Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Noora Turunen

Korjausvelkalaskentaohjelmiston jatkokehitys ja refaktorointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Ohjelmistotuotanto

Insinöörityö

1.3.2021

Tekijä Otsikko Sivumäärä Aika	Noora Turunen Korjausvelkalaskentaohjelmiston jatkokehitys ja refaktorointi 43 sivua + 4 liitettä 1.3.2021
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Simo Silander Vanhempi asiantuntija Juuso Ketonen

Tämän insinöörityön aiheena oli Rapal Oy:n Fore-tuotteen korjausvelkalaskentaohjelmiston, Koven, jatkokehitys ja refaktorointi. Tavoitteena oli suunnitella ja toteuttaa käyttäjien toivomat lisäominaisuudet Koveen ja samalla vähentää ohjelmistoon syntynyttä korjausvelkaa ja parantaa sen luettavuutta ja laajennettavuutta refaktoroinnin avulla. Tarkoitus oli tutustua refaktorointiin, sen tekniikoihin ja prosessiin sekä tehdä refaktoroinnista mahdollisimman pysyvä osa kehitystiimin toimintaa. Tämän tueksi ja Koven laadun varmistamiseksi toteutukseen haluttiin projektin aikana lisätä automaattinen koodinanalysointityökalu SonarQube.

Työ toteutettiin tutustumalla Martin Fowlerin esittelemiin refaktoroinnin tekniikoihin ja prosessiin. Jatkokehitettävät kohteet valittiin ja lisäominaisuudet suunniteltiin tiimin kesken. Uudet ominaisuudet lisättiin noudattamalla refaktoroinnin prosessia, jotta samalla voitaisiin parantaa vanhan koodin rakennetta. Koveen liitettiin SonarQube, joka analysoi koko koodin sekä uudet koodimuutokset erikseen. SonarQuben analyysia hyödynnettiin työn viimeisissä refaktoroinneissa.

Työn tavoitteet saavutettiin. Kaikki suunnitellut ominaisuudet toteutettiin, ja nyt käyttäjät voivat muun muassa priorisoida rakennuksia sekä tuoda omia priorisointikategorioita laskelmiin. Toteutuksessa käytettiin hyödyksi refaktorointia ja Koven tuonnin, viennin ja Price-Servicien koodipohjaa saatiin huomattavasti parannettua. Koodi on nyt paljon aikaisempaa luettavampaa ja laajennettavampaa.

Työn tulokset kannustivat koko kehitystiimiä jatkamaan refaktorointia muun kehityksen yhteydessä. Säännöllinen refaktorointi helpottaa koodin ylläpitoa, ja työn tulosten sekä SonarQuben ansiosta tästä tullaan varmasti pitämään kiinni. Jatkossa on tarkoitus lisätä Koven käyttöliittymän koodi SonarQuben analyysiin ja refaktoroida frontend-koodia.

Avainsanat	refaktorointi, jatkokehitys, tekninen velka, SonarQube

Author Title Number of Pages Date	Noora Turunen Further Development and Refactoring of a Maintenance Backlog Calculation Software 43 pages + 4 appendices 1 March 2021
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Simo Silander, Senior Lecturer Juuso Ketonen, Senior Developer

The purpose of this bachelor's thesis was to further develop and refactor a maintenance backlog calculation software, Kove, which is one of Rapal's Fore product applications. The objective was to design and develop new features the users had wished for and at the same time reduce the repair debt incurred on the software and improve its readability and extensibility. The purpose was to, after having got acquainted with refactoring, its techniques and process, to make refactoring as permanent a part of the development team's activities as possible. To keep this up and to ensure the quality of Kove in the future, the aim was to add an automatic code analysis tool SonarQube to the implementation during the project.

The study was carried out by getting acquainted with the refactoring techniques and process presented by Martin Fowler. The items to be further developed were selected and additional features were designed. The new features were created by following the refactoring process to improve the structure of the old code. SonarQube was installed to the Kove solution. It analyzed the entire code as well as every new code push. SonarQube's analysis was utilized in the final refactorings of the study.

The goals of the study were achieved. All the planned features were added and now the users can e.g., prioritize buildings and import their own prioritization categories to the calculations. Refactoring was used during the development and the codebases of importing, exporting and PriceServices were remarkably improved resulting in considerably more readable and extendable code.

The results of the study encouraged the development team at Rapal to continue refactoring during development. Regular refactoring makes it easier to maintain code and thanks to the results of the study and to SonarQube, this will be continued. In the future, it is planned to add Kove's user interface code to SonarQube's analysis and refactor the frontend code.

Keywords	refactoring, further development, technical debt, SonarQube

Sisällys

Lyhenteet

API	Application programming interface. Ohjelmointirajapinta. Määrittely, jonka mukaan ohjelmat voivat keskustella keskenään.
DIP	Dependency inversion principle. Olio-ohjelmoinnin periaate, jonka mukaan luokkien ei tule olla riippuvaisia käyttämiensä luokkien toteutuksista vaan niiden abstraktioista.
DTO	Data transfer object. Tiedonvälitysolio. Olio tiedon siirtoon prosessien välillä.
EF	Entity Framework. Avoimen lähdekoodin ORM-kehys ADO.NET:n käyttöön.
HTTP	Hypertext Transfer Protocol. Hypertekstin siirtoprotokolla. Selaimien ja WWW-palvelinten käyttämä tiedonsiirron protokolla.
ISP	Interface Segregation principle. Olio-ohjelmoinnin periaate, jonka mukaan asiakasohjelmien ei pidä olla riippuvaisia rajapinnoista, joita ne eivät käytä.
OCP	Open-closed principle. Olio-ohjelmoinnin periaate, jonka mukaan olioiden tulee olla avoimia laajennuksille, mutta suljettuja muutoksille.
ORM	Object-relational mapping. Oliomallin mukaisen esityksen kuvaus relaatiomallin mukaiseksi esitykseksi.
Kove	Korjausvelkalaskennan työkalu. Rapal Oy:n Fore-tuotteeseen hankittava sovellus, jonka avulla seurataan rakennetun omaisuuden kuntoa.
LSP	Liskov Substitution Principle. Olio-ohjelmoinnin periaate, jonka mukaan yläluokan olioiden tulee olla korvattavissa sen aliluokan olioilla.
REST	Representational State Transfer. Ohjelmointirajapintojen toteutukseen tarkoitettu HTTP-protokollaan perustuva arkkitehtuurimalli.

SOLID	Akronyymi, joka kuvaa olioperusteisen ohjelmoinnin viittä pääperiaatetta (SRP, OCP, LSP, ISP, DIP).
SRP	Single Responsibility Principle. Olio-ohjelmoinnin periaate, jonka mukaan luokilla tulee olla vain yksi tehtävä.

1 Johdanto

Monessa suhteessa rakennusten ja ohjelmistojen ylläpito on hyvin samanlaista. Jos perusta on huono, on uuden rakentaminen päälle paljon vaativampaa. Ongelmien ilmetessä tehdään usein vain välttämättömiä ja akuutteja korjauksia, mutta unohdetaan katsoa laajempaa kuvaa. Ajan kuluessa pienet rakenteelliset ongelmat alkavat haurastuttaa koko tekelettä ja lopulta edessä on paljon kalliimpi remontti kuin mitä pienillä jatkuvilla korjaus- ja ylläpitotoimenpiteillä olisi tarvittu. Kun tällaisia huonoja ratkaisuja tehdään, syntyy korjausvelkaa.

Rakennusalalla korjausvelan määrällä tarkoitetaan sitä summaa, joka rakennuksiin tai infrakohteisiin olisi pitänyt investoida, jotta ne olisivat käyttökelpoisessa kunnossa. Korjausvelkaa syntyy, kun ennakoivista kunnossapitotoimista tingitään ja vain välttämättömät korjaukset toteutetaan. [1.]

Vastaavalla tavalla myös ohjelmistoihin syntyy korjausvelkaa tai niin kutsuttua teknistä velkaa. Sitä syntyy usein kiireessä julkaisupaineen alla. Toteutuksissa tehdään niin kutsuttuja purkkaratkaisuja, jotka toimivat kyseiseen tarpeeseen, mutta tekevät ohjelmistosta monimutkaisemman, mikä hidastaa ylläpitoa ja jatkokehitystä. Tämä johtaa kehitysbudjetin kulumiseen ylläpitotoimiin. [2.]

Tämän insinöörityön aiheena on Rapal Oy:n Fore-tuotteen Korjausvelkalaskentaohjelmiston, Koven, jatkokehitys. Kove on erityisesti kuntien käyttöön suunnattu työkalu infraomaisuuden kunnon ja kehityksen seurantaan. Kove sisältää ominaisuudet vesihuoltoverkoston, katujen ja rakennusten korjausvelkaseurantaan sekä kohteiden priorisoinnin infraomaisuudelle. Työn tavoitteena on toteuttaa toimeksiantajan määrittelemät lisäominaisuudet sovellukseen. Kohteiden priorisointiin halutaan lisätä rakennusten priorisointi ja priorisointiin halutaan mahdollisuus tuoda omia priorisointityyppejä Rapalin määrittelemien lisäksi. Samalla vientiä halutaan päivittää selkeämmäksi.

Kovea aiemmin kehitettäessä koodin luettavuudessa ja laajennettavuudessa on huomattu ongelmia. Koven toteutukseen on kertynyt jo runsaammin teknistä velkaa ja siksi

muun jatkokehityksen ohella halutaan parantaa aiempaa koodia. Toinen tavoite projektissa onkin ohjelmakoodin laajennettavuuden ja ylläpidettävyyden parannus koodia refaktoroimalla, eli koodin rakennetta muokkaamalla muuttamatta toiminnallisuutta. Refaktoroinnin ja velan hallitsemisen varmistamiseksi myös jatkossa Koveen on työn aikana tarkoitus sisällyttää automaattinen koodintarkistustyökalu SonarQube.

Aluksi työssä kerrotaan refaktoroinnista. Refaktorointi määritellään tarkemmin ja sen prosessi esitellään. Koven jatkokehityksen aikana on tarkoitus hyödyntää tätä prosessia, siksi sen avaaminen on tärkeää. Luvun tarkoitus on myös esitellä refaktoroinnin hyötyjä ja siihen liittyviä haasteita sekä selventää, miten refaktorointia vaativat kohteet voidaan tunnistaa ohjelmakoodissa. Refaktoroinnista esitellään myös yleisimpiä tekniikoita, erityisesti niitä, joita työn aikana tullaan hyödyntämään.

Refaktoroinnin jälkeen esitellään kehitettävä ohjelma Kove. Kovesta esitellään sen käyttökohteet ja toiminta sekä sen tekninen toteutus. Sovelluksen esittelyn jälkeen käydään läpi työn alkuvalmistelut ja myöhemmin toteutus. Tässä vaiheessa esitellään työn suunnittelu ja työssä käytetyt työkalut ja ohjelmistot. Alkuvalmisteluista kertovassa luvussa esitellään Koven kehitettävät kohteet, eli projektin aikana sovellukseen lisättävät ominaisuudet ja korjaukset. Kehityskohteisiin liittyvät refaktoroinnin tarpeet esitellään myös tässä vaiheessa. Valmisteluista siirrytään työn toteutuksen esittelyyn, jossa käydään läpi, kuinka uuskehityskohteet ja refaktorointi käytännössä toteutettiin. Lopuksi esitellään työn tulokset ja pohdintaa tavoitteiden onnistumisesta.

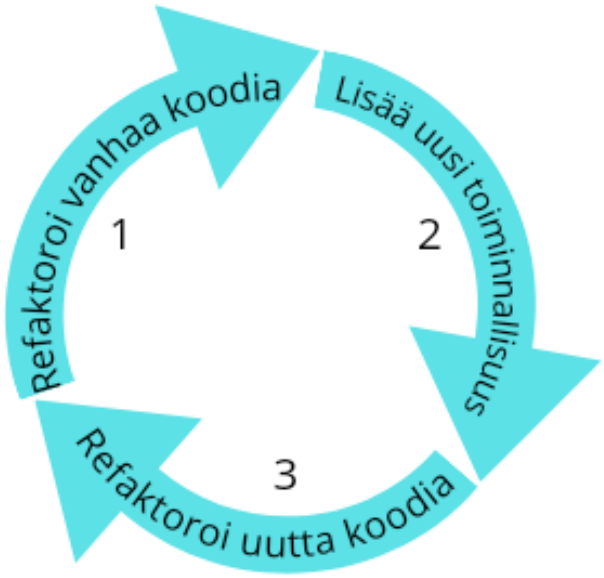
Tämän insinöörityön toimeksiantaja on Rapal Oy. Rapal on vuonna 1991 perustettu ohjelmistotalo, jolla on tuotteet infran kustannuslaskentaan sekä kiinteistö- ja työympäristöjohtamisen hallintaan. Rapal on henkilöstön omistama osakeyhtiö, joka työllistää noin 80 henkeä. Rapalin asiakkaita ovat muun muassa osa kunnista. [3.]

2 Refaktorointi

Teknisen velan hallintaan on olemassa useita keinoja. Kauppalehden kirjoituksessaan [2] Tero Niemistö nimeää yhdeksi keinoksi automaattiset koodianalysointityökalut. Hänen mukaansa niiden avulla teknistä velkaa voidaan pitää minimissä. Automaattinen analysointi ei kuitenkaan yksinään ole vastaus, vaan sen rinnalle on otettava korjausprosessi, joka hyödyntää analysoinnin tuloksia. Oikein toteutettuna refaktorointi on erinomainen esimerkki tällaisesta systemaattisesta korjausprosessista.

Refaktoroinnin kantaisä Martin Fowler määrittelee kirjassaan, Refactoring – Improwing the Design of Existing Code, refaktoroinnin olevan prosessi, jossa koodin toimintaa ei muuteta, mutta sisäistä rakennetta parannetaan. Hänen mukaansa refaktorointi on kurinalainen tapa siivota koodia välttäen samalla uusien virheiden syntyminen. Refaktoroinnin prosessissa toteutetaan useita yksittäisiä muutoksia, eli refaktorointeja. [4.] Näitä refaktoroinnin tekniikoita esitellään myöhemmin tässä luvussa.

Fowlerin [4] mukaan refaktoroinnin tulisi tapahtua osana jatkuvaa kehitysprosessia. Hän jakaa kehitysprosessin refaktorointiin ja uusien ominaisuuksien lisäämiseen, kuten Kovessakin on tarkoitus tehdä. Tämä tarkoittaa, että refaktoroidessa ei lisätä uusia toimintoja, vaan muokataan aiempaa koodia ymmärrettävämmäksi tai helpommin muokattavaksi. Fowlerin mukaan refaktoroinnille ei tarvitse kuitenkaan varata erillistä aikaa, vaan se toteutetaan muiden muutosten ohella esimerkiksi kuvan 1 esittämän prosessin mukaisesti.



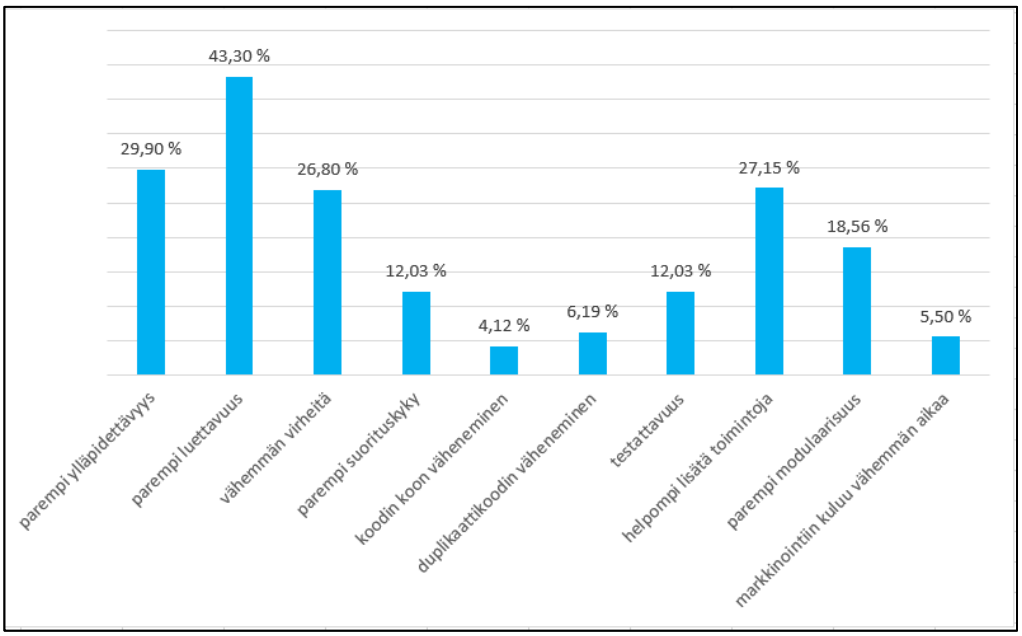
Kuva 1: Refaktoroinnin prosessi

Koven kehityksen aikana pyritään noudattamaan kuvan 1 mukaista prosessia. Ennen uuden toiminnon lisäystä, tai virheen korjausta, on tavoitteena tutkia muokattavaa koodia kriittisesti. Koodin alkuperäistä rakennetta refaktoroidaan ennen uuden ominaisuuden lisäämistä, mikäli refaktorointi helpottaa koodin luettavuutta, tai helpottaa muulla tavoin tulevaa muokkausta. Tämän jälkeen tehdään uusi ominaisuus tai korjaus. Lopuksi uutta koodia tarkastellaan taas refaktoroinnin näkökulmasta ja tehdään rakennetta parantavat muutokset.

Refaktoroinnin hyödyt

Refaktorointia siis suositellaan jatkuvana osana kehitystä. Mutta miksi? Mitä hyötyjä refaktorointi tarjoaa kehittäjille tai yritykselle? Refaktoroinnin perimmäinen tarkoitus on parantaa koodin laatua ja sitä kautta luettavuutta ja laajennettavuutta. Kirjassaan [4] Fowler listaa useita refaktoroinnin hyötyjä. Hänen mukaansa refaktoroinnin avulla voidaan parantaa koodin muotoilua sekä tehdä koodista selkeämpää ja luettavampaa, mikä taas nopeuttaa ohjelmistojen jatkokehitystä. Refaktorointi voi hänen mielestään myös helpottaa virheiden löytymistä koodista.

Refaktorointi on monille ohjelmistokehittäjille tuttu käsite ja päivittäinen työkalu. Sen hyödyt tunnistetaan kooditasolla hyvin. Esimerkiksi Microsoftilla vuonna 2012 tehdyn kyselytutkimuksen [5] mukaan ohjelmistokehittäjät kokivat refaktoroinnin tuovan heille useita hyötyjä. Kuvassa 2 esitellään prosentteina, kuinka moni työntekijöistä koki refaktoroinnin tuovan erilaisia hyötyjä.



Kuva 2: Ohjelmoijien kokemat refaktoroinnin hyödyt Microsoftilla [5]

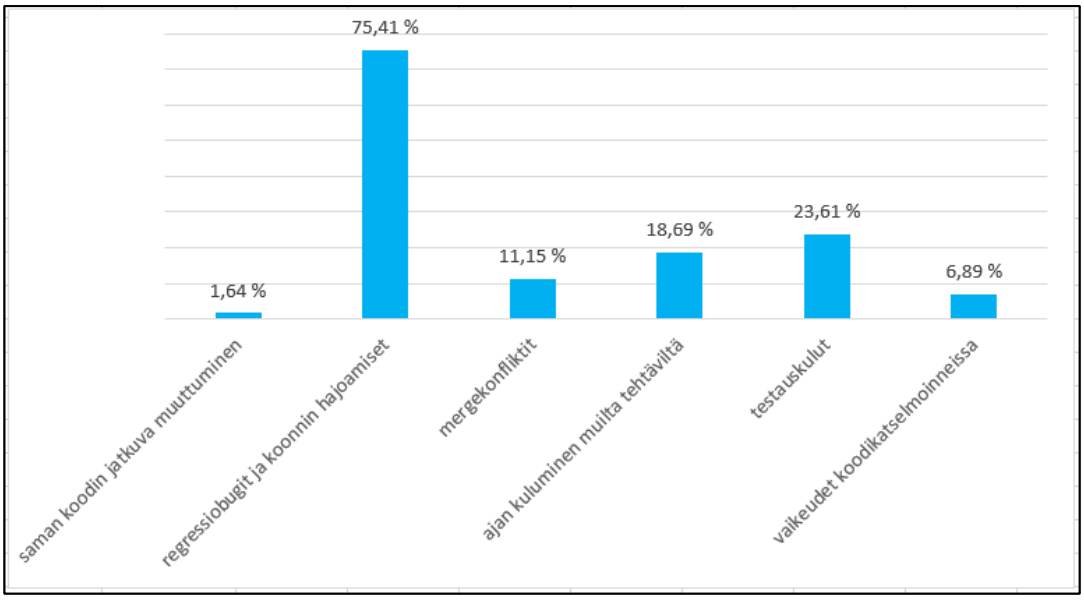
Kuten kuvassa 2 esitellään, Microsoftilla tehdyn tutkimuksen mukaan yli 40 prosenttia kehittäjistä koki refaktoroinnin parantavan koodin luettavuutta ja noin 30 prosenttia koki refaktoroinnin helpottavan koodin ylläpidettävyyttä ja uusien ominaisuuksien lisäämistä sekä vähentävän koodissa esiintyvien virheiden määrää. Tutkimuksessa kehittäjät olivat raportoineet myös monia muita hyötyjä, joita näkyy myös kuvan 2 kuvaajassa. Näitä olivat muun muassa ohjelmiston modulaarisuuden, suorituskyvyn ja testattavuuden parantuminen.

Refaktoroinnin hyödyt kehittäjille ovat ilmeiset, mutta entä yrityksen näkökulma? Miksi refaktorointiin kannattaisi kannustaa ylemmältäkin tasolta? Jos kehittäjä voi refaktoroinnin avulla nopeuttaa kehitystä ja löytää virheitä nopeammin, on se luonnollisesti myös yrityksen etu. Erilaisten tutkimusten avulla refaktoroinnin on todettu myös vähentävän

teknistä velkaa ja ylläpidosta aiheutuvia kustannuksia ohjelmistotaloissa. Rob Leitchin ja Eleni Stroulian vuonna 2003 tekemässä kustannusarviotutkimuksessa [6], Undestanding the Economics of Refactoring, tutkittiin refaktoroinnin kustannusten laskentaa. Tutkimuksessa arvioitiin refaktorointien ROI:ta (Return of Investment) eli sijoitetun pääoman tuottoprosentteja vertaamalla refaktoroinnista syntyviä arvioituja ylläpidon säästöjä refaktoroinnin toteutuksen kustannuksiin. Tutkimus osoitti tässä tapauksessa refaktorointiin investoimisen olevan kustannustehokasta, mikäli refaktorointeja tehtiin kuusi tai enemmän. Samankaltaisiin tuloksiin refaktoroinnin kustannuksista päätyy myös Jean-Pierre Fayolle Qualilogyn blogikirjoituksessaan [7]. Hänen mukaansa refaktorointia kannattaa mainostaa hyvänä sijoituksena skeptisille pomoille.

Refaktoroinnin haasteet

Vaikka refaktorointi onkin hyödyllinen ja jopa tuottoisa työkalu, on siinä silti monia haasteita sekä kehittäjä- että yritystasolla. Suurin haaste refaktoroinnin kohdalla on ehkä koko käsitteen väärinymmärrys. Monet mieltävät refaktoroinnin kaikenlaiseksi ohjelmiston muokkaukseksi keskitetyn prosessin sijaan. Refaktorointi nähdään laajana kokonaisuutena, joka vaatii paljon aikaa ja muita resursseja. Refaktorointiin liittyy myös muita pelkoja. Kuvassa 3 esitellään, millaisia riskitekijöitä Microsoftin työntekijät yhdistivät refaktorointiin ja kuinka paljon. [5.]



Kuva 3: Refaktorointiin yhdistetyt riskitekijät Microsoftilla [5]

Kuva 3 kertoo, että refaktoroinnin riskejä tunnistetaan vähemmän kuin sen hyötyjä (kuva 2, s. 5), mutta jotkin pelot ovat huomattavia. Kuten kuvasta näkee, jopa yli 75 prosenttia kehittäjistä pelkää refaktoroinnin aiheuttavan virheitä tai koontikatkoja. Noin 20 prosenttia vastaajista pelkää refaktoroinnin syövän aikaa muilta tehtäviltä ja samoin noin 20 prosenttia ajatteli, että siitä aiheutuu huomattavia testauskuluja.

Pelot ovat osittain realistisia. Refaktoroinnin ei pitäisi muuttaa koodin toiminnallisuutta ollenkaan, mutta rakenteen muokkauskin saattaa aiheuttaa odottamattomia ongelmia, varsinkin jos koodia ei täysin ymmärrä. Huomattavia testauskuluja ei pitäisi myöskään syntyä, mikäli vanhan koodin yksikkötestit ovat ajan tasalla. Sen sijaan täysin yksikkötestittömän koodin refaktorointi luonnollisestikin vaatii enemmän testausta, ja refaktoroinnin yhteydessä olisikin suositeltavaa tehdä yksikkötestit.

Refaktorointi voi viedä aikaa varsinkin, jos koodi on monimutkaista ja sitä on vaikea tulkita. On totta, että laajempien kohteiden refaktorointi voi viedä jopa viikkoja, mutta pääasiassa refaktorointi kehityssyklissä on nopeaa. Refaktoroinnin tekniikat mahdollistavat myös työskentelyn pieni pala kerrallaan, joten suuria kohteita ei tarvitse hoitaa kuntoon yhdeltä istumalta.

Fowlerin [4] mukaan refaktorointia ei myöskään aina vaadita. Hänen mukaansa monimutkaisia koodin osia voi jättää refaktoroimatta, mikäli niitä ei ole tarvetta muokata tai ymmärtää. Hänen mielestään tärkeämpää on refaktoroida aktiivisesti työstettävää koodia.

Aina refaktorointi ei kuitenkaan pure, vaan koodi tarvitsee järeämpiä toimenpiteitä. Joskus ohjelmistot pitäisi kirjoittaa kokonaan uudestaan, esimerkiksi vanhentuneiden kirjastojen tai liian haastavan perustan takia. Uudelleenkirjotus on kuitenkin aikaa vievä ja vaativa prosessi, kuten Ben Morriskin kirjoituksessaan, Why refactoring code is almost always better than rewriting it [8], toteaa. Hänen mielestään kehittäjät ovat usein liian innokkaita tekemään omia ratkaisuja ja helposti sivuuttavat vanhat toteutukset, vaikka ne toimisivatkin. Uudelleenkirjoituksessa esiintyy usein samoja ongelmia kuin alkuperäisen toteutuksen kanssa: aikataulut ja vaatimukset muuttuvat, purkkaratkaisut yleistyvät ja koodista tulee monimutkaista. Uudelleenkirjoitus voi myös aiheuttaa täysin uusia virhetilanteita. Morrisin mukaan vanha ja toimiva koodi kannattaa hyödyntää. Refaktoroinnin avulla voidaan muokata vanhoja toteutuksia ymmärrettävämmiksi aiheuttamatta uusia virheitä.

2.1 Refaktoroitavien kohteiden tunnistus

Miten sitten refaktorointia kaipaavat kohteet voidaan tunnistaa? Joskus koodin ongelmaa on suoralta kädeltä vaikeaa määritellä, mutta siinä on selkeästi jotain mätää. Martin Fowler ja Kent Back kutsuvat tätä mätää koodin haisemiseksi. Heidän mukaansa koodista voidaan löytää pahan hajuisia kohtia, jotka ovat selviä merkkejä refaktoroinnin tarpeesta. Hajun tunnistaminen helpottaa myös oikean refaktoroinnin löytämistä ongelmaan. Fowler ja Beck antavat kirjassaan esimerkkejä näistä hajuista, niiden tunnistuksesta ja korjaustoimenpiteistä. [4.] Seuraavaksi on esitelty yleisimpiä hajuja sekä hajuja, joihin tämän projektin aikana törmättiin.

Mysteerinimi

Luokkien, metodien ja muuttujien nimeäminen vaikuttaa paljon koodin luettavuuteen. Nimen tulee kertoa, mitä luokan tai metodin on tarkoitus tehdä tai mitä tietoa muuttuja sisältää. Jos nimi ei kerro näitä asioita, on kyseessä selvä refaktoroinnin paikka. Jos nimeäminen on haastavaa, voi taustalla olla suurempiakin ongelmia. Esimerkiksi metodin toiminta ei ole yksiselitteistä. [4.]

Duplikaattikoodi

Samankaltaisen koodin löytyminen useasta kohtaa koodia on yleistä, mutta ongelmallista. Samankaltaiset kohdat tulee lukea entistä tarkemmin erojen löytämiseksi ja tällaista koodia muokatessa saman muutoksen joutuu usein tekemään moneen paikkaan. Toistuvien koodien esiintyessä kannattaakin tutkia, voidaanko toteutuksia yhdistää. [4.]

Pitkä metodi

Liian pitkän metodin ongelma on koodin vaikea luettavuus. Metodien koodirivien määrä tulisi pitää noin kymmenessä tai alle, sillä useiden rivien tulkinta on huomattavasti vaikeampaa kuin lyhyiden metodien. Pitkät metodit on myös haastavampi testata. Liian pitkän metodin korjaa helposti pilkkomalla sen pienempiin metodeihin. Pilkkominen voidaan tehdä esimerkiksi kommenttien tai ehtolauseiden perusteella. [4.]

Kookas luokka

Kookkaan luokan tunnistaa pitkän metodin tavoin suuresta koodirivimäärästä tai suuresta määrästä metodeja. Kuten pitkien metodien kanssa, kookkaan luokan suurin ongelma on luettavuuden kärsiminen. Suuret luokat yrittävät usein tehdä liikaa asioita kerralla, tai ne sisältävät turhia luokkamuuttujia, joita ei lähes koskaan käytetä. Luokkien toimintaa voidaan usein hajauttaa uusiin luokkiin tai siirtää yläluokalle. [4.]

Hylätty periytyminen

Periytyminen on hyödyllinen tapa jatkaa luokkien toimintaa ja se on yleinen käytäntö olio-ohjelmoinnissa. Kuitenkin valitettavan usein aliluokat eivät tarvitse kaikkea perimäänsä, vaan perintö hylätään. Tällöin koodin hierarkia on usein ongelmallinen ja siihen tulisi puuttua refaktoroinnin avulla. [4.]

Sisäkkäiset tai liian pitkät ehtolauseet ja silmukat

Ehtolauseet ja silmukat ovat ensimmäisiä asioita, joihin ohjelmoinnin parissa tutustutaan. Ne ovatkin hyödyllinen työkalu, mutta niissäkin piilee vaaran paikka. Koodia muokatessa ehtolauseet helposti laajenevat ja monimutkaistuvat tarpeettoman paljon, jolloin niitä on vaikea tulkita. Ehtolauseet usein myös aiheuttavat useita ulostuloja samasta metodista, mitä tulisi aina välttää. Ehtolauseet on hyvä pitää yksinkertaisina ja tarvittaessa niistä on eristettävä omia pienempiä metodeja. [4.]

Fowlerin ja Beckin määrittelemien hajujen lisäksi potentiaalisia refaktoroinnin kohteita ovat hyviä koodauskäytäntöjä rikkovat toteutukset. Hyviä olio-ohjelmoinnin käytäntöjä ovat esimerkiksi SOLID-periaatteet. SOLID on akronyymi, joka kuvaa seuraavaa viittä olio-ohjelmoinnin tärkeää periaatetta:

- (SRP) Single-Responsibility Principle: luokalla tulee olla vain yksi vastuualue.
- (OCP) Open-closed Principle: olioiden tulee olla avoimia laajennuksille, mutta suljettuja muutoksille.
- (LSP) Liskov Substitution Principle: yläluokan olioiden tulee olla korvattavissa sen aliluokan olioilla.
- (ISP) Interface Segregation Principle: asiakasohjelmien ei pidä olla riippuvaisia rajapinnoista, joita ne eivät käytä.
- (DIP) Dependency Inversion Principle: luokkien ei tule olla riippuvaisia käyttämiensä luokkien toteutuksista, vaan niiden abstraktioista (rajapinnoista). [9.]

Näitä periaatteita rikkovat toteutukset ovat yleensä refaktoroinnin tarpeessa ja sisältävät myös muita koodihajuja. Monet refaktoroinnin tekniikat korjaavat myös SOLID-käytäntöjä rikkovia toteutuksia.

Kaikkia refaktoroinnin kohteita ei tarvitse löytää yksin. Ohjelmistotiimeissä refaktorointia tukevat koodikatselmoinnit ja vertaisarvioinnit. Ulkopuolisen silmin voidaan löytää paremmin kehityskohteita koodista, kuin koodin toteuttaja voisi itse löytää. Osa koodikatselmoinnista voidaan myös automatisoida erilaisilla tarkastustyökaluilla kuten SonarQubella. Monet automaattiset koodikatselmointityökalut tarkistavat koodin laatuun liittyviä asioita, kuten testikattavuutta ja tietoturvallisuutta, mutta myös etsivät refaktorointia vaativia kohteita, kuten duplikaattikoodia tai hajuja. Hyvät kehityksen käytännöt ja työkalut helpottavat refaktorointia ja vähentävät teknistä velkaa.

2.2 Yleiset refaktoroinnin tekniikat

Tässä kohtaa esitellään yleisimpiä refaktoroinnin tekniikoita, jotka auttavat muun muassa edellisessä luvussa mainittuihin hajuihin ja SOLID-rikkeisiin. Seuraavana olevat tekniikat ovat Fowlerin ja Beckin kirjasta [4], jossa he listaavat myös runsaasti muita tekniikoita refaktorointien katalogissaan. Tässä kohdassa on tarkoitus esitellä vain yleisimmät tekniikat ja sellaiset, jota tämän projektin aikana tullaan hyödyntämään. Refaktoroinnin tekniikoissa usein yhdistetään useampaa yksinkertaista tekniikkaa, joten esittely tehdään alimmalta tasolta.

Rename method – Metodin uudelleennimeäminen

Refaktoroinneista yleisimpiä ovat metodien ja muuttujien uudelleennimeämiset. Uudelleennimeäminen on helppo tapa refaktoroida koodia ymmärrettävämmäksi, sillä kuvaava nimi kertoo heti kutsujalle, mitä metodin tai luokan tulisi tehdä tai millaista tietoa muuttuja sisältää. Metodien nimien muokkauksen lisäksi kannattaa miettiä sen saamien parametrien kuvaavaa nimeämistä. Tämä refaktorointi on helppo toteuttaa muokkaamalla nimiä käsin tai hyödyntämällä useissa ohjelmointikehittimissä olevia refaktorointitoimintoja, joilla muutettu nimi muuttuu kaikkialla, missä sitä käytetään. [4.]

Pull up method/Field – Metodin tai rivin nostaminen

Saman tasoisissa aliluokissa esiintyvät duplikaatit, ja samankaltaiset toteutukset voidaan korjata nostamalla yhtenäiset osat yläluokkaan. Aluksi varmistetaan, että siirrettävät kohdat ovat aidosti samanlaisia. Tämän jälkeen yläluokkaan luodaan uusi metodi ja alaluokan koodi kopioidaan siihen tai alaluokan rivit kopioidaan yläluokkaan. Yhden alaluokan kopioitu metodi voidaan poistaa ja testata. Mikäli kaikki toimii, kopioidut osat voidaan poistaa muistakin alaluokista. Rivien ja metodien nostaminen vähentää duplikaattikoodia, sekä lyhentää metodeja ja pienentää aliluokkien kokoa. [4.]

Extract method – Metodin eristäminen

Liian pitkien ehtolauseiden, metodien tai luokkien pilkkomiseksi yleisin refaktoroinnin tekniikka on metodin eristäminen. Tämän tekniikan avulla koodiosioista pitkistä koodiosioista eristetään uusia lyhyempiä metodeja. Tässä tekniikassa luodaan uusi metodi ja annetaan sille kuvaava nimi, jotta koodi olisi luettavaa ja itsessään dokumentoivaa. Uuteen metodiin kopioidaan pitkästä metodista haluttu koodi ja metodille annetaan tarvittavat parametrit alkuperäisestä koodista. Tämän jälkeen vanha koodi korvataan uudella metodilla ja metodi voidaan testata. Uuden metodin kohdalla on myös hyvä katsoa, voidaanko toteutusta hyödyntää uudelleen muissa kohteissa. Metodin eristäminen toimii usein liian pitkien metodien ja luokkien tai duplikaattikoodin korjaamiseksi. Metodien hyvä nimeäminen myös vähentää kommentoinnin tarvetta. [4.]

Extract class – Luokan eristäminen

Pitkien luokkien korjaamiseksi voidaan hyödyntää luokan eristämistä. Jos luokka rikkoo SRP:tä, eli se tekee useampaa kuin yhtä toimintoa, voidaan erilaiset toiminnot eristää omiksi luokikseen. Luokkaa eristäessä valitaan siirrettävät vastuualueet ja luodaan luokalle uusi rinnakkainen luokka, joka toteuttaa nämä vastuut. Alkuperäiseen luokkaan lisätään linkki uuteen luokkaan, ja kaikki alkuperäisestä luokasta valitut rivit ja metodit siirretään uuteen luokkaan. [4.]

Extract superclass – Yläluokan eristäminen

Usein huomaa samantasoisten luokkien toteuttavan liian samanlaisia metodeja ja koodissa on paljon duplikaatioita. Luokat toimivat keskenään hyvin samalla tavalla, mutta eivät kuitenkaan ole yhteydessä. Silloin voidaan näistä yhtenäisyyksistä eristää kaikille luokille yhteinen yläluokka, johon siirretään yhteiset metodit ja muuttujat. Yläluokan eristämisessä luodaan rinnakkaisille luokille yhteinen yläluokka ja saman tasoisten luokkien yhteiset koodit viedään yläluokkaan metodin tai rivin nostamistekniikalla. [4.]

3 Korjausvelkalaskennan sovellus

Kove eli korjausvelkalaskennan työkalu on vuonna 2017 Rapalin Fore-tuoteperheeseen lisätty sovellus. Fore on infran kustannuksien laskenta- ja hallintaohjelmisto. Fore koostuu viidestä eri osaohjelmistosta, joista yksi on Kove. Kove on erillisellä lisenssillä toimiva sovellus infraomaisuuden kunnon ja kehityksen seurantaan sekä korjausvelan laskentaan. Aluksi Kovea voitiin käyttää vain katujen ja vesihuoltoverkoston laskentaan, mutta vuodesta 2019 lähtien Kovea on pystynyt hyödyntämään myös rakennusten korjausvelan laskentaan. Kovessa rakennusten laskenta on erillään muiden omaisuuserien laskennasta, joten tässäkin dokumentissa infraomaisuuksista puhuttaessa tarkoitetaan vain katuja ja vesihuoltoverkostoa.

Koven avulla asiakkaat, kuten kunnat, voivat laskea katujen, vesihuoltoverkostojen sekä rakennusten uudishinnan ja korjausvelan määrän, ennustaa tulevien vuosien korjausvelan kehityksen omaisuudelle ja priorisoida kriittisimmät kohteet. Asiakkaiden on myös mahdollista raportoida infraomaisuuden kunto ja arvo vuosittain.

Kove hyödyntää laskennassa Foren sisäisiä kustannustietoja ja hankeosamalleja. Hintatiedot päivitetään Foreen kerran puolessa vuodessa, jolloin laskelmat pysyvät ajantasaisina. Korjausvelan laskenta perustuu Kuntaliiton kanssa tehtyihin määritelmiin ja laskentamalleihin, jotka on kehitetty kuntainfran toimijoiden yhteistyöfoorumin kanssa. [10.]

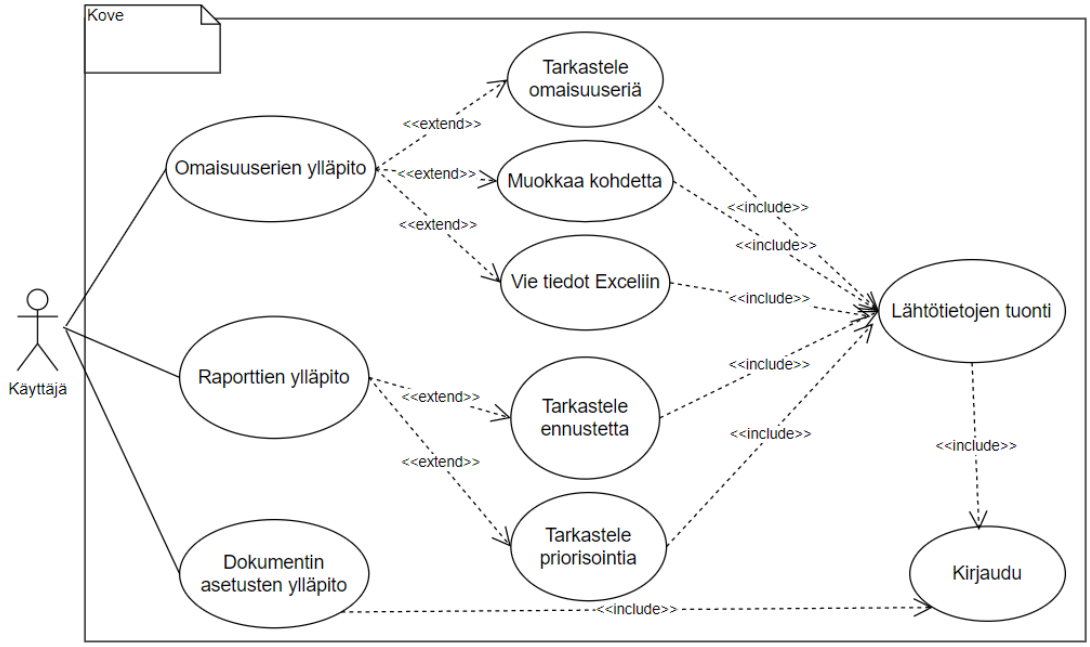
Kove on tehty korvaamaan aiemmin manuaalisesti tehdyt korjausvelkalaskelmat. Ohjelmiston avulla asiakas voi tehdä laskelmat sekä tarvittavat päivitykset ja muutokset nopeasti ja kustannustehokkaasti. Seuraavaksi esitellään tarkemmin Koven toimintoja ja käyttötapauksia sekä sovelluksen tekninen kuvaus.

3.1 Toiminnallinen kuvaus

Koven käyttöön vaaditaan erillinen lisenssi, joka voidaan ottaa käyttöön yksittäiselle henkilölle tai organisaatiolle. Kove toimii käytännössä Foressa erillisinä Kove-dokumentteina, jotka sisältävät laskennat ja muut Koven toiminnallisuudet. Näihin dokumentteihin on lisenssin lisäksi erilliset luku- ja muokkausoikeudet organisaation sisällä tai niiden

välillä. Näin esimerkiksi konsulteille voidaan jakaa lukuoikeus organisaation dokumentteihin. Kove-lisenssillä käyttäjä voi Foren Portal-näkymässä tarkastella organisaationsa dokumentteja tai niitä dokumentteja, jotka käyttäjälle tai tämän organisaatiolle on jaettu.

Kove-dokumentteja tarkastellaan verkkoselaimessa. Kovessa voi tarkastella dokumentin yleistietoja, tuoda dokumentille infra- tai rakennusdataa Excel-muodossa laskettavaksi ja tarkastella tuotujen tietojen omaisuuseriä ja priorisoida infraomaisuuden kohteita erilaisin perustein. Kuvassa 4 esitellään Koven toimintaa käyttötapauskaaviolla. Koven tärkeimpiä käyttökohteita ovat kohteiden tuonti laskentaan ja tuotujen kohteiden korjausvelan tarkastelu ja vienti sekä kohteiden priorisointi, eli järjestäminen korjaustarpeen mukaan, laskettujen arvojen perusteella.



Kuva 4: Koven käyttötapauskaavio

Käyttötapauskaavion (kuva 4) toimijana on käyttäjä, joka voi kirjauduttuaan tarkastella valitsemansa dokumentin perustietoja, muokata dokumentin asetuksia tai tuoda dokumentille lähtötietoja. Lähtötietojen tuonnin jälkeen käyttäjän on mahdollista tarkastella tuodun kohteen yksikköhintoja, omaisuuseriä tai raportteja. Tuotuja kohteita voi Omaisuuserät-näkymässä tarkastella ja muokata. Tuotuja infrakohteita voi myös raportoinnissa priorisoida.

Koven lähtötiedot tuodaan Excel-muodossa laskettavaksi. Tietojen tulee olla tietyssä muodossa Excelissä, jotta ohjelma hyväksyy tuonnin. Katujen, vesihuoltoverkoston ja rakennusten tiedot tuodaan erikseen Tuonti-välilehdellä. Tuonti-välilehdellä on myös eritelty, mitä tietoja laskelmaan voidaan tuoda ja mitkä tiedoista ovat pakollisia. Esimerkiksi kuvassa 5 on esitelty katujen tuonnin malli.

Tiedoston rakenne

Sarake	Otsikko	Pakollinen	Tyyppi
A	ID	Ei	Teksti
B	Nimi	Ei	Teksti
C	Osoite	Kyllä	Teksti
D	Pinta-ala (m²)	Kyllä	Numero
E	Rakennusvuosi	Kyllä	Teksti
F	Toiminnallinen luokka	Kyllä	Teksti
G	Pohjaolosuhde	Kyllä	Teksti
H	Katuluokka	Kyllä	Numero
I	Kulutuskerros	Kyllä	Teksti
J	Saneerausvuosi	Ei	Numero
K	Mitattu kuntotaso (%)	Jos mittausvuosi on annettu	Numero
L	Mittausvuosi	Jos mitattu kuntotaso on annettu	Numero

Kuva 5: Katujen tuonnin rakenne

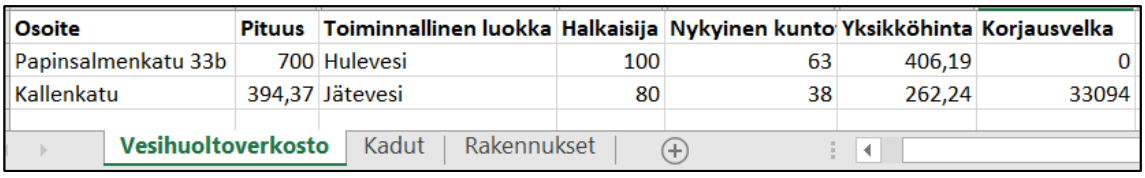
Ohjelma ilmoittaa, mikäli tuonnissa on ollut virheitä. Vääränlaisen datan korjaamiseksi ohjelma kertoo, missä Excelin soluista virhe on ollut ja antaa tarkempaa tietoa oikeasta datasta kyseiseen soluun. Onnistuneen tuonnin jälkeen käyttäjä voi tarkastella tuotuja kohteita Omaisuuserät-välilehdellä (kuva 6).

Osoite	Pituus (m)	Toiminnallinen luokka	Uudishinta (€)	Korjausvelka (€)	Korjauskustannukset (€)	
Papinsalmenkatu 33a	7	Vesijohto	2 558	1 727	2 866	Näytä lisätietoja
Papinsalmenkatu 33b	80	Vesijohto	27 896	22 317	31 410	Näytä lisätietoja

Kuva 6: Vesihuoltoverkoston omaisuuserät

Omaisuuserän sivulla näkyvät yksittäisten rivien tulokset erikseen kaduille, vesihuoltoverkostolle ja rakennuksille. Näkymässä on tuotujen tietojen lisäksi ohjelman kohteelle laskemat nykyinen kuntotaso, optimikuntotaso, yksikköhinta, uudishinta, korjausvelka, sekä korjauskustannukset. Rivejä on mahdollisuus muokata ”Näytä lisätietoja” -kohdasta.

Tuodut omaisuuserät ja niiden lasketut arvot, kuten korjausvelka ja -kustannukset, voidaan viedä Exceliin Omaisuuserät-osion Vienti-välilehdeltä. Viennissä kaikki dokumentille tuodut omaisuuserät viedään samaan Excel-dokumenttiin omaisuuserän tyypistä riippumatta. Vaikka laskelmaan olisi tuotu vain vesihuoltoverkoston rivejä, tulevat Excelille myös muiden omaisuuserien välilehdet, kuten kuvassa 7. Nämä välilehdet ovat kuitenkin tyhjiä. Excelille tulostuvat kaikki laskemaan tuodut rivit sekä niille lasketut arvot.

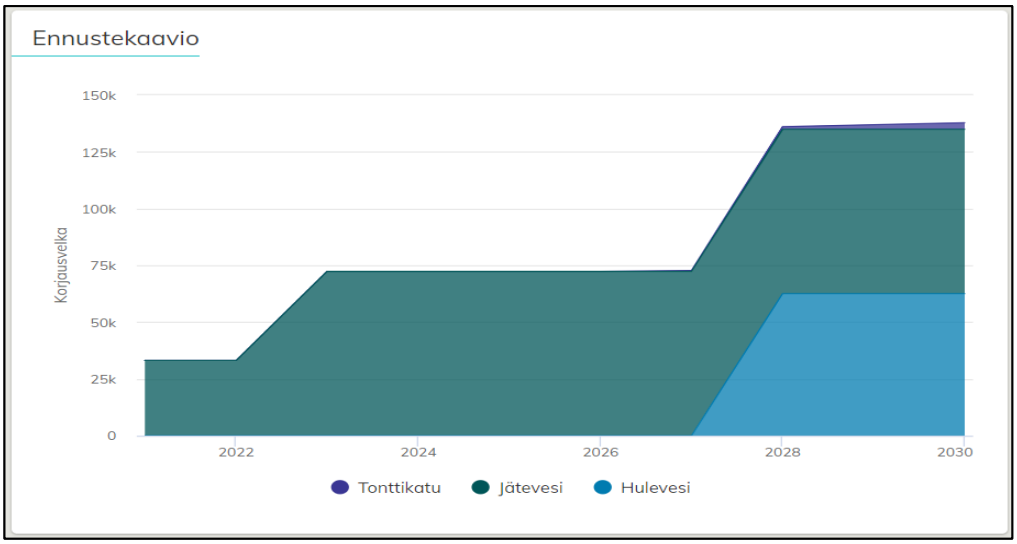


Osoite	Pituus	Toiminnallinen luokka	Halkaisija	Nykyinen kunto	Yksikköhinta	Korjausvelka
Papinsalmenkatu 33b	700	Hulevesi	100	63	406,19	0
Kallenkatu	394,37	Jätevesi	80	38	262,24	33094

Vesihuoltoverkosto | Kadut | Rakennukset

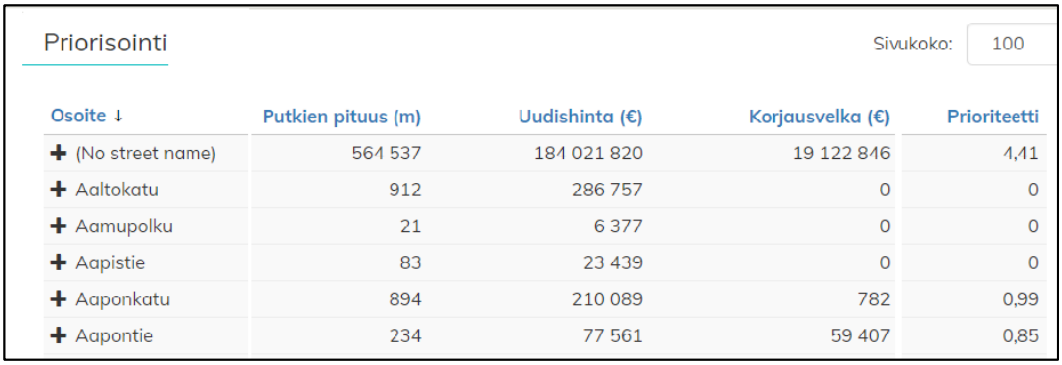
Kuva 7: Kovesta viety Excel-tiedosto

Raportoinnin osiossa on kolme erilaista toimintoa. Laskennan tuloksia voidaan tarkastella Yhteenvedossa, jossa tulokset on eritelty rakennustyyppien ja toiminallisten luokkien mukaan. Ennuste-osassa esitetään laskelman mukainen ennuste korjausvelan kehittymisestä seuraavan 10 vuoden aikana. Ennusteen tavoitteena on esittää, miten tuotujen omaisuuserien korjausvelka kehittyisi mallin mukaisesti, mikäli kohteille ei tehtäisi korjaustoimenpiteitä. Sivulla esitetään euromääräinen tai suhteellinen korjausvelan kehitys lukuina sekä graafisena kuvaajana, kuten kuvassa 8 on esitetty.



Kuva 8: Ennustekaavio

Raportoinnin kolmas toiminto on kohteiden priorisointi. Priorisoinnilla voidaan tuoda esiin kriittisimmät toimenpiteitä tarvitsevat kohteet. Priorisoinnissa kaikki kohteet pisteytetään tiettyjen kategorioiden mukaan. Kriittisin kohde saa suurimmat pisteet ja loput skaalautuvat tästä alaspäin. Priorisointi on tässä vaiheessa mahdollista vain kaduille ja vesihuoltoverkoston kohteille. Priorisointia varten tuotujen katujen ja vesihuoltoverkoston osat kootaan kokonaisuuksiksi osoitteiden perusteella. Jos laskelmalle on tuotu sekä katuja että vesihuoltoverkostoa, ne priorisoidaan yhdessä. Kuvassa 9 on esitelty infraomaisuuden priorisointi. Priorisoinnissa kohteet järjestetään aluksi osoitteen mukaan, mutta käyttäjä voi valita taulukon otsikoista, miten järjestää kohteet.


Priorisointi

Sivukoko: 100

Osoite ↓	Putkien pituus (m)	Uudishinta (€)	Korjausvelka (€)	Prioriteetti
+ (No street name)	561 537	184 021 820	19 122 846	1,11
+ Aaltokatu	912	286 757	0	0
+ Aamupolku	21	6 377	0	0
+ Aapistie	83	23 439	0	0
+ Aaponkatu	894	210 089	782	0,99
+ Aapontie	234	77 561	59 407	0,85

Kuva 9: Infraomaisuuden priorisointi

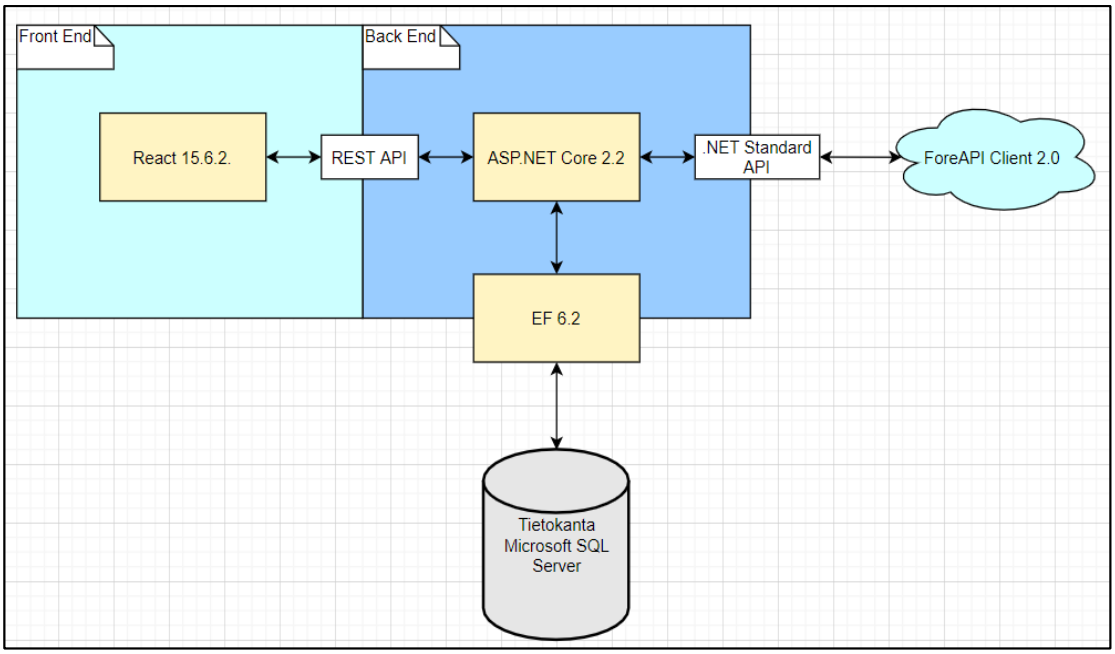
Kuvassa 10 näkyvät infraomaisuuden priorisointikategoriat ja niiden oletusmaksimipistemäärät. Näitä oletusmaksimipisteitä voidaan muokata Asetukset-sivulla Priorisoinnin pisteytys -osiossa. Priorisointipisteytys tapahtuu siten, että kriittisin kohde, eli rivi, jonka priorisointikategoriaa vastaava luku on suurin, saa kategorian mukaisen maksimipistemäärän, ja muut kohteet pisteytetään skaalautuvasti siitä alaspäin. Esimerkiksi mikäli Korjausvelka-kategorian maksimipistemäärä on 2, rivi, jolla on eniten korjausvelkaa, saa kaksi pistettä. Muut kohteet saavat vähemmän pisteitä korjausvelkansa määrän perusteella siten, että ne kohteet, joilla ei ole lainkaan korjausvelkaa, eivät saa pisteitä. Rivin kokonaisprioriteetti on riville jokaisen oletuskategorian perusteella laskettujen prioriteettipisteiden summa, ja siten kriittisimmän kohteen pisteet voivat enimmillään olla oletuspisteiden summa. Käyttäjä voi muokata oletuspisteitä ja siten määrittää tärkeimmän kriteerin priorisoinnille. Jos jotakin kategoriaa ei haluta ottaa priorisoinnissa ollenkaan huomioon, voidaan sen oletuspistemääräksi asettaa 0.

Kokonaiskorjausvelka:	2,00
Katujen kokonaispinta-ala:	1,50
Putkien kokonaispituus:	1,50
Katujen kuntotasojen keskiarvo (%):	1,00
Katujen keskimääräinen toiminnallinen luokka:	1,00
Putkien keskimääräinen halkaisija:	1,00
Putkien keskimääräinen kriittisyys:	1,00

Kuva 10: Priorisointikategoriat ja niiden oletuspisteet

3.2 Tekninen kuvaus

Kove on vuonna 2017 toteutettu erillinen moduuli Foren tuotteisiin. Koven teknologiapino muodostuu käyttöliittymälle Reactista, backendille ASP.NET Coresta ja tietokannan mallinnukselle Entity Frameworkista. Ohjelmiston tietokantaa hallitaan Microsoft SQL Serverillä. Nämä komponentit yhdistyvät toisiinsa kuvan 10 mukaisesti.



Kuva 11: Koven komponenttikaavio

Kuvan 11 komponenttikaaviosta näkee, että Koven käyttöliittymä on toteutettu käyttäen Reactin versiota 15.6.2. React on Facebookin kehittämä avoimen lähdekoodin JavaScript-kirjasto käyttöliittymien ja UI-komponenttien luomiseen. React huolehtii vain datan renderöinnistä DOM:lle ja siksi käyttöliittymän tilojen ja reitityksen hallintaan vaaditaan erilliset kirjastot. [11.] Kovessa reititykseen käytetään React Routeria ja tilojen hallintaan Reduxia.

Tiedot käyttöliittymän ja backendin välillä kulkevat REST-rajapinnan yli. Sovellus ja backend on toteutettu .NET Coren ja ASP.NET Coren 2.2-versioilla. .NET Core on Microsoftin ylläpitämä kehitysalusta erityyppisille sovelluksille [12]. ASP.NET Core puolestaan on .NET Core -alustalle optimoitu ohjelmistokehys pilvipohjaisten sovellusten, kuten web-sovellusten, kehitykseen [13].

Foren muista sovelluksista poiketen Kove käyttää omaa tietokantaansa. Tätä tietokantaa hallinnoidaan Microsoft SQL Serverillä. Tietokantamallinnus, eli ORM on toteutettu Entity Framework, EF, 6.2. versiolla. EF on .NET kanssa yhteensopiva tietokantamallinnuksen kehys. EF toteuttaa mallinnuksen luomalla DTO:ta, eli tiedonvälitysoliota. EF:n avulla

tietokantakyselyihin voidaan käyttää LINQ-kyselyitä, jotka tietokannassa kääntyvät SQL-kieleksi. [14.]

Koven varsinainen kustannuslaskenta ei tapahdu suoraan Kove-ohjelmassa, vaan laskenta tulee Foren malleilta. Tämän vuoksi Koven backend hyödyntää laskentavaiheessa komponenttikaavion (kuva 11) mukaisesti ForeAPIa.

ForeAPI sisältää kustannuslaskennan mallit, ja Kove käyttää tätä APIa .NET Standardin avulla. .NET Standard on joukko yhtenäisesti määriteltyjä ohjelmointirajapintoja, jotka kaikkien .NET-toteutusten on toteutettava. Tämä standardi mahdollistaa .NET-koodin ajon eri paikoissa, kuten Kovesta Foren puolella. [15.]

4 Työn alkuvalmistelut

Tässä osiossa esitellään työn tarkempi määrittely, suunnittelun eteneminen ja Koven kehityskohteet. Kehityskohteista esitellään jatkokehityksen kohteet, eli korjattavat virheet ja lisättävät ominaisuudet, sekä kuinka nämä kohteet valittiin kehitettäviksi. Samalla esitellään näihin muokkauksiin liittyvät refaktorointia kaipaavat kohteet. Tavoite on esittää, miksi juuri nämä kohteet valikoituivat refaktoroitaviksi.

Toisena tässä osiossa kerrotaan, millaisia ohjelmistoja ja työkaluja projektin toteutukseen tullaan käyttämään. Tavoitteena on esitellä Koven kehityksessä jo aiemmin hyödynnettyjä työkaluja sekä tämän työn aikana Koveen integroitava SonarQube. Tarkoitus on kertoa, mikä SonarQube on ja miten sitä tullaan hyödyntämään tämän projektin aikana ja sen jälkeen Koven kehityksessä ja refaktoroinnissa.

4.1 Suunnittelu ja kehityskohteet

Koven jatkokehityksen suunnittelu aloitettiin kehitystiimin ja tiedostotiimin yhteisellä suunnittelupalaverilla. Palaverin tarkoituksena oli kartoittaa, mitä ominaisuuksia Koveen halutaan lisätä projektin aikana ja millaisia korjauksia on tehtävä olemassa oleviin toimintoihin. Määrittelyssä otettiin huomioon asiakaspyynnöt sekä Koven toiminnassa huomatut virheet. Tehtävien määrittelyn jälkeen kehittäjät kävivät keskenään läpi muokattavan koodin alustavia refaktoroinnin tarpeita.

Suunnittelupalaverien jälkeen pidettiin kehitystiimin keskeinen kehitysjonon jalostamispalaveri, jossa suunnittelupalaverissa määritellyt tehtävät tarkasteltiin läpi kehittäjien ja testaajan näkökulmasta ja pisteytettiin tehtävän arvioidun vaativuuden mukaan. Näiden määrittelyjen tuloksena projektille valikoituivat alla esiteltävät kehityskohteet.

Koveen on vuonna 2019 lisätty rakennusten korjausvelkalaskenta. Rakennuksilla ei kuitenkaan vielä ole kaikkia toimintoja, jotka infraomaisuudelle voi Kovessa toteuttaa. Tämän projektin tavoitteena on laajentaa Kovea myös rakennusten priorisoinnin osalta. Itse priorisointia on myös tarkoitus kehittää ja siirtää priorisointi omalle välilehdelleen, jonne halutaan siirtää myös priorisoinnin asetusten muokkaus.

Omat priorisointikategoriat ja tuonnin kehitys

Asiakkaiden puolelta on Koven priorisointiin jo pidempään toivottu mahdollisuutta lisätä omia priorisointikategorioita oletuskategorioiden (kuva 10, s. 19) rinnalle. Asiakkaat haluavat mahdollisuuden lisätä rajaamattoman määrän omia kategorioitaan kohteiden priorisointiin. Näiden kategorioiden halutaan toimivan samoin kuin oletuskategoriat, eli niille voitaisiin asettaa oletuspisteet sekä kohteiden tuonnin yhteydessä priorisointipisteet, jotka yhdessä vaikuttavat kohteen prioriteetin laskentaan. Omien kategorioiden tuonnin lisäksi ne halutaan aiempia arvoja vastaavasti myös viedä järjestelmästä Exceliin.

Tuonnin kehityksen yhteydessä on aiemmin havaittu haasteita, kun uudentyyppistä dataa halutaan lisätä järjestelmään. Tuonnissa Excel-syötteiden tarkistus ja muunnos Koven käyttämiksi datatyypeiksi tapahtuvat yhdessä pitkässä metodissa. Metodi on vaikeasti luettava ja muokattava. Kyseessä on selvä pitkä metodi -koodihaju. Samalla metodi ja metodin luokka rikkovat SOLID-periaatetta SRP [9], sillä luokan ei tee vain yhtä toimintoa, vaan validoi ja luo datatyypit. Tämä aiheuttaa sekaannusta, ja toiminnot olisi hyvä saada eri luokkiin.

Myös tuonnin virheilmoituksissa on havaittu ongelmia. Katujen ja vesihuoltoverkoston tuonnissa rivin kaikkia virheitä ei aina ilmoiteta käyttäjälle, vaan tarkistus on jo pysähtynyt ensimmäisen virheen kohdalla. Tämä halutaan korjata siten, että kaikki tuonnin virheet näytetään käyttäjälle yhtä aikaa, jotta käyttäjä voi korjata kaikki virheet kerralla. Ongelman korjaamiseksi tuonnissa tapahtuvaa rivien validointia halutaan muokata ja toteutusta refaktoroida. Validoinnin suurimpia ongelmia ovat pitkä vaikealukuinen silmukkarakenne, josta syntyy useita ulostuloja koodissa. Tätä rakennetta on korjattava ja validoinnin toteutuksia yhtenäistettävä infraomaisuuksien ja rakennusten välillä samalla, kun virheilmoitusongelma korjataan.

Viennin kehitys

Koven Excel-vienti on asiakkaan näkökulmasta epäkäytännöllinen. Kaikki infratyypit viedään samaan Exceliin, vaikka jokin niistä olisikin tyhjä. Vienti halutaan tämän vuoksi toteuttaa omaisuuseräkohtaisesti. Samalla muokataan viennin koodia, jossa on samoja ongelmia kuin tuonnissa. Vienti toteutetaan pitkässä metodissa, joka on vaikealukuinen.

Viennin luokka sisältää myös paljon duplikaattikoodia, joka eroaa toisistaan vain vähän. Viennin metodit halutaan pilkkoa ja duplikaattikoodia vähentää yhdistämällä samanlaisia osia omiksi metodeikseen.

PriceServicen refaktorointi

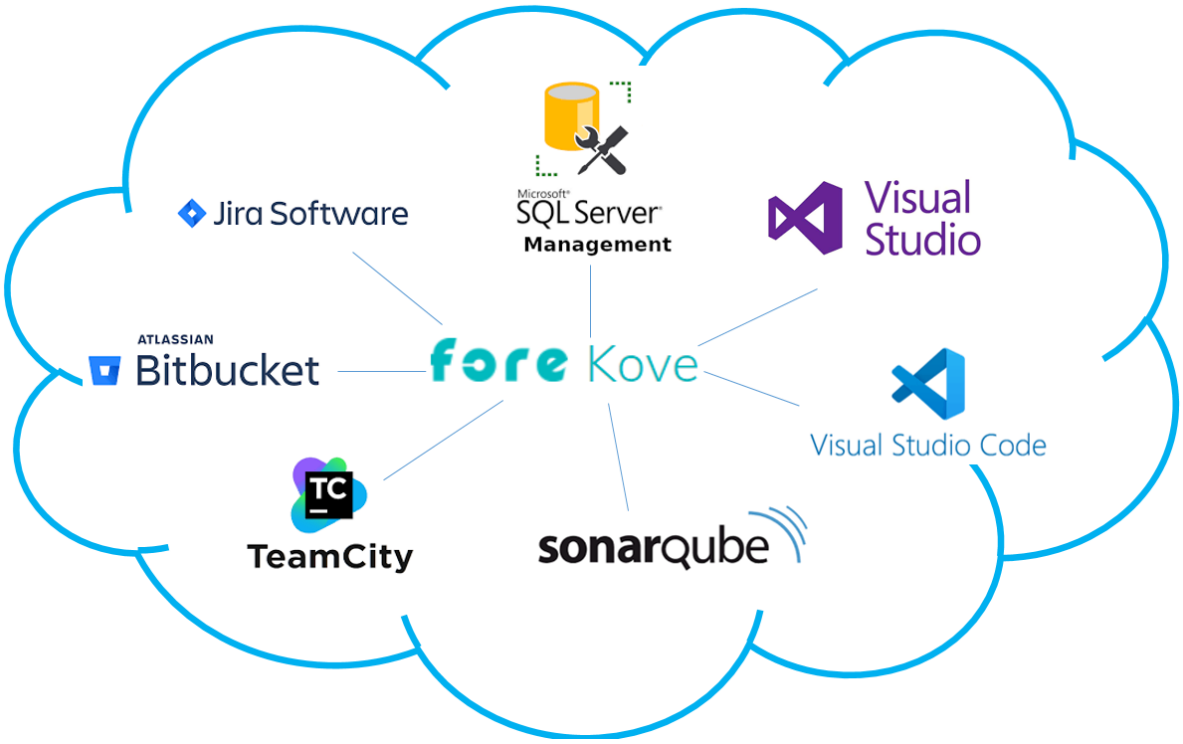
Normaalista kehityssyklistä poiketen kehitykseen halutaan ottaa myös yksi pelkkää refaktorointia vaativa kohde: Koven PriceService. Kehittäjien kanssa käydyssä aloituspalaverissa todettiin tämän kokonaisuuden olevan erittäin monimutkainen ja vaikeasti laajennettava. PriceServicet ovat Koven tärkeimpiä osia ja vastaavat omaisuuserien hintojen laskennasta. PriceServcejä ei tarvitsisi tässä vaiheessa muokata, mutta niiden koetaan hidastavan kehitystä huomattavasti. Tämän vuoksi PriceServicien siivoaminen päätettiin ottaa työn alle. PriceServicet koostuvat yläluokasta BasePriceService ja kolmesta omaisuuserän mukaisesta alaluokasta. PriceServicien vanha toteutus on esitelty tämän työn lopun liitteessä 1. Kuten liitteessä olevasta luokkakaaviosta näkee, varsinkin BuildingPriceService sisältää paljon duplikaattimetodeja, jotka eroavat toisistaan vain yksittäisten parametrien tyypin verran. Duplikaatiot on korostettu liitteessä.

Kaikki alaluokat eivät kuitenkaan tarvitse kaikkea yläluokasta perimäänsä ja hylkäävät perinnän. Esimerkiksi liitteen 1 kaavion Street- ja PipePriceServicet eivät toteuta turhaa BulkFetchPrices-metodia, vaan toteutus heittää NotImplemented-virheen. Tämä johtaa myös SOLID LSP -rikkeeseen, kun yläluokan oliot eivät aina ole korvattavissa kaikilla alaluokan olioilla. Kaikki alaluokat ovat liian laajoja ja siksi hankalasti luettavissa. Alaluokkien välillä on myös paljon vain vähän toisistaan eroavia duplikaatteja. Refaktoroinnilla halutaan siirtää kaikki yhtenäinen toteutus yläluokkaan ja jättää alaluokille vain eroavat osat.

Refaktoroinnit on tarkoitus tehdä pääasiassa muokkausten yhteydessä. Esiteltyjen isompien ongelmien lisäksi korjataan koodista samalla pienempiä ongelmia kuten nimeämisiä. Seuraavassa aliluvussa esitellään työssä käytetyt ohjelmistot ja työkalut sekä projektin aikana Koven jatkuvan kehityksen putkeen lisättävä SonarQube, jota tullaan myöhemmin käyttämään myös refaktoroinnin apuna.

4.2 Käytetyt ohjelmistot

Työn toteutukseen käytettiin useita erilaisia ohjelmistoja ja työkaluja ohjelmointiin, testaukseen, versionhallintaan ja jatkuvaan koontiin. Suurinta osaa työkaluista oli käytetty aiemmin Koven kehityksessä ennen projektin alkua ja samoja työkaluja haluttiin hyödyntää myös tässä projektissa. Kuvassa 12 esitellään käytetyt ohjelmistot ja työkalut ja niistä kerrotaan seuraavaksi tarkemmin.



Kuva 12: Koven kehityksessä käytetyt työkalut

Tärkein työkalu projektin hallinnassa ja dokumentaatiossa oli Atlassianin Jira. Jira on projektinhallinnanohjelmisto, joka hyödyttää erityisesti ketterien kehitystiimien projektien suunnittelua, seurantaa ja raportointia [16]. Projektin kaikki tehtävät kirjattiin Jiran kehitysjonoon ja niille annettiin tarinapisteet tehtävän haastavuuden arvioimiseksi asteikolla 1–5. Projektin aikana kehittäjät ottivat kehitysjonosta tehtäviä Jiran kehitystaululle, jossa niiden edistymistä kehityksestä, koodikatselmointiin, testaukseen ja lopulta valmistumiseen seurattiin. Tarinapisteiden arviointiin sisällytettiin myös mahdollisten refaktorointien arvio, sillä refaktoroinnit ovat osa kehitystä.

Projektin ohjelmistokehittiminä on käytetty pääasiassa Visual Studiota backend-koodin kehitykseen ja Visual Studio Codea frontend-koodin kehitykseen. Tietokantojen hallintaan on käytössä ollut Microsoft SQL Server Management Studio. Ohjelmointikieliä projektissa ovat olleet C#, jota käytettiin sekä front- että backendin kehitykseen. Muita käytössä olevia kieliä olivat JavaScript, HTML ja CSS.

Projektin versionhallintaan käytössä on ollut Bitbucketin Git. Se on Git-koodinhallinnan työkalu Atlassianilta. Bitbucket mahdollistaa koodihaarojen (branch) luomisen suoraan Jiran tehtävistä [17]. Projektin aikana jokaiselle sovitulle tehtävälle luotiin oma ominaisuushaara, joka yhdistettiin (merge) valmiina kehityshaaraan.

Ennen ominaisuushaarojen yhdistämistä kehityshaaraan kaikki koodi testataan jatkuvan koonnin ympäristössä TeamCityssä. TeamCity on JetBrainsin luoma koontiversioiden hallinnan ja jatkuvan integraation palvelin [18]. TeamCity tarkistaa projektin koonnin ja ajaa automaattisesti yksikkötestit.

Kuten jo mainittu, teknisen velan hallintaan voidaan hyödyntää automaattisia analyysityökaluja. Koven kehityksessä ei olla aiemmin hyödynnetty tällaisia työkaluja, mutta tämän projektin aikana asiaan halutaan tehdä muutos. Rapalilla on viime vuoden aikana alustavasti testattu SonarQubea jatkuvan koodianalysoinnin työkaluna muissa projekteissa. Nyt se halutaan liittää myös Koven jatkuvan integraation putkeen. SonarQube on automaattinen koodintarkastustyökalu, joka on yhteensopiva niin TeamCityn kuin Visual Studion kanssa. Se tarkistaa koodista automaattisesti muun muassa testikattavuutta ja koodin haavoittuvuuksia. Samalla se tutkii koodiin muodostunutta korjausvelkaa etsimällä koodista hajuja ja duplikaatioita [19].

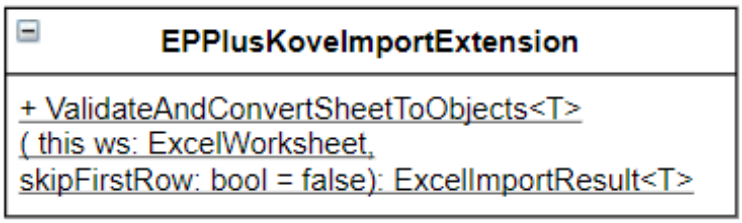
SonarQube haluttiin lisätä projektiin erityisesti näiden refaktorointia ja korjausvelan hallintaa tukevien ominaisuuksien takia. SonarQuben avulla refaktorointia jatketaan säännöllisesti kehityksessä ja syntyvän korjausvelan määrä voidaan tulevaisuudessa minimoida. Tämän projektin tavoitteena on asentaa SonarQube Koven jatkuvan integraation putkeen ja käyttöönottaa se. Projektin aikana SonarQubesta on tarkoitus saada ensimmäinen laaturaportti, jonka pohjalta refaktorointia tullaan tulevaisuudessa jatkamaan.

5 Työn toteutus

Kehitysjonon jalostamisen yhteydessä projektin työtehtävät kirjattiin Jiran Kanban-tauluun. Tämän taulun avulla seurattiin tehtävien etenemistä. Työhön käytetyt tunnit kirjattiin myös suoraan taulun työtehtävätiketeille. Aluksi työn alle otettiin rakennusten priorisoinnin kehitys sekä omien priorisointikategorioiden tuonti.

Priorisointi ja sen asetukset siirrettiin omalle välilehdelleen ja rakennusten priorisointi toteutettiin samalla tavoin kuin infraomaisuuksien priorisointi, mutta omilla oletuskategorioilla ja -pisteillä. Rakennusten priorisointi haluttiin erilleen infraomaisuuksien priorisoinnista, joten ne jaettiin vielä omiksi välilehdikseen. Rakennusten priorisoinnin uudelleenlaskenta eristettiin myös infraomaisuuksien uudelleenlaskennasta, jolloin rakennustietoja ei tarvitse ladata aina, kun infraomaisuuksia muokataan ja toisinpäin. Rakennusten priorisoinnin yhteydessä ei tarvittu suurempaa refaktorointia, pieniä nimien muutoksia vain. Kaikkea priorisointiin liittyvää vanhaa koodia pyrittiin hyödyntämään.

Omien priorisointityyppien tuonnin kehityksessä pyrittiin ensin katsomaan vanhaa koodia kriittisesti. Tuonnin tiedettiin jo valmiiksi olevan refaktoroinnin tarpeessa, joten aluksi muokattavaa koodia tutkittiin refaktoroinnin näkökulmasta. Tuonnissa Excel-rivit tarkistetaan ja muutetaan ohjelmiston hyväksymiksi olioiksi. Tämä validointi ja muunnos tapahtuu pääasiassa yhdessä luokassa EPPlusKoveImportExtension (kuva 13). Luokka sisältää yhden noin 170 riviä pitkän metodin ValidateAndConvertSheetToObjects, joka nimensä mukaan validoi tuodut tiedot ja muuttaa ne olioksi.



Kuva 13: EPPlusKoveImportExtension – vanha luokkakaavio

Tämä metodi on pitkä ja vaikeasti luettava myös monimutkaisen ehtolauserakenteensa vuoksi. Nämä ongelmat refaktoroitiin hyödyntäen metodin eristämistekniikkaa. Aluksi eh-

tojen sisällöt eristettiin omiksi metodeikseen ja ne nimettiin kuvaamaan metodin toimintaa. Metodien eristäminen yksinkertaisti ehtorakennetta, mutta luokka teki edelleen kahta eri asiaa: validointia ja olioiden luontia. Se siis rikkoi SOLID-käytäntöjen SRP-periaatetta. Tämän vuoksi validoinnista vastaavat metodit eristettiin vielä omaan luokkaansa Luokan eristäminen -refaktorointitekniikkaa hyödyntäen [4].

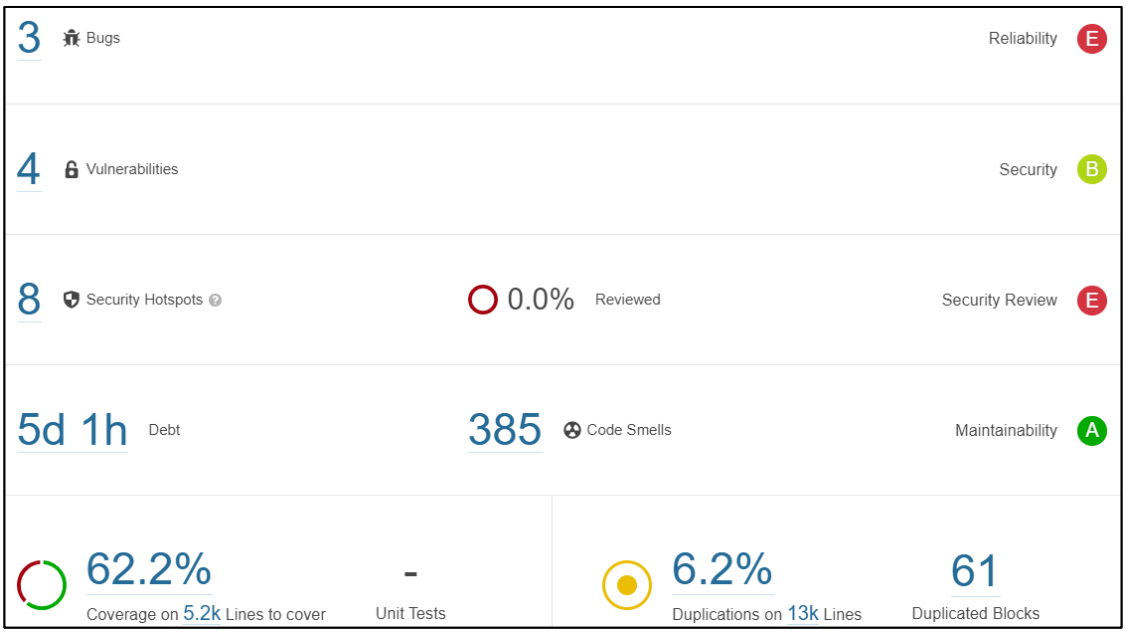
Alkuvalmistelujen jälkeen lähdettiin toteuttamaan omien priorisointityyppien tuontia. Ideana oli, että asiakas voisi tuoda Excelillä laskelmaan niin monta kategoriaa kuin haluaa ja nimetä ne haluamallaan tavalla. Ehtona oli, että rivin arvo kategorialle olisi numero väliltä 0–100. Omat kategoriat eivät ole pakollisia kenttiä. Ensin omille kategorioille lisättiin tietokannan taulut: CustomPriorizationMetric, johon tallennetaan dokumenttikohtaiset priorisointikategoriat, niiden painoarvot laskennassa ja omaisuuserän tyyppi, sekä AssetCustomMetric, johon tallennetaan asiakkaan yksittäiselle kohteelle antama lukuarvo. Omien tyyppien tuonti lisättiin vanhaan tuontiin niin, että tuonti-Excelistä poimittiin pakollisten kategorioiden jälkeen kaikki otsikkorivin täytetyt arvot. Sen jälkeen sarakkeiden arvot tarkistettiin olevan numeroita väliltä 0–100. Omien kategorioiden tuonnin tulos ja mahdolliset virheet lisättiin perusrivien CustomValues-listaan.

Uusi koodi käytti hyvin paljon samankaltaisia metodeja kuin aiemmin, mutta pienillä tyyppieroilla. Metodeja haluttiin yhdistää, jotta voitaisiin vähentää duplikaattikoodia. Tämä osoittautui kuitenkin hieman haastavaksi tehtäväksi. Excel-solujen validointi oli suhteellisen helppoa toteuttaa yhtenäisesti, mutta validoitava tieto oli eri olioissa tallennettu eri tavalla. Metodeihin oli annettava lopulta turhan pitkä lista muuttujia, mikä täytyy vielä jatkossa korjata. Kuitenkin lopulta päästiin tarpeeksi selkeään ja ainakin entistä parempaan lopputulokseen tuonnin kanssa.

Tuonnin tehtävien jälkeen siirryttiin viennin korjaukseen. Omat priorisointityypit piti pystyä viemään Exceliin, kuten muutkin kategoriat. Vientiä haluttiin parantaa erottamalla vienti omaisuuseräkohtaisesti. Viennin muokkaus aloitettiin poistamalla erillinen vientivälilehti ja siirtämällä Vie Exceliin -nappi jokaiselle omaisuuserän välilehdelle. Ennen muiden muutosten tekemistä viennin koodia refaktoroitiin samaan tapaan kuin tuonnissa. Liian pitkästä metodista eristettiin pienempiä metodeja. Metodit jaettiin omaisuus-

erittäin, tai niistä pyrittiin tekemään sellaisia, ettei ole väliä, mitä omaisuuserää käsitellään. Vienti oli helppoa toteuttaa vain halutulle omaisuuserälle, kun koodi oli aluksi siistitty. Lopuksi lisättiin omien priorisointityyppien vienti.

Viennin muutosten jälkeen kaikki Koveen halutut lisäominaisuudet oli toteutettu. Tämän jälkeen siirryttiin projektissa Koven ylläpitotehtäviin ja ensimmäisenä liitettiin SonarQube Koven integraatioputkeen. SonarQuben asennuksesta vastasi Rapalin DevOps-tiimi. Asennuksessa Kove projektin Solution ja Bitbucket liitettiin SonarQuben automaattiseen analyysiin niin, että SonarQube käy aina läpi Bitbuckettiin pusketun uuden koodin. Asennuksen yhteydessä pidettiin Foren kehitystiimin ja DevOps-tiimin välinen palaveri, jossa luotiin SonarQube-tunnukset kehittäjille ja varmistettiin ohjelman käyttöönotto. Käyttöönotto onnistui hyvin ja SonarQubesta saatiin ensimmäinen raportti ulos (kuva 14). Analyysistä näkee, että Koven koodi on vielä hallittavassa kunnossa, sillä SonarQuben antama Maintainability, eli ylläpidettävyysarvio on tasolla A. Koodista löytyy kuitenkin vielä teknistä velkaa (Debt), joka on korjattavissa noin viidessä päivässä, ja 385 hajua (Code Smells). Myös duplikaatioita on vielä paljon, jopa 6,2 % noin 13000 rivillä. Koven projektista ei tässä vaiheessa analysoitu kuin backend-koodi. Frontend-koodi on myöhemmin lisättävä analyysin piiriin.



Kuva 14: SonarQuben ensimmäinen Kove-analyysi

Viimeisenä ylläpitotehtävänä tuonnin virheilmoituksia haluttiin korjata. Virheilmoitukset toimivat halutulla tavalla rakennusten tuonnissa, joten lopulta kaikki tuonnit päätettiin toteuttaa yhtenäisellä tavalla. Tämä helpottaisi jatkossa myös ohjelman laajentamista. Tuonnin muutokset aloitettiin refaktoroinnin prosessin (kuva 1, s. 4) tavoin vanhan koodin parantamisella. ImportControllerien luokkakaaviot ennen refaktorointia ja sen jälkeen on esitelty tämän työn lopussa liitteissä 2 ja 3.

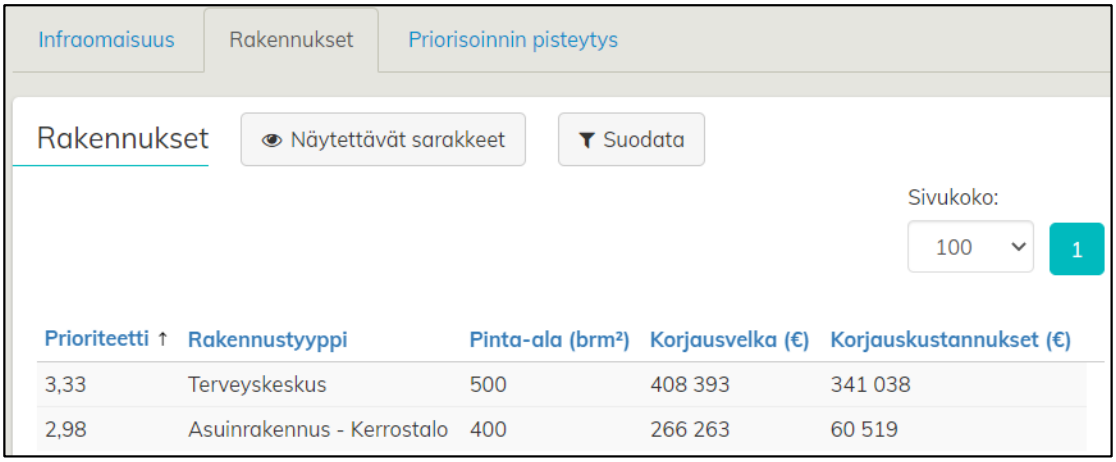
Aluksi ImportAssets metodi (liite 2, AssetImportController) pilkottiin pienemmiksi metodeiksi Metodin eristäminen -refaktorointitekniikalla [4]. Metodien oletustoteutukset olivat pääluokassa ja nyt vain tarvittavat osat, kuten BuildingImportControllerissa CreateAssets -metodi (liite 3), voitiin ylikirjoittaa aliluokissa. Rakennusten tuonti toteutettiin aluksi järjestyksessä: hae hinnat – luo DTO:t – luo omaisuuserät, ja omaisuuserien luonnissa hinnat haettiin vielä uudestaan. Infraomaisuuksien tuonnissa järjestys oli selkeämpi: luo DTO:t – hae hinnat – luo omaisuuserät. Tässä tavassa ei tarvittu hintojen tuplahakua. Infraomaisuuksien virheilmoitusongelma korjattiin ja rakennusten tuontijärjestys muutettiin vastaamaan infraomaisuuksien tuontia.

Rakennusten tuonnin järjestyksen muuttaminen vaikutti myös PriceServicen muokkaustarpeeseen. Kuten PriceServicen alkuperäisen toteutuksen luokkakaavion korostetuista kohdista (liite 1) näkee, PriceServicet sisälsivät paljon duplikaattimetodeja, etenkin rakennusten puolella. Tämä johtui rakennusten hinnastojen hausta, joka tehtiin sekä tuontiriveille että niistä luoduilla DTO:illa erikseen. Järjestyksen muutos teki monista metodeista turhia ja duplikaatteja pystyttiin poistamaan. Monet metodit pystyttiin toteuttamaan PriceService-yläluokassa pienillä tyyppikikkailuilla. PriceService-rakennetta saatiin selkeämmäksi ja laajennettavammaksi kuin ennen. Muokkaukset saatiin tehtyä muiden ylläpitotehtävien ohella, eikä PriceServicen refaktoroinnille tarvinnut varata erillistä aikaa, kuten aluksi oli suunniteltu. Refaktoroinnissa pystyttiin hyödyntämään juuri asennettua SonarQubea. Sen avulla huomattiin nopeasti refaktoroinnin tarpeet muokattavassa koodissa sekä muokkauksissa.

Järjestyksen muuttaminen aiheutti kuitenkin ongelmia rakennusten kuntotason laskennassa. Rakennusten kuntotason laskennassa on erikoinen muna–kana-ongelma, kun kunto tarvitsisi hintoja ja hinnat kuntoa. Kuntotason laskenta tullaan kuitenkin myöhemmin muuttamaan niin, ettei se tarvitse hintaa.

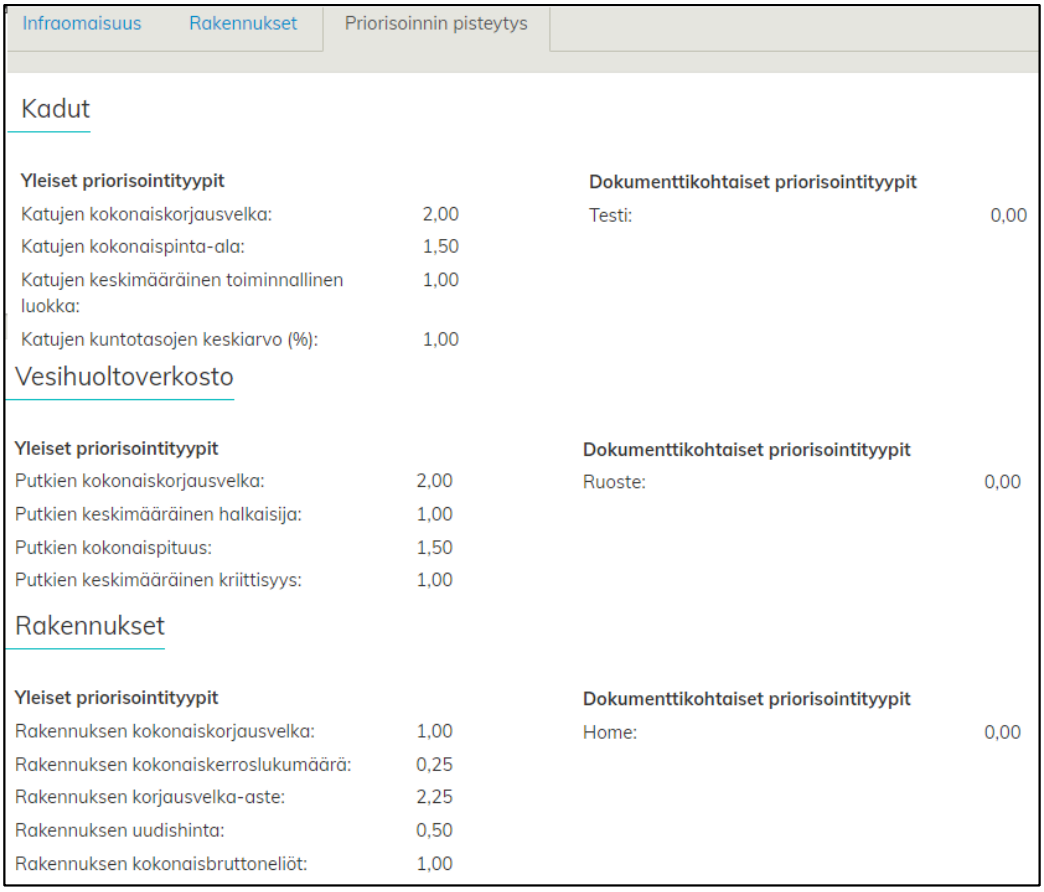
6 Työn tulokset ja jatkokehitys

Koven jatkokehitysprojekti oli kokonaisuudessaan erittäin onnistunut. Halutut ominaisuudet saatiin lisättyä ja ohjelmiston pahimpia refaktoroinnin kohteita selvitettyä jopa odotettua paremmin. Tärkeimpinä uusina ominaisuuksina toteutettiin rakennusten priorisointi ja omien priorisointityyppien tuonti. Priorisointia selkeytettiin myös huomattavasti siirtämällä se ja sen asetukset omalle välilehdelle. Kuvassa 15 on esitelty rakennusten priorisointia.



Kuva 15: Rakennusten priorisointi

Rakennusten priorisointi on kuvan mukaisesti erillään infraomaisuuksien priorisoinnista, ja se päivitetään, kun rakennustietoja muokataan tai tuodaan dokumentille. Huomion kiinnittämiseksi olennaiseen priorisoinnin pisteet on siirretty taulukon ensimmäiseksi sarakkeeksi ja kohteet järjestetään automaattisesti prioriteettipisteiden mukaan. Myös priorisoinnin pisteytys on jaettu omaisuuserittäin, kuten kuvasta 16 näkee.



Kuva 16: Priorisoinnin uusi pisteytys omaisuuserittäin

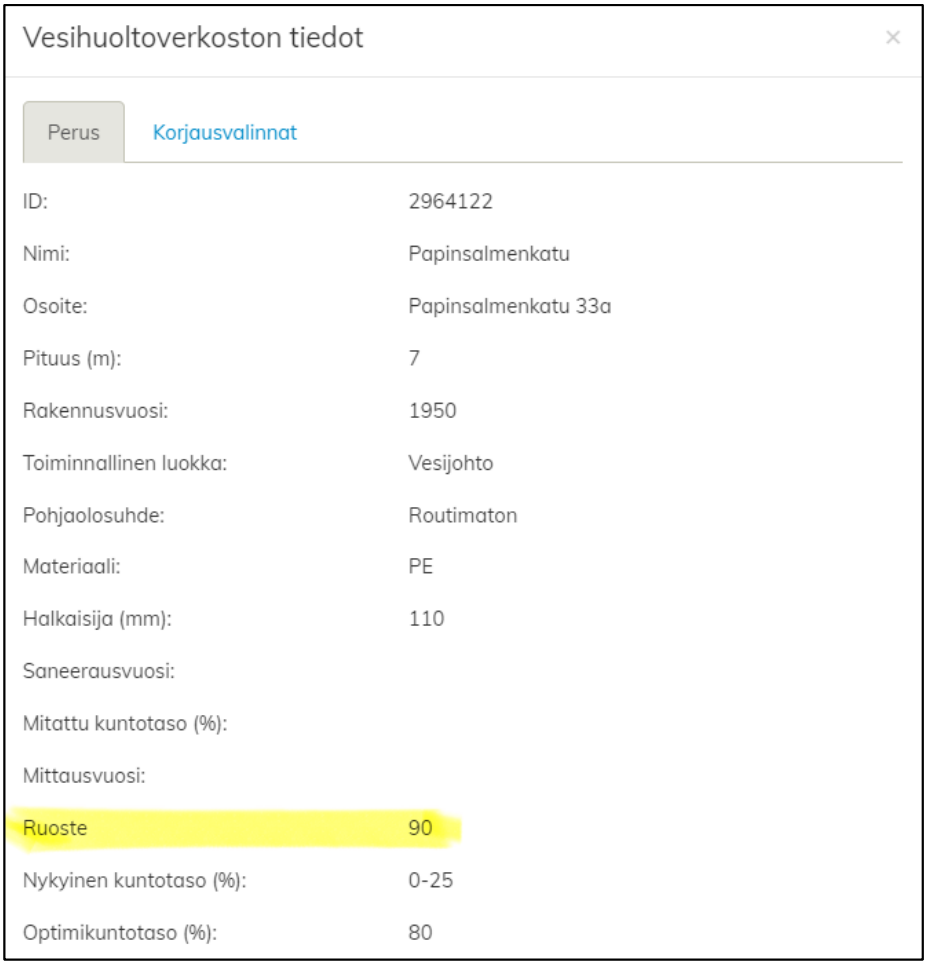
Priorisoinnin oletuspisteitä voidaan nyt muokata omaisuuseräkohtaisesti. Samalla voidaan muokata omaisuuserän dokumenttikohtaisia omia priorisointityyppejä, jos käyttäjä on niitä tuonut. Omien priorisointityyppien oletuspistemäärä on 0, joten ne eivät automaattisesti vaikuta priorisoinnin laskentaan. Käyttäjä voi muokata omien priorisointityyppien oletuspisteitä Priorisoinnin pisteytys -sivulla. Tässä kategoriassa suurimman arvon saava kohde saa asetetut maksimipisteet, ja muut skaalautuvasti siitä alaspäin. Omien kategorioiden pisteet summataan oletuskategorioiden avulla laskettuihin pisteisiin.

Omat kategoriat tuodaan tuonti-Excelissä viimeisissä sarakkeissa. Omille kategorioille voi antaa vapaavalintaisen nimen ja kohteen kategoriakohtaiseksi arvoksi voi antaa desimaaliluvun väliltä 0–100. Oman kategorian arvon voi jättää rivikohtaisesti tyhjäksi, tai niitä ei ole pakko tuoda laskelmalle ollenkaan. Kuvassa 17 on esitelty, miten käyttäjälle on ohjeistettu omien priorisointityyppien tuonti.

Sarake	Otsikko	Pakollinen	Tyyppi
A	ID	Ei	Teksti
B	Nimi	Ei	Teksti
C	Osoite	Kyllä	Teksti
D	Pinta-ala (m²)	Kyllä	Numero
E	Rakennusvuosi	Kyllä	Teksti
F	Toiminnallinen luokka	Kyllä	Teksti
G	Pohjaolosuhde	Kyllä	Teksti
H	Katuluokka	Kyllä	Numero
I	Kulutuskerros	Kyllä	Teksti
J	Saneerausvuosi	Ei	Numero
K	Mitattu kuntotaso (%)	Jos mittausvuosi on annettu	Numero
L	Mittausvuosi	Jos mitattu kuntotaso on annettu	Numero
M -	***Oma otsikko***	**Ei**	**Numero (0-100)**

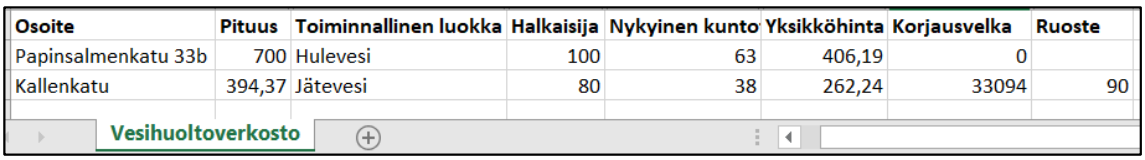
Kuva 17: Omien kategorioiden tuonnin ohjeistus

Omat priorisointityypit eivät suoraan näy omaisuuserätaulukossa (kuva 6, s. 16), sillä käyttäjän on mahdollista tuoda omaisuuserille haluamansa määrä omia priorisointityyppejä, ja tämä voisi venyttää taulukkoa tarpeettoman pitkäksi. Omaisuuseräkohtaisia omien tyyppien arvoja voidaan tarkastella ja muokata rivien lopusta löytyvästä Näytä lisätietoja -kohdasta. Tästä painikkeesta aukeaa kuvan 18 mukainen moduuli, johon kaikki omaisuuserätyypin omat kategoriat listataan mittausvuoden jälkeen.



Kuva 18: Omien kategorioiden (Ruoste) näkyminen

Omat kategoriat näkyvät myös vienti-Excelissä. Vienti siirrettiin omaisuuseräkohtaiseksi oman välilehden sijaan. Nyt käyttäjät voivat viedä vain haluamansa omaisuuserän tiedot Exceliin, eikä dokumentille tulostu tyhjiä välilehtiä kuten ennen. Omat kategoriat listautuvat vienti-Excelin loppuun, jos niitä on dokumentille tuotu, kuten kuvassa 19.

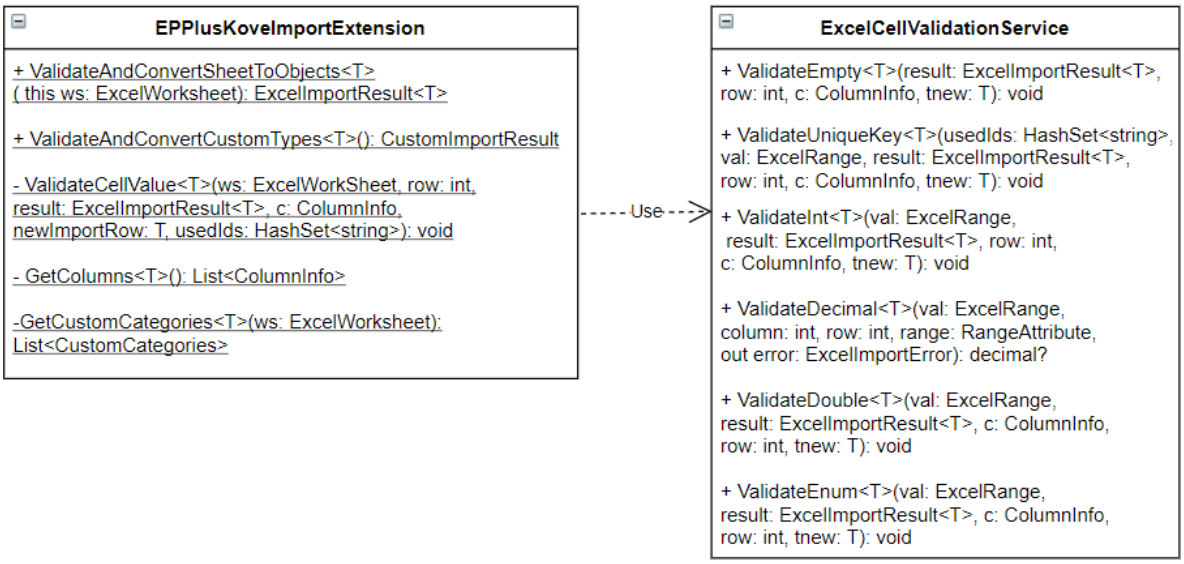


Osoite	Pituus	Toiminnallinen luokka	Halkaisija	Nykyinen kunto	Yksikköhinta	Korjausvelka	Ruoste
Papinsalmenkatu 33b	700	Hulevesi	100	63	406,19	0	
Kallenkatu	394,37	Jätevesi	80	38	262,24	33094	90

Vesihuoltoverkosto

Kuva 19: Kovesta viety Excel-tiedosto ja omat kategoriat (Ruoste)

Uusia ominaisuuksia lisätessä saatiin myös hyvin vähennettyä ohjelman korjausvelkaa vanhaa koodia refaktoroimalla. Tärkeimmät refaktoroinnin kohteet olivat tuonti, vienti ja PriceServicet. Tuontia pystyttiin yhtenäistämään eri omaisuuserätyyppien välillä ja parantamaan huomattavasti sen luettavuutta ja laajennettavuutta. Kuvassa 20 on esitelty tuonnille olennaisen EPPlusKoveImportExtension uusi luokkakaavio, jota voidaan verrata vanhaan luokkakaavioon (kuva 13, s. 27).

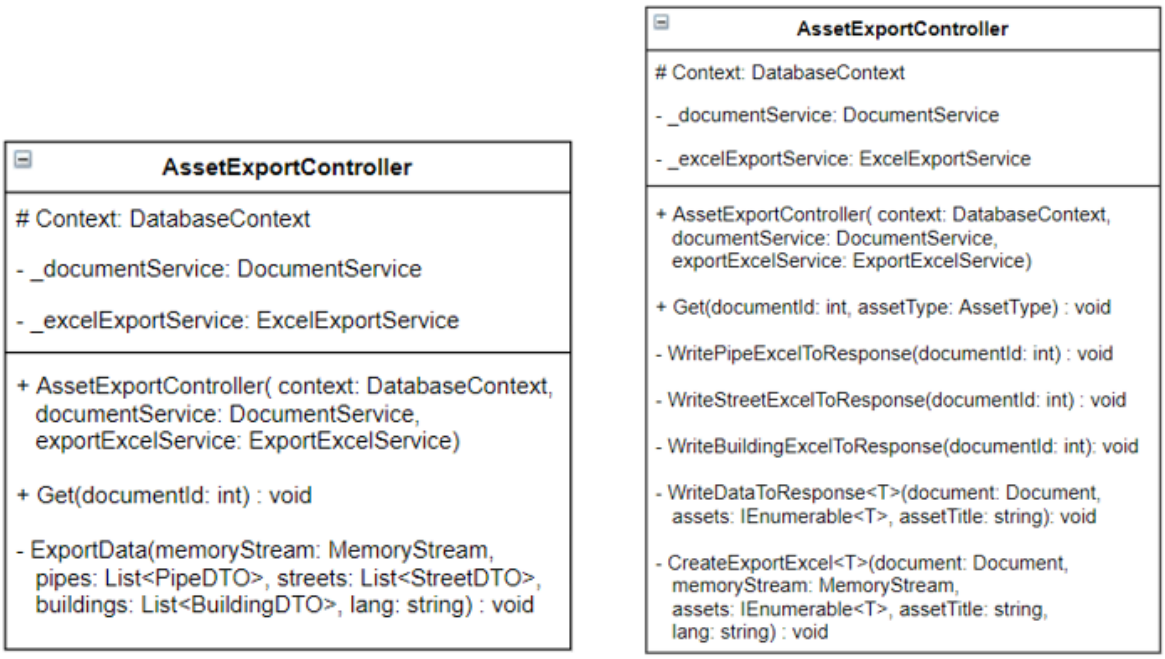


Kuva 20: EPPlusKoveImportExtension – uusi luokkakaavio

Luokka koostui aiemmin yhdestä ainoasta metodista, joka hoiti sekä tuotujen Excel-solujen tarkistuksen, että tuotujen tietojen muuttamisen Koven käsittelemiksi ImportRow-olioiksi. Tämä valtavan pitkä metodi pilkottiin pienemmiksi metodeiksi, jotka pyrittiin toteuttamaan mahdollisimman geneerisiksi, jotta niitä voitaisiin käyttää uudelleen mahdollisimman tehokkaasti. Samalla solujen validoinnista vastaavat metodit eristettiin omaksi luokakseen ExcelCellValidationService (kuva 20). Nämä muutokset lyhensivät alkuperäistä koodia huomattavasti, ja nyt koodi on helpommin tulkittavissa. Geneeriset metodit vahvistavat myös ohjelmiston laajentamista.

Tuonnin tavoin viennissäkin paranneltiin koodin luettavuutta ja laajennettavuutta. Vienti muutettiin toimimaan omaisuuserä kerrallaan ja alkuperäinen koodi pystyttiin pilkkomaan pienemmiksi metodeiksi. Itse viennistä vastaavista metodeista pystyttiin tekemään täysin

omaisuuserätyypistä riippumattomia, mikä helpottaa tulevaisuudessa viennin kehitystä ja laajentamista. Kuvassa 21 on esitelty viennin alkuperäinen ja uusi luokkakaavio.



Kuva 21: AssetExportController – vanha luokkakaavio (vas.) ja uusi luokkakaavio (oik.)

Viimeisenä ja laajimpana refaktoroinnin kohteena olivat Koven PriceServicet. Koodi sisälsi paljon duplikaatioita, ja jotkin perityt metodit olivat aliluokissa täysin turhia. Nämä ongelmat saatiin ratkottua tuonnin järjestyksen muuttamisen yhteydessä. Monia turhia metodeja pystyttiin poistamaan ja PriceServicien koodista saatiin paljon luettavampaa. Isompia metodeja pilkkomalla voitiin useampia osia toteuttaa vain yläluokassa ja vain tarvittavat metodit ylikirjoitettiin aliluokissa. Liitteessä 4 on esitelty PriceServicien uusi luokkakaavio, jota voi verrata vanhaan kaavioon liitteessä 1. Tärkeimpänä muutoksena turhat BulkFetchPrices-metodit voitiin poistaa ja toteuttaa vain yläluokassa.

PriceServicen laajennettavuus parani näillä muutoksilla huomattavasti, mutta ohjelmaan jäi vielä korjausvelkaa. Rakennusten kuntotason laskentaa pitää vielä korjata ja sen jälkeen PriceServicet voidaan vasta saada lopullisesti korjattua. PriceServicen refaktorointi

sujui kuitenkin odotettua paremmin, ja se pystyttiin toteuttamaan refaktoroinnin prosessin mukaisesti eikä erillään muusta kehityksestä kuten aluksi oli suunniteltu. Refaktoroinnin tulokset kannustavat jatkamaan hyväksi todettua prosessia.

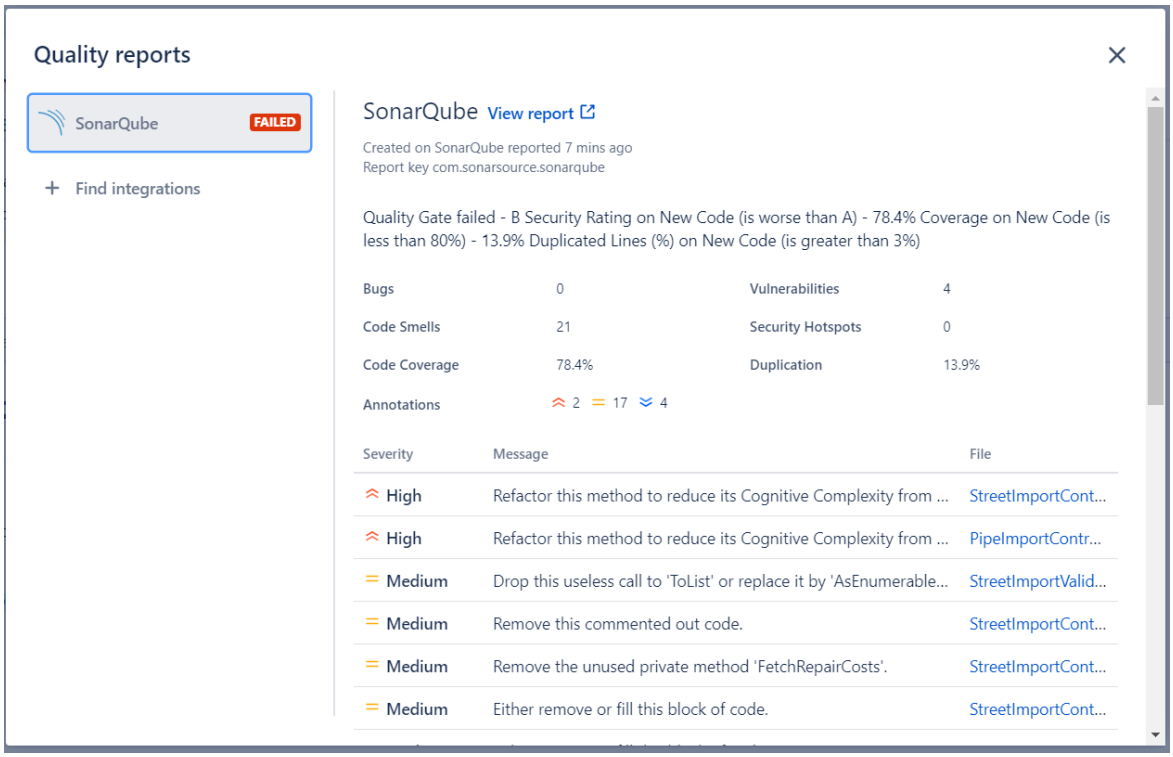
Ongelmitta ei kuitenkaan tässä projektissa selvitty. Ohjelma on laaja, ja vaikkei se olekaan vanhimmasta päästä, oli siihen kertynyt jo jonkin verran teknistä velkaa. Aikatauluja venyttivät vaikeasti hahmotettavat rakenteet, joita oli haastavaa lähteä refaktoroimaan. Samankaltaisia ongelmia duplikaatioiden kanssa oli pitkin koodia ja tarpeeksi geneeristen toteutusten tekeminen oli haasteellista. Tätä ratkottiin hiljalleen eristämällä yhtenäisiä osia omiksi metodeiksi kohteiden yläluokkiin ja jättämällä vain eroavat osat alaluokan hoidettavaksi.

Koodin huono testikattavuus oli myös pienoinen murheenkryyni. Testaamattoman koodin muokkaus altistaa aina isommille riskeille, vaikka kyseessä olisi vain refaktorointia. Julkaisupaineen alla huomasi tekevänsä huonompia ratkaisuja kuin oli suunnitellut, mutta ongelmat vain korostivat jatkuvan refaktoroinnin merkitystä ja automaattisten työkalujen tarvetta.

Projektin aikana Koven kehityksessä saatiinkin otettua käyttöön automaattinen koodin analysointityökalu SonarQube. SonarQube yhdistettiin osaksi projektin jatkuvan integraation putkea Bitbucketin ja TeamCityn ohelle. SonarQuben käyttöönotto oli hyvin onnistunut, ja Kovesta saatiinkin ulos ensimmäinen laaturaportti (kuva 14, s. 29). Sen mukaan Kove on korjausvelallisesti vielä hyvin hallittavalla tasolla, mutta hajuja silti löytyy. Koven pahimpia ongelmia on duplikaattikoodin määrä ja huono testikattavuus. Korjausvelkaa ja testikattavuutta tullaan korjaamaan refaktoroinnin prosessin mukaisesti muiden tehtävien ohella. SonarQube antaa myös mahdollisuuden korjata kaikki ongelmat kerralla workshop-tyyppisesti. Toimintasuunnitelmaa tullaan vielä päivittämään, kun myös frontend-koodi saadaan SonarQube-analyysin piiriin.

Alkuraportin lisäksi SonarQubesta saatiin myös uusiin koodimuutoksiin liittyvää analyysiä. Bitbuckettiin yhdistettynä SonarQube analysoi jokaisen uuden koodimuutoksen ja tekee siitä muutoskohtaisen raportin (kuva 22). Raportti kertoo heti uuden koodin laadusta ja antaa vinkkejä mahdollisiin korjauksiin. Raportti myös ilmoittaa, onko koodi tarpeeksi laadukasta yhdistettäväksi alkuperäiseen kehityshaaran. Esimerkiksi kuvan 22

koodi ei pääse laadunvarmistuksesta läpi ja raportissa näkyy punaisella Failed. Raportti kertoo, ettei koodi läpäise laatuvaatimuksia, koska turvaluokitus on huonompi kuin A, testikattavuus on alle 80 % ja duplikaatioita on yli 3 %. Tarkastuksen rajoja on mahdollisuus säätää SonarQubessa. Tämä ominaisuus halutaan ottaa myös Koven kehitystiimissä käyttöön, kunhan pahimmat laatuongelmat on ensin saatu korjattua.



Kuva 22: SonarQuben Bitbucket-raportti

Projektin suurimpana antina oli tutustuminen tarkemmin refaktoroinnin tekniikoihin ja prosessiin. Pienen ja suhteellisen uuden kehitystiimin on heti alussa hyvä omaksua oikeita käytäntöjä, ja koodin parannuksessa refaktorointi on erittäin hyödyllinen taito. Refaktoroinnin tueksi saatiin myös automaattisia työkaluja. Refaktoroinnin suhteen oli aluksi samoja ennakko-oletuksia kuin Microsoftin tutkimuksessa [5] oli havaittu. Suurten rakennemuutosten pelättiin aluksi aiheuttavan virheitä, tai koontikatkoja varsinkin, koska testikattavuus on Kovella melko huono. Tämä myös aiheutti huolta siitä, riittävätkö tiimin testausresurssit refaktorointien tulosten tarkistamiseen.

Ennakkoluulot kuitenkin haihtuivat projektin edetessä ja tiedon lisääntyessä. Kun oppi, mitä refaktorointi oikeasti tarkoittaa ja millainen sen prosessi on, ymmärsi refaktoroinnista olevan huomattavasti enemmän hyötyä kuin haittaa. Koodista tuli paljon luettavampaa ja hallittavampaa, kun metodit ja luokat jakoi pienemmiksi. Pienemmät metodit helpottivat myös koodin testausta. Toimimalla prosessin mukaan refaktoroimalla koodia ennen uuden lisäämistä huomasi, kuinka paljon refaktorointi voi helpottaa uusien ominaisuuksien tekoa. Työssä koetut refaktoroinnin hyödyt olivat hyvin samankaltaisia kuin Microsoftin tutkimuksessa [5]. Jo näiden hyötyjen perusteella refaktorointia aiotaan jatkaa Koven sekä muiden Fore-tuotteiden kehityksessä myös tulevaisuudessa.

Refaktoroinnin prosessi on hyvä, ja sitä halutaan jatkaa. Prosessia täytyy vielä kehittää ottamalla erityisesti yksikkötestaus huomioon. Kuten todettu testaamattoman koodin refaktorointi on riskialttiimpaa. Siksi jatkossa vanhoja toimintoja muokatessa olisi hyvä refaktoroida myös vanhoja testejä, tai lisätä puuttuvat testit kokonaan itse.

SonarQuben analyysi koskee tällä hetkellä vain Koven backend-koodia. Jatkossa analyysin piiriin pitää lisätä myös frontend. Käyttöliittymää tullaan jonkin verran päivittämään yksittäisten rivien lisäämisominaisuutta tehtäessä. Samalla olisi hyvä refaktoroida sen koodia ja testejä. Koveen tullaan myöhemmin lisäämään myös uusia ominaisuuksia, mutta tämän projektin ansiosta niiden lisääminen tulee olemaan varmasti helpompaa.

7 Yhteenveto

Tämän insinöörityön tavoitteena oli toteuttaa toivotut uudet ominaisuudet Koven priorisointiin ja korjata vanhoja virheitä ohjelmistossa. Nämä jatkokehitystoimet oli tarkoitus toteuttaa niin, että samalla voitaisiin vähentää ohjelmistoon kertynyttä teknistä velkaa ja parantaa ohjelmiston rakennetta. Ohjelmiston tiedettiin olevan vaikealukuinen ja huonosti laajennettavissa huonojen rakenneratkaisujen vuoksi. Ohjelmiston laatua haluttiin parantaa käyttämällä Fowlerin esittelemiä refaktoroinnin tekniikoita ja prosessia [4]. Refaktoroinnista haluttiin tehdä tiimissä mahdollisimman pysyvä prosessi ja tämän tueksi Koveen haluttiin projektin aikana asentaa automaattinen koodin laadun analysointityökalu SonarQube.

Suurin osa projektin tavoitteista saavutettiin erittäin onnistuneesti, jopa paremmin kuin aluksi oli ajateltu. Kaikki uudet ominaisuudet toteutettiin ja bugit korjattiin. Samalla refaktorointia hyödyntäen pystyttiin parantamaan koodin rakennetta ja tekemään ohjelmistosta helpommin luettavaa ja ylläpidettävää sekä laajennettavaa. Erityisesti koodin pahimpia ongelmia tuonnin kontrollereissa ja PriceServiceissä saatiin selvitettyä. Rakennusten laskentajärjestystä koskevat muutokset aiheuttivat kuitenkin ongelmia kuntotason laskennan kanssa, mutta tämä kuntotason laskenta tullaan pian uudistamaan joka tapauksessa, joten asian annettiin olla.

Työn aikana haasteita aiheutti etenkin Koven huono testikattavuus, jonka vuoksi pienistäkin muutoksista johtuvat virheet huomattiin vasta paljon myöhemmin taskin jo edettyä testaajalle. Myös refaktoroinnista saattoi aiheutua pieniä virheitä, jotka kunnon testeillä olisi löytänyt paljon nopeammin. Testien kirjoitus on ehdottomasti otettava tiukasti mukaan tiimin kehitysprosessiin ja vanhoja toimintoja muokatessa olisi hyvä myös pyrkiä refaktoroimaan niiden testejä.

Toinen haaste oli pienistä tyyppikohtaisista eroista johtuvien duplikaattikoodien vähentäminen. Tyyppikohtaisia toteutuksia pyrittiin purkamaan eristämällä samanlaisia osia kohteiden yläluokille ja jättämällä vain eroavaisuudet alaluokan hoidettavaksi. Tämä oli käytännössä kuitenkin melko haasteellista, mutta kuitenkin toteutettavissa. Julkaisupaineen alla ei kuitenkaan aina päästy niin siisteihin lopputuloksiin kuin olisi haluttu, mutta koodi on kuitenkin nyt paljon parempi, kuin mitä se aiemmin oli.

Refaktoroinnin prosessinomainen käyttö tuotti hyviä tuloksia ja oli lopulta helposti toteutettavissa. Prosessia on nyt myös tukemassa SonarQube, joka varmistaa vähintään sen, että refaktorointia kaipaavat kohteet tullaan jatkossa huomaamaan. Tavoitteena on saada korjattua SonarQuben esittämät laatupoikkeamat muun jatkokehityksen ohella, tai workshop-tyylillä kerralla esimerkiksi kehitysjakson aikana, jolloin uusia ominaisuuksia ei lisätä (feature freeze). Myöhemmin SonarQuben halutaan toimivan koodimuutosten laadunvarmistajana: uutta koodia ei tulla yhdistämään kehityshaaraan, ennen kuin SonarQube-analyysi antaa halutun tuloksen.

Projekti oli loppujen lopuksi erittäin onnistunut ja tuotti paljon hyötyä myös tekijälleen. Projektin aikana oppi paljon lisää refaktoroinnista ja kuinka sitä kannattaa hyödyntää. Oikeanlainen kehitysprosessi tulee jatkossa varmasti vähentämään kehitykseen käytettyä aikaa ja helpottamaan kommunikaatiota tiimin sisällä.

Lähteet

1 Rakennusteollisuus. Korjausvelka. Verkkoaineisto. < https://www.rakennusteollisuus.fi/Tietoa-alasta/Korjausrakentaminen1/Korjausvelka/>. Luettu 8.11.2020.

2 Niemistö, T. 2016. Tekninen velka – yrityksesi suurin digitalisaation jarru? Verkkoaineisto. Kauppalehti. < https://blog.kauppalehti.fi/digiarjessa/tekninen-velka-yrityksesi-suurin-digitalisaation-jarru>. Päivitetty 27.12.2016. Luettu 8.11.2020.

3 Rapal. Meistä. Verkkoaineisto. <https://www.rapal.com/fi/meista>. Luettu 25.9.2020.

4 Fowler, M. 2018. Refactoring: Improving the Design of Existing Code. E-kirja. Addison-Wesley Professional.

5 Kim, M & Nagappan, N & Zimmermann, T. 2012. A Field Study of Refactoring Challenges and Benefits. The University of Texas, Microsoft Research. ResearchGate. Luettu 16.11.2020.

6 Leitch, R & Stroulia, E. 2003. Understanding the Economics of Refactoring. ResearchGate. Luettu 13.11.2020.

7 Fayolle, J. 2014. Legacy Application – Technical debt and ROI of a refactoring. Verkkoaineisto. < http://qualilogy.com/en/legacy-application-technical-debt-roi-refactoring/>. Luettu 13.11.2020.

8 Morris, B. 2012. Why refactoring code is almost always better than rewriting it. Verkkoaineisto. < https://www.ben-morris.com/why-refactoring-code-is-almost-always-better-than-rewriting-it/>. Luettu 27.11.2020.

9 Simon LH. 2019. SOLID Principles: Explanation and examples. Verkkoaineisto. < https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4>. Luettu 2.12.2020.

10 Rapal. Korjausvelan laskenta. Verkkoaineisto. <https://www.rapal.com/fi/korjausvelan-laskenta>. Luettu 25.9.2020.

11 React. React – A JavaScript library for building user interfaces. Verkkoaineisto. < https://reactjs.org/>. Luettu 7.11.2020.

12 TutorialsTeacher. .NET Core Overview. Verkkoaineisto. < https://www.tutorialsteacher.com/core/dotnet-core >. Luettu 7.11.2020.

13 TutorialsTeacher. ASP.NET Core Overview. Verkkoaineisto. < https://www.tutorialsteacher.com/core/aspnet-core-introduction >. Luettu 7.11.2020.

14 Entity Framework Tutorial. What is Entity Framework? Verkkoaineisto. <https://www.entityframeworktutorial.net/what-is-entityframework.aspx>. Luettu 7.11.2020.

15 Microsoft. What is .NET Standard? Verkkoaineisto. < https://dotnet.microsoft.com/platform/dotnet-standard>. Luettu 8.11.2020.

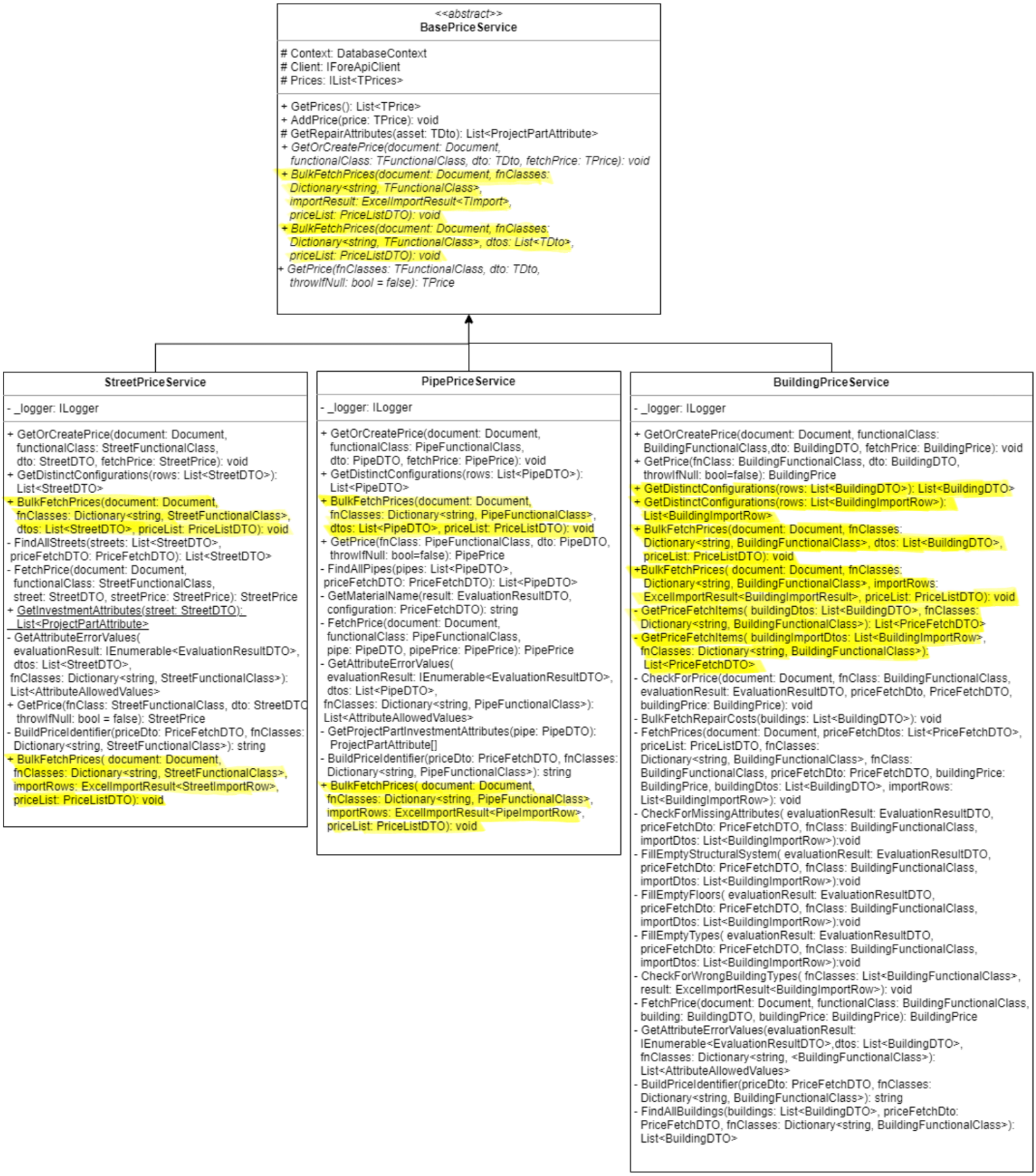
16 Atlassian. The #1 software development tool used by agile teams. Verkkoaineisto. < https://www.atlassian.com/software/jira>. Luettu 20.11.2020

17 Atlassian Bitbucket. Version control software for professional teams. Verkkoaineisto. < https://bitbucket.org/product/version-control-software>. Luettu 20.11.2020.

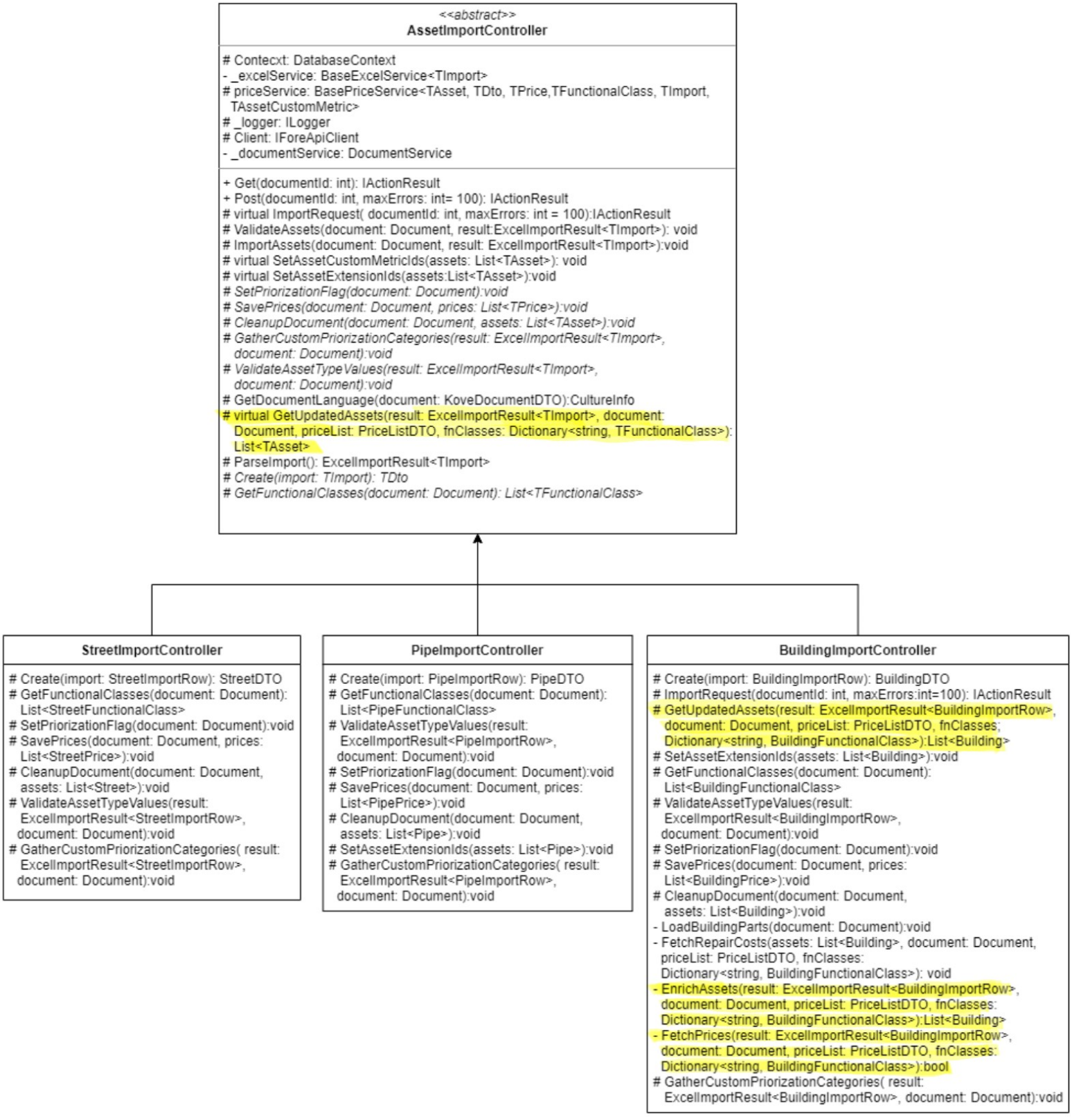
18 JetBrains. TeamCity. Verkkoaineisto. <https://www.jetbrains.com/teamcity/ >. Luettu 20.11.2020.

19 SonarQube. Your teammate for Code Quality and Security. Verkkoaineisto. <https://www.sonarqube.org/>. Luettu 11.12.2020.

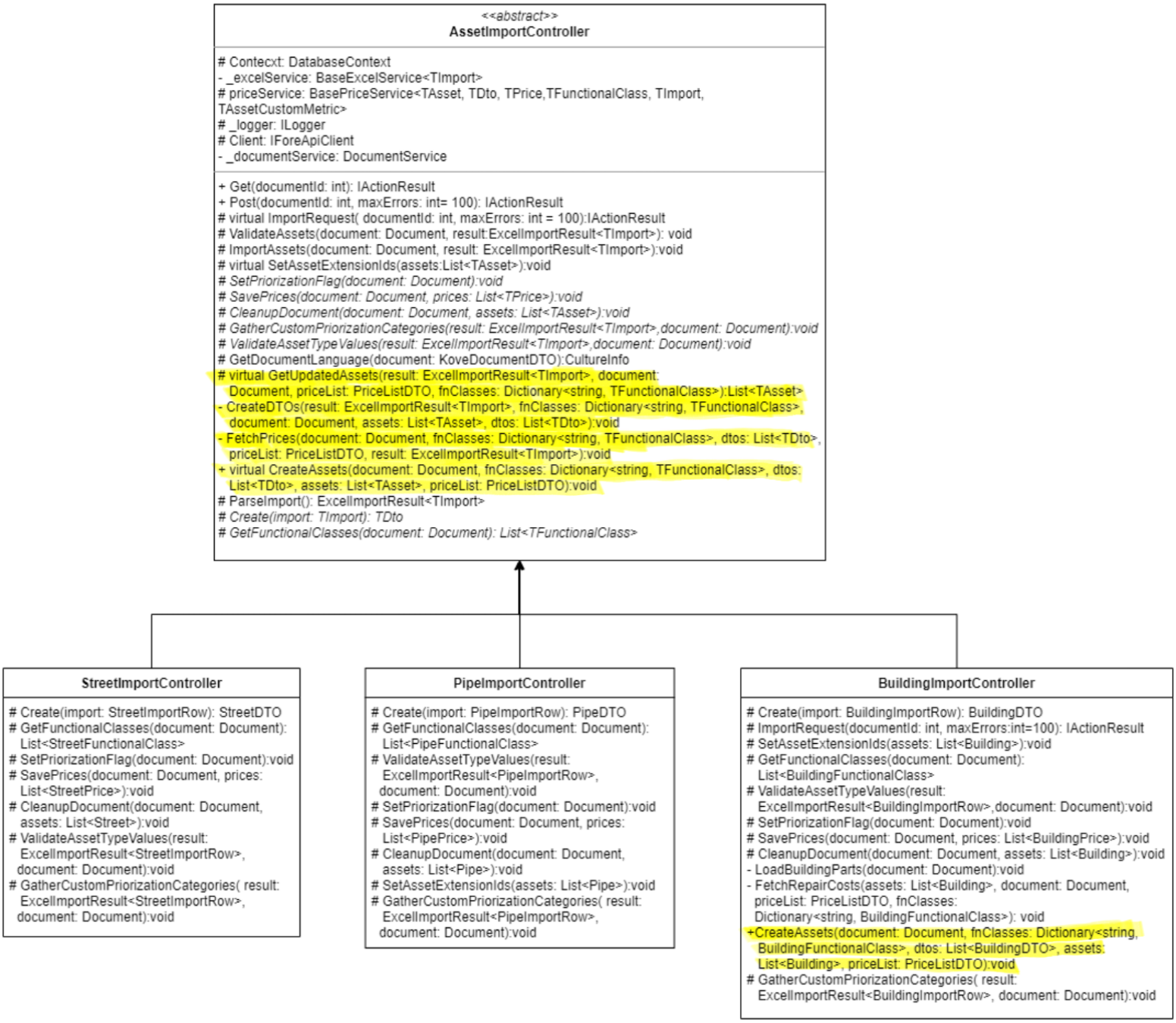
PriceService – vanha luokkakaavio



ImportController – vanha luokkakaavio



ImportController – uusi luokkakaavio



PriceService – uusi luokkakaavio