constructors	move field definitions into constructors
flatten	eliminate inner classes
mixin	mixin composition
cleanup	platform-specific cleanups, generate reflective calls
delambdafy	remove lambdas
icode	generate portable intermediate code
jvm	generate JVM bytecode
terminal	the last phase during a compilation run