Tampereen yliopisto

Joni Saarinen

LÄHDEKOODIN UUDELLEENKÄYTTÖ LASKUTUS- JA PERINTÄJÄRJESTELMÄSSÄ

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Kesäkuu 2021

TIIVISTELMÄ

Joni Saarinen: Lähdekoodin uudelleenkäyttö laskutus- ja perintäjärjestelmässä
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-tutkinto-ohjelma
Kesäkuu 2021

Ohjelmistojen uudelleenkäyttö on aihe, jota on tutkittu jo kymmenien vuosien ajan. Hyvin tehtynä uudelleenkäytön hyödyt ovat selvät. Aikaa säästyy ja ohjelmistojen laatu nousee. Hyötyjen saavuttamiseksi uudelleenkäyttö pitää kuitenkin toteuttaa järkevällä tavalla.

Tässä työssä tutkitaan miten lähdekoodia kannattaa uudelleenkäyttää valmiista järjestelmästä uuden moduulin teossa ajankäytön, ylläpidettävyyden ja virheherkkyyden suhteen. Tavoitteena on hyötyä uudelleenkäytön hyvistä puolista ja minimoida riskit, jotka liittyvät tuotantokäytössä olevan lähdekoodin muokkaamiseen. Tätä varten työssä tutustutaan ohjelmistojen uudelleenkäytön näkökulmiin sekä komponenttipohjaiseen kehitykseen, joka on yleisesti käytetty lähestymistapa uudelleenkäytettävien ohjelmistojen tuottamiseen.

Työssä esitellään ostolaskumoduulin suunnittelu ja toteutus laskutus- ja perintäjärjestelmään. Suunnittelu aloitettiin vertailemalla kahta eri vaihtoehtoa siitä, miten lähdekoodia voidaan uudelleenkäyttää valmiista ja käytössä olevasta myyntilaskutoteutuksesta. Lähdekoodia voitiin joko refaktoroida tai sitä voitiin hyödyntää kopioimalla ja muokkaamalla. Lähdekoodia päädyttiin refaktoroimaan. Myyntilaskutoteutuksesta erotettiin ja muokattiin yleiskäyttöinen laskukomponentti. Komponenttia hyödynnettiin ostolaskumoduulin toteutuksessa.

Valitun toteutustavan laskettiin hidastaneen moduulin valmistumista 80 tunnilla. Yhteensä moduulin toteutukseen käytettiin 450 tuntia. Toteutusvaiheessa käytetyn ylimääräisen ajan pitäisi kuitenkin näkyä jatkossa nopeutuneena ylläpitona ja jatkokehityksenä. Lisäksi jo tuotantokäytössä olleen myyntilaskutoteutuksen laatua saatiin samalla kasvatettua. Lähdekoodin refaktorointi ei myöskään aiheuttanut virheitä jo käytössä olleisiin myyntilaskuominaisuuksiin, vaikka se nähtiin isona riskinä.

Avainsanat: koodin uudelleenkäyttö, ohjelmistojen uudelleenkäyttö, komponenttipohjainen kehitys

ABSTRACT

Joni Saarinen: Source code reuse in invoicing and collection software
Master of Science Thesis
Tampere University
Degree Programme in Information Technology, MSc (Tech)
June 2021

Software reuse is a topic that has been studied for decades. If done well, the benefits of reuse are clear. Time is saved and the quality of the software increases. However, in order to achieve the benefits, reuse must be done in a sensible way.

This thesis studies how to reuse source code from a system in production in the implementation of a new module in terms of time used, maintainability, and error sensitivity. The goal is to benefit from the advantages of reuse and minimize the risks associated with modifying source code in production use. To achieve this, this thesis examines the aspects of software reuse as well as component-based development, which is a commonly used approach to produce reusable software.

This thesis presents the design and implementation of a purchase invoice module for an invoicing and collection system. The design began by comparing two different options for how the source code can be reused from an existing and in production use sales invoice implementation. The source code could either be refactored or utilized by copying and editing. It was decided that the source code would be refactored. A generic invoice component was separated and modified from the sales invoice implementation. The component was utilized in the implementation of the purchase invoice module.

The selected method of implementation was calculated to have slowed down the completion of the module by 80 hours. A total of 450 hours were spent on the implementation of the module. The extra time spent in the implementation phase should be, however, reflected in more swift maintenance and further development in the future. In addition, the quality of the sales invoice implementation was increased at the same time. The refactoring of the source code also did not cause any errors in the sales invoice features, which were already in use, although it was seen as a big risk.

Keywords: code reuse, software reuse, component-based development

ALKUSANAT

Diplomityön suunnittelu aloitettiin joulukuussa 2019 ja se saatiin puristettua valmiiksi asti isojen hidasteiden jälkeen kesäkuussa 2021. Haluan kiittää ohjaajaani Outi Sievi-Kortea aina nopeasti tulleesta palautteesta ja neuvoista diplomityön teon aikana. Lisäksi haluan kiittää OneByte Oy:n Tampereen toimistoa häiriövapaasta sijainnista ja kavereitani Hatea henkisestä tuesta ja erityisesti Villeä vertaistuesta diplomityön kirjoittamisen aikana.

Tampereella, 13. kesäkuuta 2021

Joni Saarinen

SISÄLLYSLUETTELO

KUVALUETTELO

TAULUKKOLUETTELO

LYHENTEET JA MERKINNÄT

COTS	Kaupallinen ohjelmistokomponentti (engl. Commercial-off-the-shelf)
HTTP	Tiedonsiirtoprotokolla (engl. Hypertext Transfer Protocol)
ICSE	Kansainvälinen ohjelmistokehityskonferenssi (engl. International Conference on Software engineering)
IEEE-standardi	Kansainvälisen tekniikan alan järjestön (engl. Institute of Electrical and Electronics Engineers) määrittelemä standardi
JSON	Datan esitystapa (engl. JavaScript object notation)
MVC-arkkitehtuuri	Ohjelmistoarkkitehtuuri (engl. Model-view-controller)
ORM	Tietokannan abstrahointi oliopohjaiseksi (engl. Object relational mapping)
REST	Arkkitehtuurityyli (engl. Representational state transfer)
SFTP	Tiedonsiirtoprotokolla (engl. Secure File Transfer Protocol)
URI	Tiedon paikka (engl. Unique Resource Identifier)
XML	Datan esitystapa (engl. Extensible markup language)

1 JOHDANTO

Lähdekoodin uudelleenkäyttö on ohjelmistokehittäjille vakiintunut ja hyväksi todettu käytäntö. Uudelleenkäytöllä pyritään säästämään aikaa. On nopeampaa käyttää valmista ratkaisua kuin tehdä kaikki alusta asti itse. Lisäksi useasti käytetyn ratkaisun voidaan ajatella olevan koetellumpi kuin täysin uuden. Uudelleenkäyttöön liittyy kuitenkin monia haasteita. Uudelleenkäytettävien ratkaisujen kehittäminen vaatii aikaa ja vaivaa. Nykyajan kiireessä uudelleenkäytettävyydestä luovutaan helposti ensimmäisenä, vaikka jatkossa siitä olisi merkittävääkin hyötyä. Lisäksi huonosti tehtynä uudelleenkäyttö voi pahentaa ongelmia, joita uudelleenkäytöllä pyrittiin ratkaisemaan.

Tämän työn tavoitteena on tutkia miten lähdekoodia kannattaa uudelleenkäyttää valmiista järjestelmästä uuden moduulin teossa ajankäytön, ylläpidettävyyden ja virheherkkyyden suhteen. Työssä toteutetaan laskutus- ja perintäjärjestelmään ostolaskumoduuli uudelleenkäyttämällä jo tuotantokäytössä olevan myyntilaskutoteutuksen lähdekoodia. Uudelleenkäytön toteutuksessa pyritään hyödyntämään alan tutkimuksissa hyväksy todettuja käytäntöjä kuten komponenttipohjaista kehitystä.

Eri tapoja uudelleenkäyttää lähdekoodia verrataan keskenään ja niistä pyritään valitsemaan tilanteeseen sopivin. Aikaa halutaan säästää pitkällä aikavälillä. Ylläpidon halutaan olevan mahdollisimman helppoa. Lisäksi halutaan minimoida käytössä olevan lähdekoodin muokkaamiseen liittyvät riskit.

Tämän työn luku 2 käsittelee ohjelmistojen uudelleenkäyttöä yleisesti sekä sitä mistä näkökulmista uudelleenkäyttöä voidaan tarkastella. Luvussa 3 käsitellään komponenttipohjaista kehitystä ja sitä miten komponenttipohjaisella kehityksellä pyritään ratkaisemaan uudelleenkäyttöön liittyviä haasteita. Luvussa 4 käydään läpi miten sähköiset myynti- ja ostolaskuprosessit toimivat. Luvussa 5 esitellään ostolaskumoduulin toteutuksen ympäristö eli käytettävät teknologiat sekä laskutus- ja perintäjärjestelmä, johon moduuli tehdään. Luku 6 sisältää kuvauksen ostolaskumoduulin toteutuksesta ja siihen liittyvästä uudelleenkäytöstä. Luvussa 7 arvioidaan toteutuksen onnistumista. Työn yhteenveto on luvussa 8.

2 OHJELMISTOJEN UUDELLEENKÄYTTÖ

Termi ohjelmistojen uudelleenkäyttö (engl. software reuse) on ensimmäisen kerran mainittu vuonna 1968 ensimmäisessä ICSE-konferenssissa (International Conference on Software Engineering) [1]. Ohjelmistojen uudelleenkäytön tavoitteena on saavuttaa parempaa tuottavuutta ja laatua sekä säästää kustannuksissa [2][3][4].

Ohjelmistojen uudelleenkäytölle on useita määritelmiä [4]. IEEE-standardi 1517-2010 (Standard for Information Technology-System and Software Life Cycle Processes-Reuse Processes) määrittelee termin ohjelmistojen uudelleenkäyttö seuraavasti: "Ohjelmistojen uudelleenkäyttö merkitsee olemassa olevien ohjelmistojen ja järjestelmien hyödyntämistä uusien tuotteiden luomiseksi."[5] Ohjelmistojen uudelleenkäyttö voi myös kuvata vanhojen ohjelmien ylläpitämistä. Päivitetyt ohjelmat voidaan nähdä vanhan käytetyn ohjelman uutena versiona. [2]

Tässä luvussa käydään läpi yleisimpiä tapoja uudelleenkäyttää ohjelmistoja. Uudelleenkäytön muotoja on useita riippuen tarkastelutasosta. Matalan tason keinoja ovat muun muassa ohjelmointikielten ominaisuudet, kirjastot ja viestiprotokollat [6, s. 152]. Tässä työssä keskitytään korkeamman tason tapoihin. Alan tutkimuksissa käytetyimmät keinot ovat komponenttipohjainen kehitys (engl. component-based development) ja tuoterungot (engl. software product line) [4]. Komponenttipohjainen kehitys ja tuoterungot ovat ennalta suunniteltuja uudelleenkäytön lähestymistapoja. Tässä työssä merkittävässä osassa on myös pragmaattinen uudelleenkäyttö (engl. pragmatic reuse), jossa uudelleenkäytetään lähdekoodia, jota ei ole suunniteltu uudelleenkäytettäväksi [7].

2.1 Uudelleenkäytön näkökulmat

Ohjelmistojen uudelleenkäyttöä voidaan tarkastella monesta näkökulmasta [8]. Näitä näkökulmia on koottu taulukkoon 2.1. Sisältö määrittelee uudelleenkäytettävän asian olemuksen. Laajuus määrittelee uudelleenkäytön muodon ja laajuuden. Strategia määrittelee miten elementtejä uudelleenkäytetään. Tapa määrittelee miten uudelleenkäyttö suoritetaan. Käyttöaste määrittelee kuinka paljon elementtiä uudelleenkäytetään. Lähestymistapa määrittelee uudelleenkäytön tarkoituksen. Tuote määrittelee mitä ohjelmistotuotteita uudelleenkäytetään.

Sisältö

Ideoiden ja konseptien uudelleenkäyttö sisältää yleiskäyttöiset ratkaisut tietyn kategorian

Taulukko 2.1. *Uudelleenkäytön näkökulmat. Perustuu lähteeseen [8]*

Sisältö	Laajuus	Strategia	Tapa	Käyttöaste	Lähestymistapa	Tuote
Ideat, konseptit	Vertikaalinen	Musta laatikko organisaation sisällä	Ennalta suunniteltu, systemaattinen	Räätälöidyt ratkaisut	kehittäminen käyttäen valmiina olevia toteutuksia	Lähdekoodi
Artefaktit, komponentit	Horisontaalinen	Musta laatikko organisaation ulkopuolella	Ad-hoc, pragmaattinen	Standardoidut ohjelmistopaketit	Uudelleenkäytettävien toteutuksien kehittäminen	Design
Proseduurit, taidot		Lasilaatikko				Suunnitteludokumentit Teksti Arkkitehtuuri

ongelmiin [8]. Esimerkki tällaisesta uudelleenkäytöstä on suunnittelumallit, joita käsitellään luvussa 3.4.

Artefaktien ja komponenttien uudelleenkäyttö on varsinaista lähdekoodin uudelleenkäyttöä. Tähän osa-alueeseen on tutkimuksissa keskitytty eniten. [8] Tässä työssä uudelleenkäytön tarkastelu keskittyy tähän sisällön osa-alueeseen.

Proseduurien uudelleenkäyttö keskittyy ohjelmistokehityksen proseduurien määrittämiseen ja kapselointiin. Tavoitteena on luoda uudelleenkäytettäviä prosesseja, joita voidaan yhdistää ja luoda täten uusia monimutkaisempia prosesseja. Proseduurien uudelleenkäyttö tarkoittaa myös taitojen ja tiedon uudelleenkäyttöä. [8]

Laajuus

Vertikaalinen uudelleenkäyttö tarkoittaa uudelleenkäyttöä samalla domain- tai sovellusalueella. Sen tavoitteena on kehittää järjestelmäperheille geneerisiä malleja, joita voidaan käyttää mallineena kasattaessa uutta järjestelmää. [8] Esimerkki vertikaalisesta uudelleenkäytöstä on tuoterungot, joita käsitellään luvussa 3.10.

Horisontaalisen uudelleenkäytön tavoitteena on kehittää yleiskäyttöisiä osia käytettäväksi erilaisissa sovelluksissa. Matemaattiset kirjastot ja Unixin työkalut ovat hyviä esimerkkejä horisontaalisesta uudelleenkäytöstä. [8]

Strategia

Uudelleenkäyttöstrategiat voidaan jakaa karkeasti kolmeen osaan. Näitä ovat musta laatikko -uudelleenkäyttö (engl. black-box reuse) käyttäen organisaation sisällä kehitettyjä

komponentteja, musta laatikko -uudelleenkäyttö käyttäen kolmannen osapuolen komponentteja ja lasilaatikkouudelleenkäyttö (engl. white-box reuse).

Musta laatikko -uudelleenkäytössä ohjelmistokomponenttia käytetään sellaisenaan ilman lähdekoodimuutoksia. Kehittäjän tarvitsee tietää pelkästään komponentin toiminnallisuus ja kuinka komponentti vuorovaikuttaa ympäristönsä kanssa. Tarvetta komponentin sisäisen logiikan muuttamiseen ei pitäisi olla. Musta laatikko -ajattelua hyödynnetään komponenttipohjaisessa kehityksessä, jota käsitellään tämän työn luvussa 3. [9]

Lasilaatikkouudelleenkäytössä kehittäjällä on pääsy uudelleenkäytettävän komponentin lähdekoodiin ja mahdollisuus muokata sitä. Lasilaatikkouudelleenkäyttö antaa kehittäjälle laajat mahdollisuudet muokata olemassa olevaa lähdekoodia uusiin tarpeisiin. Tämä mahdollistaa ohjelmistokomponenttien joustavan uudelleenkäytön. Vaikka tällä tavoin maksimoidaan lähdekoodin uudelleenkäyttömahdollisuudet, olemassa olevan lähdekoodin muokkaus on pääsyy ongelmille uudelleenkäytön yhteydessä. Lähdekoodin muokkaus vaatii tarkkaa tietämystä komponentin toteutusyksityiskohdista. Tällaista tietämystä on yleensä vain, jos lähdekoodi on tehty organisaation sisällä tai lähdekoodilla on erinomainen dokumentaatio. Kun ohjelmiston osia uudelleenkäyttävät kehittäjät, jotka eivät olleet osallisena alkuperäisessä kehitystyössä, on välttämätöntä, että saatavilla on henkilöitä, joille alkuperäinen lähdekoodi on tuttua. Ainoastaan tällöin uudelleenkäyttö on mahdollista kustannustehokkaalla tavalla. [9]

Tapa

Ennalta suunnitellussa uudelleenkäytössä suuntaviivat ja prosessit uudelleenkäytölle on ennalta määritelty. Dataa uudelleenkäytöstä kerätään ja metriikoiden avulla tutkitaan uudelleenkäytön tehokkuutta. Ennalta suunniteltu uudelleenkäyttö vaatii merkittävää etukäteisinvestointia ja -sitoutumista. Investoinnin tuottoprosenttia on kuitenkin vaikea ennustaa. [8]

Lasilaatikkouudelleenkäyttöön liittyy läheisesti termi pragmaattinen uudelleenkäyttö. Pragmaattinen uudelleenkäyttö on epämuodollinen käytäntö, jossa lähdekoodia uudelleenkäytetään ad-hoc-tyyppisesti. [8]

Ohjelmoinnin yhteydessä kehittäjät päätyvät usein tilanteisiin, joissa nykyinen ohjelmointitehtävä on heille tuttu. He ovat joko ennen toteuttaneet nykyisen tehtävän vaatimat toiminnallisuudet tai heillä on pääsy toiminnallisuudet sisältävään lähdekoodiin. Koska uudelleenkäytettävää lähdekoodia on kallista tehdä, on todennäköistä, että lähdekoodi, josta kehittäjä on kiinnostunut, ei ole tehty uudelleenkäyttö mielessä. Näissä tilanteissa kehittäjällä on yleensä kolme vaihtoehtoa: [7]

- toteuttaa toiminnallisuus uudestaan
- refaktoroida alkuperäinen järjestelmä siten, että toiminnallisuus on uudelleenkäytettävää
- uudelleenkäyttää toiminnallisuuden lähdekoodia kopioimalla ja muokkaamalla sitä.

Kaikissa vaihtoehdoissa on omat heikkoutensa. Toiminnallisuuden toteuttaminen uudes-

taan on kallista eikä se hyödynnä testausta tai muita maturoituneen lähdekoodin positiivisia ominaisuuksia. Alkuperäisen lähdekoodin muokkaus ei ole usein mahdollista teknisistä ja organisaatiosta johtuvista syistä. Kehittäjällä ei välttämättä ole oikeuksia olemassa olevaan lähdekoodiin, lähdekoodi voi olla jo tuotannossa eikä sitä voi muokata, kehittäjä voi olla haluton refaktoroimaan toimivaksi todettua lähdekoodia, koska se lisää riskejä virheille tai kehittäjä voi olla haluton ottamaan tietoturvariskiä, joka liittyy jaettuun lähdekoodiin. Lähdekoodin uudelleenkäyttö kopioimalla ja muokkaamalla saattaa aiheuttaa kehittäjän tekemään huonoja päätöksiä. Viimeisin vaihtoehto on kuitenkin yleensä teollisuudessa käytetyin vaihtoehto. [7]

Pragmaattinen uudelleenkäyttö ei estä ennalta suunniteltua uudelleenkäyttöä. Esimerkiksi uudelleenkäytettävää komponenttia alun perin kehitettäessä voidaan hyödyntää pragmaattista uudelleenkäyttöä ja näin kerätä molemmista parhaat puolet. Sekä ennalta suunnitellussa että pragmaattisessa uudelleenkäytössä on samoja haittapuolia. Oletukset niiden kontekstista eivät välttämättä ole eksplisiittisiä tai ainakaan automaattisesti tarkastettavissa. Esimerkiksi kenelle tulisi mieleen dokumentoida, että viikossa on 7 päivää? Lisäksi molemmat lähestymistavat vaativat uudelleenkäytettävän toiminnallisuuden paikantamista. Ennalta suunnitellut lähestymistavat olettavat, että on olemassa komponentti, joka joko sopii tarkoitukseen täydellisesti tai kohdejärjestelmä on muokattavissa siten, että komponentti saadaan sopimaan siihen täydellisesti. Pragmaattinen lähestymistapa olettaa, että toiminnallisuus on ylipäätään pätevä kohde muokkaukselle. [7]

Käyttöaste

Perinteinen ohjelmistokehitys voidaan jakaa karkeasti kahteen eri ääripäähän uudelleenkäyttöasteen suhteen. Toisessa päässä on räätälöidyt ratkaisut, jotka tehdään täysin perustuen yksittäisen asiakkaan vaatimuksiin. Toisessä päässä taasen on standardoitujen ohjelmistopakettien kehitys perustuen kokonaiseen asiakassegmenttiin. [6, s. 4]

Räätälöityjen ratkaisujen hyvä puoli on asiakkaan omien prosessien täysi tuki. Jos ne ovat uniikkeja, saadaan kilpailuetua verrattuna asiakkaan kilpailijoihin [10]. Huonona puolena on korkeat kehityskustannukset ja yleensä pitkään kuluva aika ennen valmista tuotetta [11, s. 89]. Toinen huono puoli on epävarmuus uuden järjestelmän kyvyssä kommunikoida olemassa olevien ja tulevien järjestelmien kanssa [6].

Standardoiduilla ohjelmistopaketeilla ei ole edellä mainittuja ongelmia. Niillä on kuitenkin omat ongelmansa. Standardoidun paketin käyttöönotto saattaa vaatia muutoksia vallitseviin liiketoimintaprosesseihin. Toinen huono puoli on, että kilpailijat voivat hankkia saman paketin eikä tällöin saada kilpailuetua. [11, s. 89]

Aiemmin mainitut ovat kaksi ääripäätä. Käytännössä ohjelmistokehitysprojektit koostuvat näiden kahden yhdistelmästä. Yleensä kehitysorganisaatio keskittyy ydinliiketoimintaansa kehittäen osia, joissa organisaatiolla on kilpailuetua. Osiin, jotka eivät ole olennaisia liiketoiminnan kannalta, ei käytetä vaivaa. [11, s. 90] Esimerkiksi tässä työssä käsiteltävässä ostolaskumoduulissa keskeisiä osia ovat ostolaskujen käsittely ja siihen liittyvä rahaliikenne ja rahan käsittely. Web-teknologiaan ja siihen liittyviin komponentteihin hyö-

dynnetään valmista ja yleiskäyttöistä Django-web-ohjelmistokehystä.

Lähestymistapa

Kehittäminen käyttäen valmiina olevia toteutuksia (engl. with-reuse) on lähestymistapa, jolla varsinaisesti saavutetaan uudelleenkäytön tavoitteita eli parempaa tuottavuutta, laatua ja säästöä kustannuksissa. Uudelleenkäytettävien toteutuksien kehittäminen (engl. for-reuse) [3] on sen sijaan kallista [8] ja hyödy saadaan vasta, kun toteutusta on käytetty useasti. Tutkimusten perusteella hyötyä voidaan saada jo kolmella uudelleenkäytöllä [12].

Tuote

Taulukossa 2.1 on osittainen listaus ohjelmistotuotteista, joita voidaan uudelleenkäyttää. Huomionarvoista on, että perinteisen ja yleisimmin käytetyn lähdekoodin uudelleenkäytön lisäksi ohjelmistoissa on muitakin uudelleenkäytettäviä elementtejä kuten suunnitteludokumentit, tekstit ja arkkitehtuurit.

2.2 Näkökulmat tässä työssä

Uudelleenkäytön näkökulma vaikuttaa merkittävästi siihen minkä tyyppisiä ratkaisuja uudelleenkäyttö vaatii. Tässä työssä toteutetaan uudelleenkäytettävä laskukomponentti uudelleenkäyttämällä olemassa olevaa lähdekoodia. Sen jälkeen laskukomponenttia hyödynnetään uuden ostolaskumoduulin toteutuksessa. Sisällöltään uudelleenkäyttö kohdistuu siis pelkästään lähdekoodiin. Laajuudeltaan uudelleenkäyttö on vertikaalista. Komponenttia tullaan uudelleenkäyttämään pelkästään yhden järjestelmän sisällä. Uudelleenkäytön kapeus helpottaa toteutusta. Komponentista ei tarvitse tehdä kovinkaan geneeristä.

Uudelleenkäyttö on strategialtaan lasilaatikkouudelleenkäyttöä. Olemassa olevaa lähdekoodia muokataan uusin tarpeisiin. Tähän liittyy myös pragmaattinen uudelleenkäyttö. Lasilaatikkouudelleenkäyttö ja pragmaattinen uudelleenkäyttö aiheuttavat tässä työssä uudelleenkäytön suurimmat haasteet. Olemassa olevan lähdekoodin muokkaus sisältää isoja riskejä.

3 KOMPONENTTIPOHJAINEN KEHITYS

Komponenttipohjainen kehitys on lähdekoodin uudelleenkäyttöön perustuva lähestymistapa ohjelmistojen tuottamiseen. Se sisältää komponenttien määrittelyn, toteutuksen ja integroinnin uuteen järjestelmään. Komponentit voivat olla kehitetty joko uudelleenkäyttöä varten tai tulla käytetyksi toisesta järjestelmästä. Komponenttipohjaisen kehityksen perusidea on, että samantyyppisiä toiminnallisuuksia voidaan käyttää eri tilanteissa. Tällöin toiminnallisuuden tarjoavasta komponentista voidaan tehdä uudelleenkäytettävä. [11]

3.1 Komponentit

Komponentti on komponenttipohjaisen kehityksen keskeinen käsite. Komponentti on uudelleenkäytettävä käyttöönottoyksikkö (engl. unit of deployment) ja kooste (engl. composition), johon pääsee käsiksi rajapinnan kautta. [11, s. 4] Käyttöönottoyksikkö tarkoittaa, että komponenttia ei ikinä käyttöönoteta osittain [6, s. 36].

Komponentti ja luokka ovat käsitteinä hyvin lähellä toisiaan [11, s. 8]. Komponentti voi olla yksittäinen luokka [6, s. 11]. Kuitenkin yleensä komponentin voidaan nähdä sisältävän useita luokkia, jotka tekevät tiivistä yhteistyötä keskenään ja ovat vahvasti sidoksissa toisiinsa [13]. Raja komponenttien välillä on tarkasti määritelty. Komponentin sisällä sen sijaan luokilla on pääsy toistensa toteutukseen. Komponentin ulkopuolelta ei kuitenkaan ole pääsyä luokkien toteutukseen. Komponentti voi sisältää luokkien sijaan proseduureja ja globaaleja muuttujia. Tällöin komponenttipohjainen kehitys mukautuu myös funktionaaliseen ohjelmointiin ja jopa assembly-ohjelmointikieliin. [11, s. 8]

Komponentin tärkein ominaisuus on sen rajapinnan erottaminen komponentin varsinaisesta toteutuksesta. Tämä tarkoittaa, että komponentin integrointi ja käyttöönotto pitäisi olla riippumaton komponentin elinkaaresta. Komponenttia päivitettäessä ei pitäisi olla tarvetta linkittää sitä uudelleen sovellukseen. Komponentin pitäisi myös olla käytettävissä pelkästään sen rajapinnan kautta. [11, s. 5]

Komponentit ymmärretään usein akateemisessa maailmassa ja teollisuudessa eri tavalla [14]. Akateeminen näkemys on, että komponentti on hyvin määritelty, usein pieni ja siinä on helposti ymmärrettävät toiminnalliset ja ei-toiminnalliset ominaisuudet. Komponentti on musta laatikko, koska se on kapseloitu rajoittaen pääsyn sisäiseen toteutukseen. Teollisuudessa noudatetaan tätä ajatusta pääpiirteissään. Kuitenkin monesti komponentit nähdään isoina ohjelmistojen osina, jotka ovat uudelleenkäytettäviä ja niissä on monimutkainen sisäinen rakenne. Komponentilla ei välttämättä ole helposti ymmärrettä-

vää rajapintaa, joka kapseloisi ja estäisi pääsyn sisäiseen toteutukseen. [11, s. 6] Tämä pätee erityisesti tuoterunkoarkkitehtuureissa, joissa eri konsepteja ja komponenttimalleja käytetään samassa järjestelmässä [15].

Jotta komponentti täyttää yleiset komponentin vaatimukset ja jotta komponentille varmistetaan oikeanlainen integraatio, ylläpito ja päivittäminen, komponentin pitäisi sisältää seuraavat elementit. Komponentilla pitää olla joukko rajapintoja, jotka tarjotaan ympäristölle tai jotka on vaatimuksena ympäristöltä. Kyseiset rajapinnat ovat tarkoitettu erityisesti kommunikointiin muiden komponenttien kanssa perinteisten ohjelman osien sijaan. Komponentin pitää sisältää suoritettavaa lähdekoodia, jonka voi liittää muiden komponenttien lähdekoodin rajapintojen kautta. [11, s. 7]

Lisäksi komponentin yleisiin vaatimuksiin voidaan lisätä seuraavat elementit, jos komponentin laatua halutaan kasvattaa. [11, s. 7]

- Tarjottujen ja vaadittujen ei-toiminnallisten vaatimusten määrittely
- Validointikoodi, joka varmistaa yhteyden toiminnan toiseen komponenttiin
- Dokumentaatio kuten vaatimusmäärittely, suunnitteluratkaisut ja käyttötapaukset

3.2 Rajapinnat

Komponentin rajapinta voidaan määritellä komponentin yhteyspisteen määritelmäksi. Käyttäjä, yleensä toinen komponentti, pääsee käsiksi komponentin tarjoamiin palveluihin näiden yhteyspisteiden kautta. [6, s. 42] Rajapinnan ei ole tarkoitus tarjota toteutusta mihinkään sen sisältämistä operaatioista, vaan se tarjoaa pelkästään operaatioiden kuvaukset ja protokollat. Tämä erottelu mahdollistaa toteutuksen vaihtamisen ilman rajapinnan muuttamista sekä uusien rajapintojen lisäämisen ilman nykyisten rajapintojen muuttamista. [11, s. 9]

Ideaalimaailmassa komponenttien pitäisi olla mustia laatikoita, jotta käyttäjät voivat uudelleenkäyttää niitä ilman tietoa komponenttien sisäisestä toteutuksesta. Toisin sanoen komponentin rajapinnan pitäisi tarjota kaikki käyttäjän tarvitsema tieto. Tämän tiedon pitäisi olla ainoa käyttäjän tarvitsema tieto. Rajapinnan pitäisi myös olla ainoa yhteyspiste komponenttiin. [11, s. 23]

Rajapinnat voidaan erotella kahteen eri tyyppiin. Komponentit voivat viedä tai tuoda rajapintoja ympäristöistä, joissa on muita komponentteja. Ympäristöön viety rajapinta kuvaa komponentin tarjoamat rajapinnat ympäristölle. Ympäristöstä tuotu rajapinta kuvaa komponentilta vaaditut palvelut ympäristön toimesta. [11, s. 9]

3.3 Sopimussuunnittelu

Komponenttien käyttäytymistä voidaan kuvata sopimusten avulla. Tällöin rajapinta määritellään eräänlaisena sopimuksena. Sopimus määrittää mitä käyttäjän pitää tehdä käyttääkseen rajapintaa. [6, s. 53] Sopimus listaa komponentin ylläpitämät globaalit rajoitteet

eli invariantit (engl. invariant). Invariantit testaavat, että komponentti ei ole virheellisessä tilassa. Sopimus listaa myös käyttäjän rajoitteet ennen kutsua eli esiehdot (engl. precondition) ja kutsun jälkeiset rajoitteet eli jälkiehdot (engl. postcondition), jotka komponentti lupaa asettaa vastineeksi. Esi- ja jälkiehdot listataan jokaisesta komponentin operaatiosta. Invariantit, esiehdot ja jälkiehdot yhdessä määrittävät komponentin käyttäytymisen. [16]

Yksittäisen komponentin käyttäytymisen kuvaamisen lisäksi sopimus voi määrittää myös komponenttijoukon välistä vuorovaikutusta. Tällöin sopimukset toimivat kuitenkin hieman eri tavalla. Sopimus kuvaa komponenttien välistä vuorovaikutusta seuraavien asioiden suhteen: [11, s. 11]

- Osallistuvien komponenttien joukko
- Kunkin komponentin rooli sopimusvelvoitteidensa, kuten tyyppivaatimusten, kautta. Tyyppivaatimukset edellyttävät, että komponentti tukee tiettyjä muuttujia ja rajapintoja sekä että komponentti suorittaa toimenpiteitä määrätyssä järjestyksessä mukaan lukien viestien lähettäminen toisille komponenteille.
- Komponenttien ylläpitämä invariantti
- Komponenttien alustusoperaatioiden määrittely

Komponentit eivät ainoastaan tarjoa palveluita, vaan ne myös vaativat palveluita muilta komponenteilta. Tämä pätee sekä toiminnallisiin että ei-toiminnallisiin vaatimuksiin. Tämän vuoksi komponenttijoukon sopimusvelvoitteet eroavat merkittävästi vain yksittäisen komponentin operaatioiden esi- ja jälkiehdoista. [11, s. 11]

Valitsemalla komponentit ja operaatiot oikein komponentit toimivat yhteisessä sopimuksessa saavuttaakseen tietyn tavoitteen tai ylläpitääkseen invarianttia. Tällaisessa komponenttijoukossa jokainen komponentti tarjoaa osan vaaditusta toiminnallisuudesta ja kommunikoi muiden joukon jäsenten kanssa. [11, s. 11]

Sopimuksien käyttö komponenttien välisen vuorovaikutuksen määrittämiseen tukee moniosaisten ohjelmistokomponenttien uudelleenkäyttöä ja jatkokehitystä seuraavilla tavoilla: [11, s. 11-12]

- Sopimukset sallivat ohjelmistokehittäjän eristää ja määrittää eksplisiittisesti korkealla abstraktiotasolla eri komponenttien roolit tietyssä ympäristössä.
- Erilaiset sopimukset mahdollistavat jokaisen komponentin roolin muokkaamisen ja laajentamisen itsenäisesti.
- Uusia sopimuksia voidaan määrittää yhdistelemällä eri komponentteja erilaisilla rooleilla.

Yksittäisen komponentin käyttäytymistä voi olla melko monimutkaista määrittää, koska komponentti saattaa olla osallisena monessa sopimuksessa. [11, s. 12]

3.4 Suunnittelumallit

Suunnittelumallit (engl. pattern) määrittävät yleisiä ratkaisuja usein toistuviin ongelmiin korkeammalla abstraktiotasolla. Suunnittelumallit tallentavat ei ilmeisiä ratkaisuja, eivät pelkästään abstrakteja periaatteita tai strategioita. Tämä erottaa suunnittelumallit monista muista ongelmanratkaisutekniikoista, kuten ohjelmointiparadigmoista, jotka johtavat ratkaisuja abstrakteista periaatteista. Suunnittelumallien ratkaisujen pitäisi todistetusti ratkaista tietty ongelma sen sijaan että kyseessä olisi teoria tai pohdinta. Suunnittelumallit kuvaavat syvempien järjestelmän rakenteiden ja mekanismien, eivät pelkästään itsenäisten moduulien, välisiä suhteita. Lisäksi inhimilliset tekijät ovat osa suunnittelumalleja. [17]

Suunnittelumallit voidaan luokitella kolmeen eri kategoriaan riippuen niiden abstraktiotasosta. Korkeimmalla abstraktiotasolla olevat arkkitehtuurisuunnittelumallit (engl. architectural pattern) käsittelevät laajoista komponenteista kasatun järjestelmän globaaleja ominaisuuksia ja arkkitehtuuria. Alemmalla abstraktiotasolla suunnittelumallit (engl. design pattern) käsittelevät alijärjestelmien rakennetta ja käyttäytymistä sekä niiden välisiä suhteita. Alimmalla abstraktiotasolla on idioomat, jotka riippuvat valitusta ohjelmointiparadigmasta ja ohjelmointikielestä. [11, s. 13]

Suunnittelumalleja on hyödynnetty monen oliopohjaisen järjestelmän suunnittelussa ja niitä pidetään hyödyllisenä järjestelmän kokonaisarkkitehtuurin kannalta. Suunnittelumallien käyttö ei kuitenkaan ole ongelmatonta. [11, s. 13]

- Suunnittelumalleihin sisällytetty tieto on epämuodollista ja se sisältää paljon monitulkintaisuutta, koska ratkaisun kuvaus on vapaamuotoinen.
- Tietyn toteutuksen rakenteen validointi suunnittelumallin määrittelyä vasten on haasteellista.
- Suunnittelumalleissa on hankala päätellä, onko joku suunnittelumalli toisen suunnittelumallin erikoistapaus.

Suunnittelumalleja hyödynnetään laajasti komponenttipohjaisissa järjestelmissä, joissa pitää tunnistaa uudelleenkäytettävät yksiköt. Käyttämällä suunnittelumalleja uudelleenkäytettävät osat ovat helpompi tunnistaa. Osat voivat olla joko olemassa olevissa komponenteissa tai ne voidaan kehittää erikseen. Suunnittelumallia voidaan käyttää myös kuvaamaan komponentin käyttäytymisen ja rakenteen matalan tason toteutusyksityiskohtia. Täten suunnittelumalleja voidaan käyttää myös komponenttien kehittämiseen. Lisäksi suunnittelumalleja voidaan käyttää komponenttikomposition kuvaamiseen kehitettäessä ohjelmistokehystä, johon liittyy useita komponentteja. [11, s. 13]

3.5 Ohjelmistokehykset

Ohjelmistokehyksille (engl. framework) on useita määritelmiä riippuen näkökulmasta ja yksityiskohtien tasosta [11, s. 14]. Yleisellä tasolla ohjelmistokehys voidaan määritellä "sovelluksen rungoksi, jota sovelluskehittäjä voi kustomoida"[18]. Toinen määritelmä ark-

kitehtuurin näkökulmasta: "sovelluskehys on mikroarkkitehtuuri, joka tarjoaa osittaisen mallin järjestelmille tietyssä domainissa". [19] Ohjelmistokehyksen voi myös ajatella liittyvän ideaan, jossa tietyn järjestelmän suunnittelun yhteydessä voi tulla abstrakteja toteutuksia, joita voidaan uudelleenkäyttää toisessa tilanteessa. Ohjelmistokehys kuvaa silloin asiaa, joka on käytettävissä pienten muutosten jälkeen myös muissa vastaavissa tilanteissa. [11, s. 14]

Ohjelmistokehykset liittyvät läheisesti suunnittelumalleihin. Suunnittelumallit ovat abstraktimpia ja ne kuvaavat tietoa ja kokemusta, jota on saatu ohjelmista. Ohjelmistokehykset ovat taasen suoritettavaa lähdekoodia. Ne ovat siis yhden tai useamman suunnittelumallin fyysinen toteutus. [17]

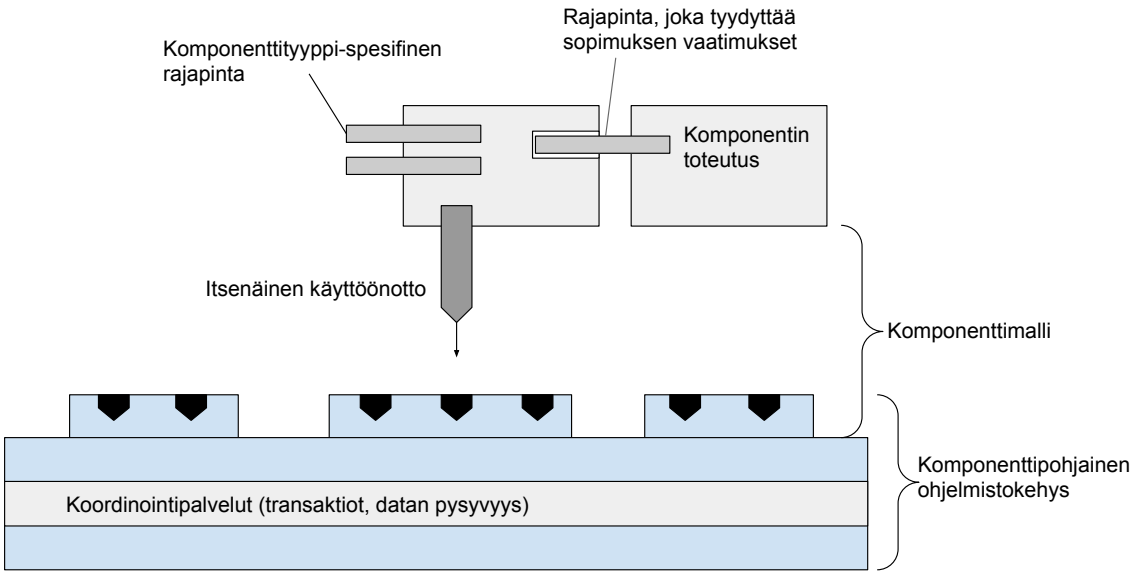
Ohjelmistokehykset voidaan erotella matalan tason sovelluskehyksiin, yleisiin/liiketoiminta ohjelmistokehyksiin tai komponenttipohjaisiin ohjelmistokehyksiin (engl. component framework). Esimerkki matalan tason sovelluskehyksestä on Smalltalkin MVC-ohjelmistokehys. [11, s. 14] Komponenttipohjainen ohjelmistokehys voidaan kuvata tyhjiä paikkoja sisältävänä piirilevynä. Ohjelmistokehys alustetaan sijoittamalla komponentteja tyhjiin paikkoihin. Vaatimusmäärittelyllä osoitetaan miten komponenttien pitää mukautua, jotta ne toimivat suunnitellusti piirillä. Tyhjät paikat voidaan täyttää atomisilla komponenteilla tai toisilla ohjelmistokehyksillä. Kun ohjelmistokehys alustetaan komponenteilla, siitä syntyy sovellus ja lisäksi siitä tulee uusi komponentti käytettäväksi muihin ohjelmistokehyksiin. [11, s. 17-18]

Ohjelmistokehykset tarjoavat kontekstin, jossa komponentteja voidaan koota yhteen ja niitä voidaan käyttää tehokkaasti yhdessä. [11, s. 16] Nykyään komponenttipohjaisia ohjelmistokehyksiä käytetään esimerkiksi web-kehityksessä. Web-ohjelmistokehyksiä ovat muun muassa PHP-pohjainen Laravel [20] ja python-pohjainen Django [21].

3.6 Käsitteiden väliset suhteet

Kuvassa 3.1 esitetään komponenttipohjaisen kehityksen keskeisimpien käsitteiden väliset suhteet. Komponentti toteuttaa yhden tai useamman rajapinnan. Komponentin rajapinta täyttää sopimuksen määrittämät vaatimukset. Komponenttipohjainen järjestelmä perustuu tietyn tyyppisille komponenteille, joista jokainen täyttää tietyn roolin järjestelmässä. Jokaisen komponentin kuvaa rajapinta. [22]

Komponenttimalli on joukko komponenttityyppejä ja niiden rajapintoja. Komponenttimalliin kuuluu lisäksi määritelmät sallituista komponenttityyppien välisen interaktion malleista. Komponenttipohjainen ohjelmistokehys tarjoaa joukon käyttöönotto- ja ajonaikaisiapalveluilta komponenttimallin tueksi. [22] Joidenkin näkemysten mukaan ohjelmistokehysten ei välttämättä tarvitse tarjota palveluita [23].



Kuva 3.1. *Komponenttipohjaisen kehityksen keskeisimpien käsitteiden väliset suhteet. Perustuu lähteisiin [22] ja [11, s. 16].*

3.7 Komponenttien kehitys

Kuten aiemmin on mainittu, komponenttien pääsuunnitteluperiaatteena on uudelleenkäytettävyys. Verrattuna spesifisiin ratkaisuihin komponentit pitää huolellisesti generalisoida, jotta niitä voi käyttää erilaisissa ympäristöissä. Geneerisen ongelman ratkaisu vaatii enemmän työtä kuin spesifin ongelman. Lisäksi erilaisten ympäristöjen takia asianmukaisen dokumentaation, testien, tutoriaalien ja muun vastaavien luonti vaatii enemmän resursseja komponenttien kehittämisessä kuin spesifissä ratkaisussa. Tämä pätee erityisesti, jos komponentteja on tarkoitus myydä erillisenä tuotteena. [6, s. 17]

Komponenttien kehitys on täten järkevää ainoastaan, jos niiden kehittämiseen käytetty investointi saadaan takaisin tuottona käyttöönoton myötä. Jos komponentti kehitetään sisäiseen käyttöön, tuotto voi olla rahan sijasta epäsuoria etuja, jotka saavutetaan käyttämällä komponenttia monoliittisen ratkaisun sijasta. Tällaisia etuja ovat yleensä nopeampi valmis tuote ja parempi hallittavuus, ylläpidettävyys, muunneltavuus ja joustavuus. [6, s. 17]

Tuottoa saadaan myös myymällä komponentteja. Komponenttien myynnin yhteyteen voidaan myös liittää palveluiden myynti. Komponentti voi olla hinnaltaan halpa tai jopa ilmainen, mutta sen käyttö voi vaatia erityistä asiantuntijuutta, jota myydään tällöin palveluna. [6, s. 17]

3.8 Komponenttien käyttö

Komponentit voidaan jakaa kolmeen tyyppiin niiden käyttötarkoituksen perusteella. Räätälöidyt komponentit (engl. custom-built components) ovat komponentteja, jotka organisaatio kehittää tiettyä tarkoitusta varten. Uudelleenkäytettävät komponentit (engl. reusable

components) ja kaupalliset komponentit (engl. COTS, Commercial-off-the-shelf) ovat molemmat jo olemassa olevia komponentteja. Uudelleenkäytettävät komponentit ovat yleisiä komponentteja, jotka ovat jo organisaation hallussa. Kaupalliset komponentit ovat komponentteja, jotka on kehitetty myytäväksi. Projektin suunnittelun yhteydessä pitää yleensä päättää sopiva komponenttien lähde. Projektin tyyppi ja kehitysorganisaation kulttuuri saattavat myös vaikuttaa päätökseen. Komponentin tyyppi vaikuttaa sen laatuominaisuuksiin. [11, s. 49]

Räätälöityjen komponenttien kustannukset ovat korkeita rahassa ja ajassa mitattuna. Lopputuloksena saadaan kuitenkin todennäköisemmin vaatimuksia vastaava komponentti kuin jo olemassa olevilla komponenteilla. Ohjelmistokehityksessä räätälöityjen komponenttien käyttö kannattaa yleensä ohjelmistoissa, jotka ovat epätavallisia tai turvallisuuskriittisiä. [11, s. 50]

Jo olemassa olevien komponenttien käytössä on huomioitava, että järjestelmän suunnittelijalla ei välttämättä ole pääsyä olemassa olevien komponenttien tai ohjelmistokehysten lähdekoodiin. Liiketoimintapäätöksenä on saatettu päättää, että nykyisestä järjestelmästä uudelleenkäytetään tiettyjä osia, jotka ovat jo käytössä. Markkinointipuolella on saatettu päättää, että käytetään tiettyä teknologiaa. Organisaatiolla saattaa myös olla kokemusta tietyistä komponenteista. Tiukat aikataulut voivat kannustaa kaupallisten komponenttien käyttöön. [11, s. 50]

Olemassa olevien komponenttien ja ohjelmistokehysten, johon suunnittelijalla ei ole hallintaa, käyttö järjestelmässä on olennaisesti erilainen ongelma kuin räätälöidyt ratkaisut. Tällaisissa tapauksissa komponentit ja ohjelmistokehykset ajavat suunnittelua. Komponentit eivät yleensä ole helposti muunneltavissa, joten ratkaisu on keskittyä komponenttien väliseen interaktion sovittamiseen komponenttien sovittamisen sijasta. [11, s. 50]

Monenlaisia komponentteja voidaan pitää uudelleenkäytettävinä. Toisessa päässä on räätälöidyt komponentit, jotka on suunniteltu organisaation toimesta tiettyä tarkoitusta varten, mutta ei kuitenkaan uudelleenkäytettävyys huomioiden. Niitä pitää muokata, jotta komponentit soveltuvat uuteen ympäristöön. Lisäksi niitä pitää analysoida, jotta saadaan tietoa, miten komponentit varsinaisesti koostuvat yhteen. Toisessa päässä on komponentit, jotka ovat kehitetty uudelleenkäyttö mielessä. Nämä voivat olla parametrisoituja komponentteja, joita voidaan käyttää suoraan erilaisilla asetuksilla. Komponenttien integroinnin ja koostamisen helppous riippuu komponenttien uudelleenkäytettävyyden tasosta. Tuoterungot ovat esimerkki etukäteen uudelleenkäytettäväksi suunnitelluista komponenteista. [11, s. 50-51] Tuoterunkoja käsitellään tämän työn luvussa 3.10.

Kaupallisten komponenttien käyttö on lisääntynyt modernissa ohjelmistotuotannossa. Tämä johtuu seuraavista syistä: [24, s. 62]

- Kaupallisten komponenttien käytöllä saadaan merkittävää lisäystä tuottavuuteen
- Tuotteen läpäisyaika vähenee
- Tuotteen laadun oletetaan olevan korkea, olettaen että kaupallinen komponentti on hyvin testattu

- Henkilöstön käyttö on tehokkaampaa, koska kehittäjien aikaa vapautuu toisiin tehtäviin

Kaupallisten komponenttien käyttöön liittyy myös riskejä. Kaupallisten komponenttien käyttö järjestelmässä synnyttää paljon epävarmuuksia suunnitteluprosessiin. Markkinoiden vaatimukset vaikuttavat komponenttien laatuominaisuuksiin. Komponentit voivat olla monimutkaisia, koska markkinat vaativat jatkuvasti uusia ominaisuuksia. [11, s. 51] Järjestelmän suunnittelijoilla ei myöskään ole yleensä mahdollisuutta muokata komponentteja vastaamaan paremmin käyttäjien vaatimuksia. Lisäksi suunnittelijoilla ei ole pääsyä komponentin jatkokehitykseen ja ylläpitoon, vaan he ovat täysin riippuvaisia komponentin toimittajasta. [24, s. 62] Komponentit voivat olla idiosynkraattisia, koska erottuvuudesta on hyötyä markkinoilla verrattuna standardisointiin. Markkinoiden paineet voivat johtaa komponenttien epävakauteen. [11, s. 51]

3.9 Edut ja haasteet

Komponenttipohjaisen kehityksen hyödyntämisessä on monia etuja. Näitä ovat tehokas monimutkaisuuden hallinta, nopeampi valmis tuote, korkeampi tuottavuus, parempi laatu, suurempi johdonmukaisuus ja laajempi käytettävyys [25]. Komponenttipohjaisessa kehityksessä on kuitenkin myös riskejä ja haasteita, jotka pitää ottaa huomioon. Näitä on listattuna alle.

Uudelleenkäytettävien komponenttien kehittäminen vaatii enemmän aikaa ja vaivaa [6] kuin kehittäminen ilman huolta uudelleenkäytettävyydestä. Kiireellisissä projekteissa uudelleenkäytettävyydestä on helppo luopua, vaikka siitä olisi jatkossa merkittävääkin hyötyä.

Yksi suurimmista ongelmista ohjelmistotuotannossa on epäselvät, monitulkintaiset ja keskeneräiset vaatimusmäärittelyt. Komponenttien kehityksessä tästä aiheutuvat ongelmat ovat vielä suurempia. Ongelma kohdistuu sekä toiminnallisiin että ei-toiminnallisuus vaatimuksiin. Kehitettävän komponentin vaadittuja ominaisuuksia on vaikea tunnistaa ja täten sen tekeminen onnistuneesti on vaikeampaa. [26]

Komponentin kehityksessä käytettävyyden ja uudelleenkäytettävyyden välillä on ristiriita. Jotta komponentti olisi mahdollisimman uudelleenkäytettävä, sen pitäisi olla yleiskäyttöinen, laajennettava ja mukautuva. Toteuttaakseen nämä vaatimukset komponentista pitää tehdä monimutkaisempi. Monimutkaisuuden lisääntyessä komponentin käytettävyys heikkenee. Ongelmaa voidaan yrittää ratkaista tekemällä komponentista korkeamman abstraktiotason toteutus. Vaikka tällöin voidaan menettää hienosäädön mahdollisuus, saadaan toteutuksesta yksinkertaisempi. [6]

Komponenttien ylläpitokulut ovat korkeat. Vaikka koko järjestelmän ylläpitokustannuksia voidaan vähentää uudelleenkäytettävien komponenttien avulla, yksittäisen komponentin pitää toimia erilaisilla vaatimuksilla erilaisissa ohjelmistoissa erilaisissa ympäristöissä. Lisäksi komponentilta saatetaan vaatia erityyppistä luotettavuutta ja ylläpitotukea. [15]

Komponentit ovat herkkiä muutokselle. Komponenteilla ja ohjelmistoilla on eri elinkaaret ja vaatimukset. Tällöin on mahdollista, että komponentti ei täysin vastaa tietyn ohjelmiston tiettyjä vaatimuksia. Komponentilla voi myös olla ominaisuuksia, joita ohjelmiston kehittäjä ei tunne. Lisäksi vaikka yksittäinen komponentti voi sopia hyvin järjestelmään, eri komponentit eivät välttämättä toimi yhdessä. Tällöin ohjelmiston tasolla tehdyt muutokset, kuten käyttöjärjestelmän päivitys, toisen komponentin päivitys tai muutokset ohjelmistossa, ovat iso riski ja voivat johtaa koko järjestelmän kaatumiseen. [27]

Kriittisissä järjestelmissä, joissa turvallisuus- ja laatuvaatimukset ovat korkeat, komponenttipohjainen kehitys on erityisen hankalaa. Komponenttien laadun ja ei-toiminnallisten vaatimusten varmistaminen voi olla lähes mahdotonta eikä kyseisiä komponentteja voida tällöin käyttää. [11, s. xxxiii]

Haasteet pitää ottaa huomioon komponenttipohjaista kehitystä tehdessä. Ongelmien välttämiseksi lähestymistavan pitää olla systemaattinen prosessi- ja teknologiatasoilla [11, s. xxix].

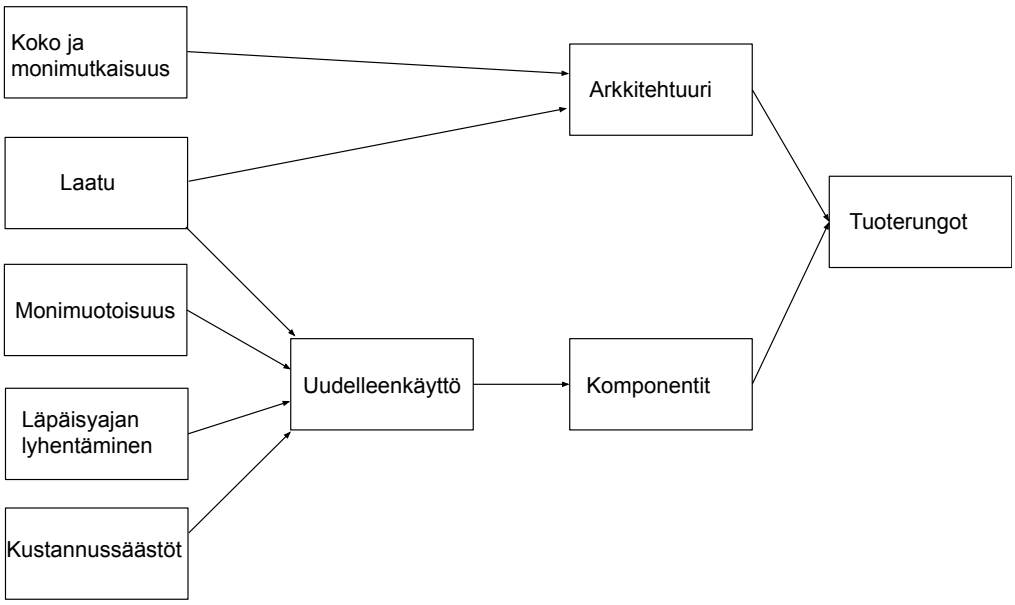
3.10 Tuoterungot

Tuoterunkoarkkitehtuuri on proaktiivinen, järjestelmällinen, suunniteltu ja organisoitu lähestymistapa tehdä uudelleenkäytettäviä komponentteja organisaation sisällä. Kuvassa 3.1 esitetään syitä, jotka johtavat tuoterunkojen käyttöön. Monimutkaisuutta ja laatua on paras hallita tarkalla arkkitehtuurilla. Laatua, monimuotoisuutta ja läpäisyajan lyhentämistä saavutetaan komponenttien uudelleenkäytöllä. [11, s. 207-208]

Arkkitehtuurivetoisen ylhäältä alas (engl. top-down) -tyyppisen ohjelmistokehityksen ja alhaalta ylös (engl. bottom-up) -tyyppisen komponentteja koostavan ohjelmistokehityksen välillä on herkkä tasapaino. Käytettävä ratkaisu riippuu tuotteelle vaaditusta monimuotoisuudesta. Pienille tuoteperheille tuoterunkoarkkitehtuurin käyttö saattaa muistuttaa yksittäisen tuotteen kehitystä. Isoille tuoteperheille kehitys voi sisältää monia elementtejä alhaalta ylös -tyyppisestä komponenttien koostamisesta. [11, s. 208].

Tuoterunkojen käytön hyödyt ovat hyvin samantyyppisiä kuin muussakin lähdekoodin uudelleenkäytössä. Kehityskustannuksia saadaan alaspäin. Kun tuoterunkoa varten kehitettyjä komponentteja käytetään useissa järjestelmissä, kustannukset vähenevät jokaisen järjestelmän osalta. Komponenttien laatu kasvaa, kun niitä arvioidaan ja testataan erilaisissa tuotteissa. Läpäisyaika on alussa pidempi kuin yksittäisen tuotteen kehityksessä, kun yhteisiä komponentteja rakennetaan. Tämän jälkeen läpäisyaika kuitenkin lyhenee, kun komponentteja voidaan uudelleenkäyttää jokaisella uudella tuotteella. [28, s. 9-10]

Ylläpito helpottuu, koska komponenttiin tehty korjaus voidaan levittää kaikkiin tuotteisiin, joissa komponentti on käytössä. Parhaimmillaan ylläpitohenkilöstön ei tarvitse tietää yksittäisten tuotteiden ja niiden osien yksityiskohtia, mikä vähentää opettelun vaivaa. Komponentin muuttuessa jokainen tuote pitää kuitenkin testata erikseen. Testauksessa voidaan kuitenkin myös soveltaa tuoterunkoarkkitehtuurin periaatteita, mikä vähentää tes-



Kuva 3.2. *Tuoterunkojen käyttöön ajavia tekijöitä. Perustuu lähteeseen [11, s. 208].*

tien ylläpidon vaivaa. [28, s. 11]

Yhteisten osien käyttö tuoterungossa vähentää monimutkaisuutta merkittävästi. Alusta tarjoaa rakenteen, joka päättää mitä komponentteja voidaan uudelleenkäyttää missäkin paikassa määrittämällä muuttuvuuden tiettyihin kohtiin. [28, s. 12]

4 MYYNTI- JA OSTOLASKUPROSESSI

Taloushallinnon prosesseja ovat ostolaskuprosessi, myyntilaskuprosessi, matka- ja kululaskuprosessi, kassanhallinta ja maksuliikenne, käyttöomaisuuskirjanpito, pääkirjanpitoprosessi ja raportointiprosessi. Edellä mainitut prosessit liittyvät toisiinsa pääkirjanpidon kautta. Yhdessä nämä muodostavat taloushallinnon kokonaisuuden. [29, s. 93-94] Tässä luvussa käydään läpi tämän työn kannalta merkittävät osto- ja myyntilaskuprosessit, kun ne tehdään sähköisesti.

4.1 Myyntilaskuprosessi

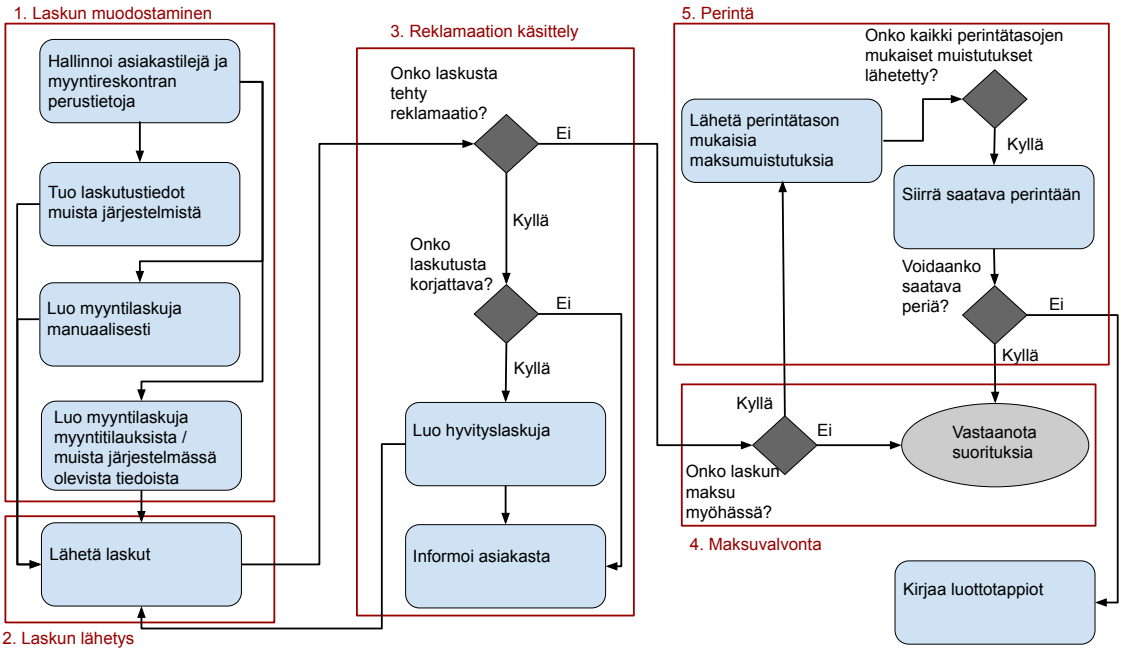
Laskutus on monelle yritykselle kriittinen osa yrityksen toimintaa. Virheet tai viiveet laskutusprosessissa voivat heikentää yrityksen maksuvalmiutta ja vaarantaa täten yrityksen koko toiminnan. Lisäksi laskutus näkyy ulospäin yrityksen asiakkaille ja on täten osa yrityksen imagoa ja asiakaspalvelua. [29, s. 120]

Myyntilaskuprosessi kattaa vaiheet myyntitilauksesta laskutukseen sekä maksuvalvonnasta ja perinnästä maksusuorituksen käsittelyyn. Myyntilaskutuksen kokonaisprosessia kutsutaan termillä ”tilauksesta kassaan” tai ”Order to Cash”. [29, s. 93] Kokonaisprosessi käynnistyy laskun laatimisesta. Prosessi päättyy siihen, kun laskun vastaanottajan maksusuoritus on kohdistettu myyntireskontraan ja maksusuoritukseen liittyvät kirjaukset näkyvät pääkirjanpidossa. Laskun laatimista voi edeltää esimerkiksi tarjouspyynnön vastaanotto, tarjouksen hinnoittelu ja toimitus asiakkaalle sekä myyntitilauksen vastaanotto ja vahvistaminen. Nämä toimenpiteet on kuitenkin rajattu tämän prosessitarkastelun ulkopuolelle. Sähköisen myyntilaskutuksen kokonaisprosessin vaiheet on kuvattu kuvassa 4.1. [29, s. 121]

1. Laskun muodostaminen

Ennen kuin myyntilaskut voidaan lähettää sähköisesti, on ne ensin laadittava joko muodostamalla lasku järjestelmien sisältämän datan perusteella automaattisesti tai tallentamalla laskutiedot manuaalisesti. Tehokkuuden kannalta laskun laatimisvaiheessa on tärkeää saada tieto siirtymään automaattisesti ja oikeellisena tiedon alkulähteiltä laskulle ja välttää saman tiedon käsittelyä useaan kertaan. Tämän saavuttamiseksi asiakas-, sopimus- ja hinnoittelutietojen on oltava järjestelmässä oikeellisina. Tiedon alkulähteenä toimii yleensä jokin esijärjestelmä. [29, s. 122-123]

Laskutukseen tietoa syöttäviä esijärjestelmiä ovat: [29, s. 123]



Kuva 4.1. *Laskutus- ja perintäprosessi. Perustuu lähteeseen [29, s. 121]*

- myyntitilausjärjestelmät
- projektiohjausjärjestelmät
- sopimustietokannat
- operatiivisen liiketoiminnan ohjausjärjestelmät (esimerkiksi puhelinoperaattorin omat puheluseurantajärjestelmät tai asiantuntijaorganisaatioiden tuntiseurantajärjestelmät)
- toiset laskutusjärjestelmät.

Esijärjestelmien suuren määrän vuoksi laskutusjärjestelmät sisältävät yleensä useita integraatioita toisiin järjestelmiin. Integraatiot voivat olla myös kaksisuuntaisia. Tällöin laskutusjärjestelmä lähettää tietoa takaisin esijärjestelmään esimerkiksi laskun tullessa maksetuksi.

Käytännössä laskutuksessa vallitsee kaksi päälinjaa. Joko laskut generoidaan esijärjestelmistä ja mahdollisesti lasku myös lähetetään asiakkaille sieltä. Kaikissa esijärjestelmissä laskun lähettäminen ei kuitenkaan ole mahdollista. Laskutusdata voidaan myös siirtää pääjärjestelmään, jossa laskut luodaan ja lähetetään. Yleensä laskujen lähetyksen keskittäminen kannattaa siksi, ettei tarvitse ylläpitää useita laskujen lähetysliittymiä ja siksi, että keskitetyssä myyntireskontrassa saamisten seuranta sekä asiakkaiden selvityspyyntöjen hoito on helpompaa, kun samassa järjestelmässä on nähtävillä kaikkien laskujen tiedot. [29, s. 124]

Käytännössä yrityksen liiketoiminta määrittää hyvin pitkälle sen, minkälainen laskun laatimisprosessi yrityksellä on. Suunniteltaessa prosessin vaiheita on välttämätöntä ymmärtää yrityksen liiketoiminta, sen vaikutus laskutusprosessiin ja vaatimukset laskutusjärjestelmälle. [29, s. 123]

Myyntilaskuprosessiin ja erityisesti laskun muodostamiseen liittyy myös asiakkuudenhallinta. Asiakas ja asiakkaan perustiedot ovat olennainen osa laskutusprosessia. Asiakastietojen ylläpito ja hallinta voidaan hoitaa järjestelmämielessä usealla eri tavalla. Käytettävät ratkaisut vaihtelevat yrityksien ja yritysten käyttämien järjestelmien perusteella. Olennaista on varmistaa, että tietojen ylläpidossa samaa tietoa ei tarvitse päivittää manuaalisesti useaan eri järjestelmään. [29, s. 124]

2. Laskun lähetys

Sähköisessä myyntilaskuprosessissa lasku lähetetään verkkolaskuna. Verkkolaskut lähetetään joko verkkolaskuoperaattorien tai pankkien välityksellä. Jos vastaanottajalla ei ole valmiuksia ottaa vastaan verkkolaskuja, operaattorit voivat yleensä tulostuspalvelun avulla tulostaa laskun paperille ja lähettää sen vastaanottajalle postitse. [30]

3. Reklamaation käsittely

Virheellisestä laskusta voi tulla asiakkaalta reklamaatio. Lasku voidaan hyvittää osittain tai kokonaan, jolloin laskusta syntyy hyvityslasku. Hyvityslasku voidaan luoda joko esijärjestelmässä tai pääjärjestelmässä riippuen yrityksen laskutusprosessissa. Hyvityslaskun lähetys tapahtuu samalla tavalla kuin normaalin laskun lähetys.

4. Maksuvalvonta

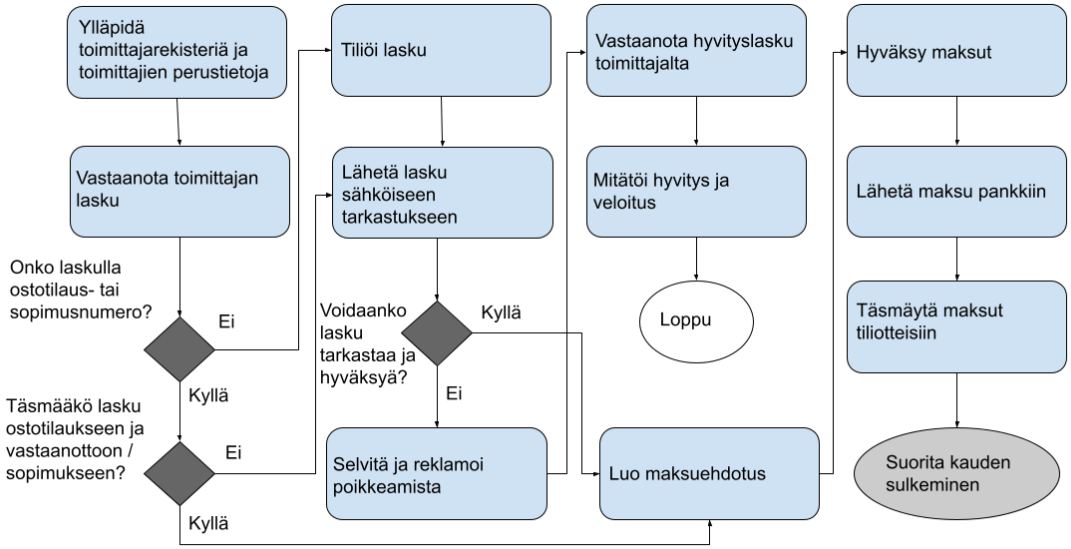
Laskutusjärjestelmät muodostavat valmiista laskusta myyntireskontratapahtuman maksuvalvontaa varten sekä pääkirjanpidon kirjaukset. Myyntireskontran tehtävä on pitää rekisteriä myyntilaskuista ja niiden tilasta. Myyntireskontran päävaiheisiin kuuluu suoritusten kohdistaminen, avointen laskujen seuraaminen ja mahdolliset perintätoimenpiteet. [29, s. 130]

Suoritusten kohdistamisessa hyödynnetään viitenumerojärjestelmää. Laskulle luodaan sitä muodostettaessa uniikki viitenumero. Suoritus kohdistetaan viitenumeron perusteella oikealle laskulle. Mikäli asiakas on maksanut oikealla viitenumerolla, kohdistus voidaan tehdä käytännössä täysin automaattisesti viiteaineiston perusteella. [29, s. 130]

5. Perintä

Mikäli myyntilaskuun kohdistuva suoritus saapuu ajallaan eräpäivään mennessä, on myyntireskontraprosessi kyseisen laskun osalta päättynyt [29, s. 131]. Mikäli asiakas ei maksa laskua eräpäivään mennessä, ryhdytään yleensä perintätoimenpiteisiin maksun saamiseksi.

Ensimmäinen perintätoimenpide on yleensä maksumuistutuksen lähettäminen asiakkaalle. Maksumuistutuksien lähettäminen ei vaadi erillistä lupaa perintätoiminnan harjoittamisesta. Laskun lähettänyt yritys voi siis lähettää muistutuksia omissa nimissä. Myyntireskontraohjelmissa onkin yleensä toiminnallisuus maksumuistutusten muodostamiseksi ja lähettämiseksi [29, s. 131-132]. Nopeimmillaan yritykset lähettävät maksumuistutuksen jo muutaman päivän päästä laskun eräpäivästä. Maksumuistutuksia lähetetään yleensä yksi ennen seuraavaa perinnän vaihetta, mutta niitä voidaan lähettää myös useita.



Kuva 4.2. *Ostolaskuprosessi. Perustuu lähteeseen [29, s. 99]*

Maksumuistutuksia lähetetään yleensä yritysasiakkaille nopeammin ja tiheämmin kuin kuluttaja-asiakkaille, koska laki [31] rajoittaa kuluttajaperintää huomattavasti enemmän. Laki myös asettaa rajat perinnän prosessille ja aikatauluille, mikä tulee ottaa huomioon.

Mikäli muistutuksista huolimatta laskuun ei saada suoritusta, siirrytään prosessissa varsinaiseen perintävaiheeseen. Muistutuksen jälkeiset perintätoimenpiteet vaativat luvan perintätoiminnan harjoittamiseen, joten monet yritykset hyödyntävät perinnässä tähän erikoistuneita perintätoimistoja. Tällöin laskut voidaan siirtää myyntireskontrasta integraation avulla perintäpalveluntarjoajan järjestelmään. [29, s. 132] Perintätoimistot tarjoavat palveluita myös koko myyntilaskuprosessin hallintaan. Lasku voidaan luoda suoraan perintäpalveluntarjoajan järjestelmässä tai se voidaan siirtää järjestelmään mihin tahansa edellä mainittuun vaiheeseen (laskun lähetys, maksuvalvonta, maksumuistutus tai perintä), josta prosessi jatkuu.

4.2 Ostolaskuprosessi

Ostolaskuprosessi sisältää vaiheet ostoehdotuksesta tai ostotilauksesta ostolaskun maksuun. Ostolaskun kokonaisprosessia kuvataankin usein termillä ”ostosta maksuun” tai "Procure to Pay”. Prosessiin voi sisältyä myös ostosopimusten hallintaa ja tavaran tai palvelun vastaanottotapahtumia. [29, s. 93]

Sähköisen ostolaskuprosessin vaiheet on kuvattu kuvassa 4.2.

1. Ostolasku vastaanotetaan verkkolaskuna tai skannattuna. Laskun perustiedot tallennetaan.
2. Ostolasku kohdistetaan ostotilaukseen tai ostosopimukseen, jos se liittyy tilaukseen

tai sopimukseen.

3. Ostolasku tiliöidään tilauksen, sopimuksen tai muiden laskutietojen pohjalta.
4. Ostolasku tarkastetaan ja hyväksytään joko tilausta tai sopimusta vastaan automaattisesti tai sen tekee tilaaja ja hyväksyjä itse. Tarvittaessa laskusta reklamoidaan toimittajalle.
5. Hyväksytyt ostolaskut kirjautuvat ostoreskontraan ja kirjanpitoon.
6. Ostoreskontrasta muodostetaan maksuaineisto, joka lähetetään pankkiin. Maksut kuitataan pankista saadun tiliotteen tai palautusaineiston perusteella.

Ostolaskujen käsittelyjärjestelmän päätehtäväksi voidaan määritellä "mahdollistaa laskun vastaanotto, tiliöinti, mahdollinen täsmäytys tilaukseen tai sopimukseen, hyväksyntä sekä koko prosessin hallinta". Näiden toimenpiteiden jälkeen lasku päivitetään ostoreskontraan, josta se kirjautuu pääkirjanpitoon ja on maksettavissa toimittajalle. [29, s. 104]

Ostolasku vastaanotetaan sähköiseen käsittelyjärjestelmään joko verkkolaskulta tai paperilaskun skannauksen kautta ja laskulle tallennetaan valmiiksi perustiedot. Ostoreskontran tehtäväksi jää tietojen tarkistus, tiliöinti sekä laskun lähettäminen hyväksymiskiertoon. Edeltävät työvaiheet ovat yleensä täysin tai osittain automatisoitavissa ostolaskujärjestelmän toiminnoilla. Automaatiota voidaan tehdä manuaalisesti asetetuilla tiliöintisäännöillä. Tiliöintisääntöjä voidaan myös luoda automaattisesti koneoppimisella. Koneoppiminen ja tiliöintisäännöt vaativat suuren määrän laskuja, jotta niitä kannattaa hyödyntää. Pienissä laskumäärissä sääntöjen luonti ja ylläpito ovat työläitä verrattuna niistä saatavaan hyötyyn. [29, s. 104-105]

Ostolaskujen hyväksymiskierto ostolaskujen käsittelyjärjestelmässä tarkoittaa prosessia, jossa lasku ensin asiatarkastetaan esimerkiksi tavaran tai palvelun tilaajan toimesta ja sen jälkeen lasku hyväksytään mahdollisesti toisen henkilön, kuten tilaajan esimiehen, toimesta [30][29, s. 107]. Yksinkertaisimmassa hyväksymiskierrossa laskun asiatarkastaja ja hyväksyjä on sama henkilö. Monimutkaisemmissa suurten yritysten käyttämissä hyväksymiskierroissa sekä asiatarkastajia että hyväksyjiä voi olla useita.

5 YMPÄRISTÖ

5.1 Django

Django on Python-pohjainen web-ohjelmistokehys. Django on ilmainen ja sen lähdekoodi on avointa [21]. Django on suunniteltu käytettäväksi yhdessä SQL-tietokannan kanssa. Django noudattaa muiden web-ohjelmistokehysten tavoin pääosin MVC-arkkitehtuuria [32][33, s. 16]. Django suhtautuu MVC-arkkitehtuuriin kuitenkin käytännönläheisesti eikä pakota tiukkoja rooleja malleille, kontrollereille ja näkymille [33, s. 16].

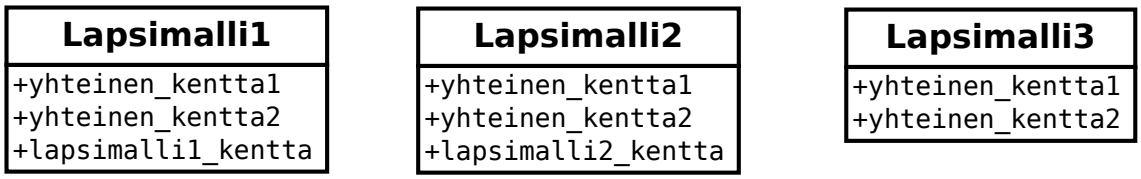
Django-mallit sisältävät järjestelmään säilöttävän datan keskeisimmät kentät ja käyttäytymisen eli ne toimivat rajapintana tietokannalle. Pohjimmiltaan Django-malli on Pythonin luokka, johon Django on lisännyt tietokantakäsittelyyn liittyviä ominaisuuksia. Yleisesti ottaen yksittäinen Django-malli kuvaa yhtä tietokannan taulua. Mallin jokainen attribuutti vastaa yhtä tietokannan kenttää. Mallista muodostettu instanssi taasen kuvastaa tiettyä tietokannan taulun riviä. Djangossa on myös tietokantakyselyt abstrahoiva rajapinta, jonka avulla dataa voi luoda, noutaa, päivittää ja poistaa. [21]

Mallien ja oman tietokantarajapinnan avulla Django abstrahoi tietokannan. Tämän tyyppisestä abstrahoinnista käytetään nimitystä ORM (Object Relational Mapping). ORM:n avulla muutetaan tietokannan käsittely puhtaista SQL-kyselyistä oliopohjaiseksi.

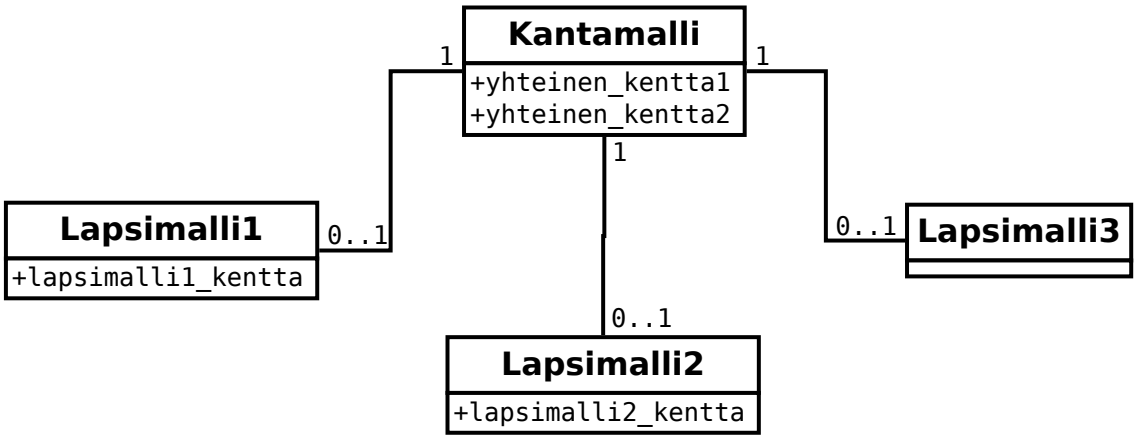
Mallien periyttäminen Djangossa toimii lähes kuten normaalien luokkien periytyminen Pythonissa. Periyttämisessä on kolme eri vaihtoehtoa liittyen siihen, halutaanko kantamallien (engl. parent model) olevan malleja itsessään omalla tietokantataululla vai vain yleisen tiedon säilöjiä, joihin pääsee käsiksi lapsimallien (engl. child model) kautta. [21] Merkittävä ero mallien periyttämisen ja pythonin normaalin luokan periyttämisen välillä on, että mallien tapauksessa Djangon ORM ei tue polymorfismia. Asian korjaamiseen on olemassa avoimen lähdekoodin ratkaisuja [34].

Abstraktit kantaluokat

Abstraktit kantaluokat ovat hyödyllisiä, kun kantaluokan halutaan säilyttävän yhteistä tietoa, jota ei haluta lisätä erikseen jokaiseen lapsimalliin. Abstraktia kantaluokkaa ei siis ole ikinä tarkoitus käyttää itsenäisesti. Abstraktista kantaluokasta ei luoda omaa tietokantataulua. Sen sijaan sitä käytetään kantaluokkana muille malleille, joihin kantaluokan kentät lisätään. Näin voidaan kasata yhteistä tietoa Pythonin tasolla samalla luoden yhden tietokantataulun yhtä lapsimallia kohden tietokannan tasolla. [21] Kuvassa 5.1 esitetään esimerkinomaisesti miltä abstraktien kantaluokkien käyttö näyttää tietokantatasolta



Kuva 5.1. *Tietokantakuvaus abstrakteja kantaluokkia käytettäessä.*



Kuva 5.2. *Tietokantakuvaus monitauluperiytymistä käytettäessä.*

kuvattuna. Kolmelle lapsimallille tehdään kolme eri taulua tietokantaan. Abstraktista kantaluokasta ei synny omaa taulua tietokantaan, vaan yhteiset kentät kopioidaan erikseen jokaiseen lapsimalliin.

Monitauluperiytyminen

Monitauluperiytymisessä (engl. Multi-table inheritance) jokainen periytymishierarkiassa oleva malli on malli itsessään. Käytännössä siis jokaisesta mallista luodaan tietokantaan oma taulu. Tällöin periytymissuhteen seurauksena lapsimallien ja kantamallien välille syntyy linkki, joka on käytännössä yhden suhde yhteen vierasavainviittaus tietokannassa. [21] Kuvassa 5.2 esitetään miltä monitauluperiytymisen käyttö näyttää tietokantatasolta kuvattuna. Kolmelle lapsimallille tehdään kolme eri taulua tietokantaan. Lisäksi kantamallille tehdään oma taulu, johon lapsimallit viittaavat.

Proxy-mallit

Proxy-malleja (engl. Proxy models) hyödyntävässä periyttämisessä ideana on, että alkuperäisestä mallista luodaan proxy-malli. Proxy-malleja voi luoda, poistaa ja päivittää. Data tallennetaan kuten alkuperäistä mallia käytettäessä. Proxy-malliin voi kuitenkin määrittää Pythonin tasolla uusia metodeita eikä näitä lisätä tällöin alkuperäiseen malliin. [21] Kuvassa 5.3 esitetään miltä proxy-mallien käyttö näyttää tietokantatasolta kuvattuna. Kolmelle lapsimallille tehdään vain yksi taulu, jossa on kaikkien kentät. Lisäksi tauluun voidaan lisätä kenttä tyypille, mikäli lapsimallien erottaminen tietokantatasolla on tarpeellista.

Kuva 5.3. *Tietokantakuvaus proxy-malleja käytettäessä.*

5.2 REST

5.2.1 Arkkitehtuurityyli

REST (REpresentational State Transfer) on Roy Fieldingin väitöskirjassaan esittelemä arkkitehtuurityyli hajautetuille hypermediajärjestelmille ja erityisesti web-pohjaisille arkkitehtuureille. REST määritellään joukkona rajoitteita, jotka lisätään yksitellen toistensa päälle aloittaen nollajoukosta. Arkkitehtuurillisesta näkökulmasta nollajoukko on järjestelmä, jossa komponenttien välillä ei ole selkeitä rajoja. [35]

Asiakas-palvelin

Asiakas-palvelin-rajoitteen (engl. client-server) periaate on huolenaiheiden erottaminen (engl. separation of concerns). Erottamalla käyttöliittymä (asiakas) tietovarastosta (palvelin) parannetaan käyttöliittymän siirrettävyyttä eri alustoille. Samalla skaalautuvuus paranee, kun palvelimen komponentit yksinkertaistuvat. Rajoite tarjoaa myös kehitykseen joustavuutta, koska asiakassovellusta voidaan kehittää ilman, että se vaikuttaa palvelimen toteutukseen ja toisinpäin. [35]

Tilattomuus

Asiakas-palvelin-rajoitteen päälle lisätään tilattomuuden rajoite. Asiakkaan ja palvelimen välisen kommunikaation pitää olla tilatonta. Tämä tarkoittaa, että jokaisessa asiakkaalta lähtevässä pyynnössä pitää olla kaikki tarvittava tieto pyynnön käsittelyyn palvelimen päässä ilman, että palvelimen tarvitsee hyödyntää sinne tallennettuja tietoja. Tilattomuuden rajoitteen ansiosta arkkitehtuuri saa monia hyödyllisiä ominaisuuksia: [35]

- Näkyvyys: Järjestelmän monitorointi helpottuu, koska kaikki tieto on pyynnön sisällä.
- Luotettavuus: Tilaton järjestelmä helpottaa virheistä toipumista [36].
- Skaalautuvuus: Koska tietoa ei tarvitse tallentaa pyyntöjen välillä, palvelin voi vapauttaa resursseja nopeammin.

Tilattomuuden rajoitteen heikkous on verkon suorituskyvyn heikentyminen, koska pyynnön mukaan liitetään aina tilatieto. Tilatiedon lisääminen olisi mahdollista välttää, mikäli tilatieto voitaisiin tallentaa palvelimelle. [35]

Välimuisti

Välimuistirajoitteet parantavat verkon suorituskykyä. Välimuistirajoitteet vaativat, että jokainen palvelimen vastaus sisältää tiedon voiko asiakas laittaa vastauksen välimuistiin. Jos vastauksen voi laittaa välimuistiin, asiakas voi hyödyntää vastauksen tietoa myöhempiin vastaavanlaisiin pyyntöihin. Välimuistin käytön heikkous on luotettavuuden väheneminen, jos välimuistissa oleva tieto on vanhentunutta. [35]

Yhtenäinen rajapinta

Merkittävä erottava tekijä REST-arkkitehtuurityylin ja muiden verkkoihin keskittyvien arkkitehtuurityylien välillä on yhtenäisen rajapinnan painotus. Vaatimalla yhtenäinen rajapinta komponenttien välillä koko järjestelmän arkkitehtuuri yksinkertaistuu ja komponenttien välisen vuorovaikutuksen näkyvyys paranee. Asiakaspään toteutus saadaan erotettua itsenäiseksi palvelinpäästä, mikä edistää molempien itsenäistä kehitystä. Yhtenäisen rajapinnan heikkoutena on, että standardisoitu ja yhtenäinen rajapinta voi heikentää tehokuutta, koska asiakas ei voi käyttää itselleen sopivinta ja optimoiduinta rajapintaa. [35]

Kerroksinen järjestelmä

Kerroksisen järjestelmän rajoitteilla pyritään erottamaan komponentit eri hierarkiakerroksiin. Komponentin vuorovaikutus rajoitetaan pelkästään sen lähikerroksiin. Rajoittamalla komponentin tietoisuus järjestelmästä yhdelle kerrokselle koko järjestelmän monimutkaisuus vähenee ja lisäksi komponenttien välisiä kytkentöjä kyetään rajoittamaan. Kerroksien avulla voidaan esimerkiksi kapseloida legacy-palveluita ja suojella uusia palveluita legacy-asiakkailta. [35] Kerroksisen järjestelmän suurin heikkous on tiedon prosessointiin tuleva overhead ja latenssi [37].

Koodia tilauksesta

Viimeinen lisättävä rajoite on koodia tilauksesta (engl. code-on-demand). Rajoite mahdollistaa asiakkaiden lataavan ja suorittavan koodia, jota palvelin tarjoaa. Tämä yksinkertaistaa asiakkaita, koska niiden ei täydy toteuttaa kaikkia ominaisuuksia etukäteen. Rajoite kuitenkin heikentää järjestelmän yleistä näkyvyyttä ja on sen vuoksi REST-arkkitehtuurityylissä vapaaehtoinen. [35]

5.2.2 REST-rajapinta

REST-rajapinta soveltaa REST-arkkitehtuurityylin periaatteita. REST-rajapinnat on suunniteltu HTTP-protokollalla käytettäväksi [38].

Resurssit

REST-rajapinnat koostuvat resursseista. Resurssi voi olla mitä tahansa minkä voi nimetä (web-sivu, kuva, henkilö, raportti). Resurssit määrittävät mitä palveluita on tarjolla, minkälaista tietoa välitetään ja mitä toimintoja niihin liittyy. [38]

Representaatio

Representaatio on resurssin sisältämän datan esitystapa (binääri, JSON, XML). Yksittäi-

Taulukko 5.1. *HTTP:n verbit ja niihin liittyvät REST-rajapinnan toiminnot. Perustuu lähteeseen [38]*

HTTP-verbi	Toiminto
GET	Hakee resurssin
POST	Lisää resurssin
PUT	Päivittää resurssia
DELETE	Poistaa resurssin
HEAD	Kysyy onko resurssi olemassa ilman, että palautetaan mitään sen representaatioista
OPTIONS	Palauttaa listan sallituista toiminnoista tietylle resurssille

sellä resurssilla voi olla monta eri representaatiota. [38]

Resurssin tunniste

Resurssin tunnisteen pitää olla uniikki tapa tunnistaa resurssi millä tahansa hetkellä ja sen pitäisi tarjota täysi polku resurssiin. HTTP-protokollaa käytettäessä se on käytännössä täysi URI (Unique Resource Identifier). [38]

Toiminnot

REST-rajapinnan kanssa kommunikointi perustuu toimintoihin, joita HTTP-protokollan verbit tarjoavat [38]. Taulukossa 5.1 kuvataan mitä toimintoa mikäkin verbi vastaa.

5.3 Anitta

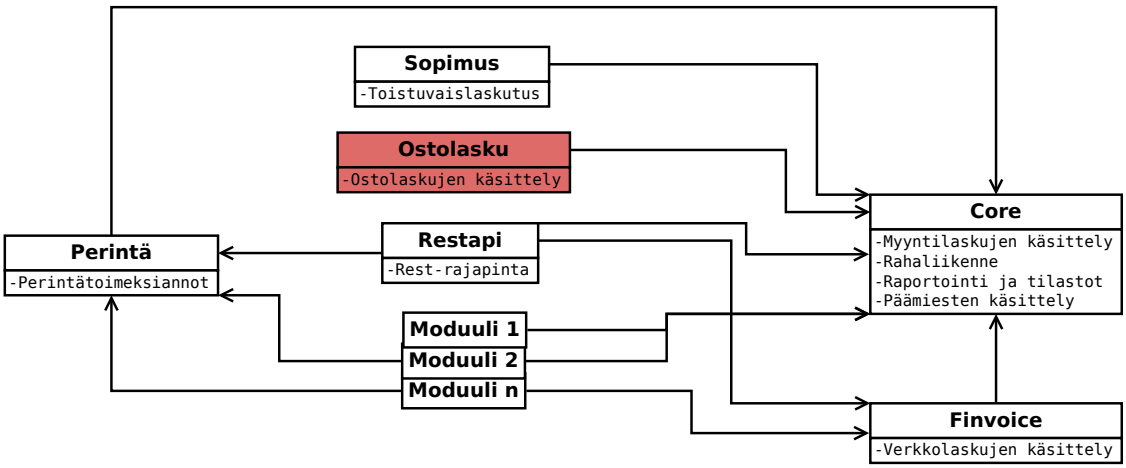
5.3.1 Yleistä

Anitta on suomalaisten perintätoimistojen käyttöön suunniteltu laskutus- ja perintäjärjestelmä. Anitta on web-pohjainen ja se on suunniteltu Internet-selaimella käytettäväksi. Anitta on toteutettu Django-web-ohjelmistokehyksellä.

Anitta on hajautettu kymmeniin erilaisiin moduuleihin. Keskeisimmät moduulit ovat core ja perintä, joista suurin osa muista moduuleista riippuu. Kuvassa 5.4 esitetään typistetty kuvaus Anittan moduuleista ja niiden riippuvuudesta toisistaan. Tässä työssä toteuttava Ostolasku-moduuli on merkitty punaisella. Kuvassa moduuleita on esimerkinomaisesti vain muutama.

Tämän työn kannalta merkittävin moduuli on core. Core sisältää järjestelmän keskeisimmät toiminnot sekä järjestelmän kannalta yleiskäyttöiset luokat ja funktiot. Ostolasku-moduulissa tullaan hyödyntämään osittain coreen toteutettuja laskun tietomallia ja laskun käsittelyyn liittyviä metodeita. Rahaliikenteen puolelta tullaan hyödyntämään WS-rajapintatoteutusta tiedonsiirtoon pankkien kanssa. Lisäksi tullaan hyödyntämään maksutapahtumiin liittyvää logiikkaa.

Anittan toteutuksessa on pieniltä osin hyödynnetty tuoterunkoarkkitehtuuria. Pääosin Anit-



Kuva 5.4. *Anittan moduulit ja niiden riippuvuus toisistaan.*

ta on kuitenkin asiakaskohtaisesti konfiguroitava tuote. Eri asiakkaiden vaatimukset ja käyttötapaukset eivät eroa merkittävästi toisistaan, sillä Suomessa laskutuksen ja perinnän toimintaa ohjaavat lait ja standardit. Perintätoimintaa ohjaavat muun muassa perintälaki [31] ja korkolaki [39]. Laskutustoimintaa määrittävät verkkolaskutuksen standardit. Verohallinnon mukaan sähköisellä laskulla eli verkkolaskulla "tarkoitetaan laskua, joka annetaan ja vastaanotetaan sähköisessä muodossa"[40]. Verkkolaskutuksesta on EU-direktiivi 2014/55/EU [41], joka määrittää verkkolaskun tietosisällön. Lisäksi Suomessa laskutuksen toiminnassa on merkittävässä osassa Finvoice-esitystapa ja Finvoice-välityspalvelu. Finvoice-esitystapa on xml-muotoinen verkkolaskulaskukuvaus [42]. Uusin Finvoice-versio 3.0 huomioi EU-direktiivin 2015/55 muutokset [42]. Finvoice-esitystavasta käytetään usein termiä Finvoice-formaatti. Finvoice-välityspalvelussa välitetään Finvoice-formaatin mukaisia aineistoja [43]. Finvoice-formaattia käytetään myös yleisesti muussakin kuin Finvoice-välityspalvelun kautta tapahtuvassa laskujen sähköisessä siirrossa.

5.3.2 REST-rajapinnat

Kaikki Anittan REST-rajapinnat toimivat JSON-formaatilla. REST-rajapinnat on toteutettu avoimen lähdekoodin Django REST framework -kirjaston avulla [44].

Anittassa on REST-rajapinnat myyntilaskun hakemiseen, lisäämiseen ja muokkaamiseen. Myyntilaskun lisäämiselle on kaksi eri REST-rajapintaa. Toisen rajapinnan myyntilaskun representaatio perustuu Anittan myyntilaskun tietomalliin. Myyntilaskun tiedot annetaan suoraan JSONin kenttinä. Toisen rajapinnan myyntilaskun representaatio perustuu taasen Finvoice-formaattiin. Finvoice-xml annetaan tällöin yhdessä JSONin kentässä. Anittan tietomalliin perustuvaan rajapintaan on lähettävän järjestelmän helpompi tehdä integraatio, jos integraatio rakennetaan tyhjästä. Lähettävissä järjestelmissä on usein kuitenkin Finvoice-xml:n muodostus valmiina, jolloin Finvoice-formaattiin perustuvan rajapinnan käyttö on helpompaa.

Anittassa on REST-rajapintoja myös myyntilaskuihin liittyvien keskeisimpien toimenpiteiden suorittamiseen. Näitä ovat laskun perinnän estäminen ja salliminen, laskun keskeyttäminen ja jatkaminen, laskun hyvittäminen ja laskun eräpäivän siirto.

5.3.3 Myyntilaskuprosessi

Yleistä sähköistä myyntilaskuprosessia on käsitelty luvussa 4.1. Tässä luvussa kuvataan sähköinen myyntilaskuprosessi Anittassa.

Laskun muodostaminen

Anitta on täysivaltainen laskutusjärjestelmä eli siihen on mahdollista tallentaa kaikki laskutuksen kannalta olennainen tieto kuten asiakasrekisterit, tuoterekisterit ja laskulla näkyvät yleisimmät tiedot. Anittassa on myös mahdollista syöttää myyntilaskuja manuaalisesti. Suurin osa Anittaan tulevista myyntilaskuista tulee kuitenkin esijärjestelmistä. Esijärjestelmä voi syöttää myyntilaskuja Anittan REST-rajapintojen kautta tai vaihtoehtoisesti Anitta voi hakea myyntilaskuja ulkoisen järjestelmän REST-rajapinnan kautta. Lisäksi myyntilaskuja tulee Anittaan tiedostoina, joita syötetään Anittan käyttöliittymän kautta tai siirtoina Anittan palvelimelle SFTP:llä. Lähes kaikki tiedostopohjaiset siirrot ja osa REST-rajapinnoista käyttää Finvoice-formaattia. Myyntilaskujen tullessa ulkoisista järjestelmistä tietojen ylläpito tapahtuu aina ulkoisessa järjestelmässä. Käytännössä uuden aineiston sisäänluvun yhteydessä Anitta ylikirjoittaa vanhat tiedot.

Laskun lähetys

Anitta lähettää verkkolaskut perintätoimiston kanssa sovitun verkkolaskuoperaattorin kautta. Myyntilaskuja voidaan lähettää myös sähköpostilla tai maapostina.

Maksuvalvonta

Anitta seuraa myyntilaskujen tilaa pitämällä kirjaa niiden avoimesta summasta ja niihin kohdistuneista suorituksista. Anittassa ei ole varsinaista kirjanpitoa, vaan kirjanpito pitää tehdä erillisessä ohjelmassa. Käytännössä Anittasta on saatavilla myyntilaskuista ja suorituksista raportteja, jotka lähetetään kirjanpidon ohjelmaan.

Suoritukset haetaan pankeista päivittäin. Pankki kerää sisäänpäin tulevat maksut päiväkohtaisesti yhteen ja välittää tiedot tiliotteilla ja viitemaksutiedostoina [29, s. 132]. Anitta kohdistaa suoritukset viitenumeron perusteella.

Perintä

Mikäli todetaan, että perintätoimenpiteillä ei tulla saamaan saatavia perittyä, saatavat merkitään luottotappioiksi. Tämä on kirjanpidollinen toimenpide eli käytännössä luottotappioksi merkityistä saatavista luodaan raportti, joka viedään kirjanpidon ohjelmaan.

Tilitys

Myyntilaskuihin tulleet suoritukset maksetaan perintätoimiston asiakasvaratilille. Asiakasvaratililtä suoritukset tilitetään eteenpäin perintätoimiston asiakkaille. Tilityksistä muodos-

tetaan maksuaineisto, joka lähetetään pankkiin. Pankki tekee maksuaineiston sisältämät veloitukset perintätoimiston asiakasvaratililtä [29, s. 132].

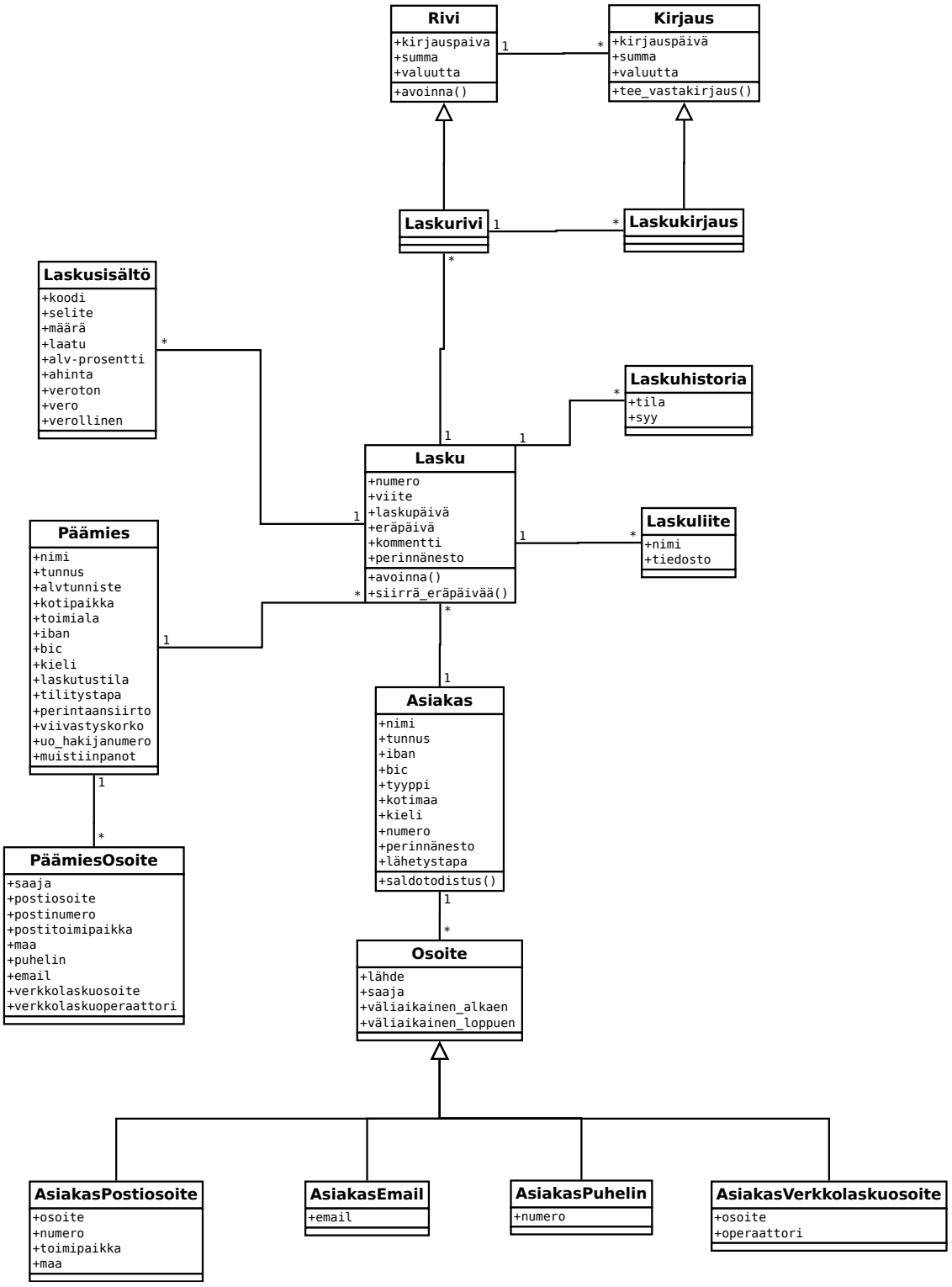
5.3.4 Myyntilaskun tietomalli

Anittan myyntilaskun tietomalli perustuu pääosin Verohallinnon ohjeeseen Laskutusvaatimukset arvonlisäverotuksessa [40] ja luvussa 5.3.1 mainittuun Finvoice-formaattiin. Lisäksi tietomallissa on yksittäisiä kenttiä, joita nämä määritykset eivät vaadi. Kuvassa 5.5 esitetään yksinkertaistettu kuvaus Anittan myyntilaskuun liittyvistä Django-malleista.

Päämies-malli vastaa järjestelmän käyttäjän yritystä eli myyntilaskun myyjää. Asiakas-malli vastaa myyntilaskun ostajaa. Sekä myyjälle että ostajalle tallennetaan mahdollisesti neljä erityyppistä osoitetta: postiosoite, sähköpostiosoite, verkkolaskuosoite ja puhelinnumero. Päämies-malliin liittyvät osoitteet tallennetaan samaan malliin. Asiakas-malliin liittyvät osoitteet tallennetaan erikseen omiin malleihin. Ero johtuu niiden erityyppisestä käsittelystä järjestelmässä.

Rivi- ja Kirjaus-mallit on suunniteltu yleiskäytettäviksi rahaliikenteen komponenteiksi. Rivimalli kuvastaa kirjanpidon riviä, johon odotetaan kirjattavaksi rivin summan mukaista rahamäärää. Riville tulevat kirjaukset tallennetaan Kirjaus-malliin. Jos kyseessä on pankkitapahtumaan liittyvä kirjaus, kirjaus viedään tilitettäväksi eteenpäin. Rivistä ja Kirjauksesta voidaan periyttää rahaliikenteen eri tyyppejä. Kuvassa näkyvän Laskurivin ja Laskukirjauksen lisäksi tällaisia tyyppejä ovat muun muassa Selvitysrivi ja -kirjaus, Perintäkulurivi ja -kirjaus ja Viivästyskorkorivi ja -kirjaus.

Lasku-malli vastaa myyntilaskua. Laskusisältö-malli vastaa laskuriviä laskun kuvalla. Laskusisältö-mallien verollisen summan perusteella lasketaan koko myyntilaskun summa. Myyntilaskusta muodostetaan laskun summaa vastaava laskurivi laskua luotaessa. Laskurivin yhteyteen syntyy Laskukirjaus, kun laskun ostaja maksaa laskun pankissa ja pankkiaineisto ladataan Anittaan sisään. Tämän jälkeen kirjaus tilitetään eteenpäin laskun myyjälle. Tähän liittyviä malleja ei ole kuvassa. Lasku voidaan myös peruuttaa, jolloin Laskurivin yhteyteen ei synny Laskukirjausta, tai hyvittää, jolloin Laskukirjaus luodaan, mutta sitä ei tilitetä eteenpäin.



Kuva 5.5. *Yksinkertaistettu kuvaus Anittan myyntilaskuun liittyvistä Django-malleista*

6 TOTEUTUS

Ostolaskumoduulin käyttöliittymä toteutettiin VueJS-JavaScript-ohjelmistokehyksellä [45]. Käyttöliittymä kommunikoi palvelimen kanssa REST-rajapinnan yli. Ostolaskumoduulin käyttöliittymän tarkempi toteutus on rajattu tämän työn ulkopuolelle, koska se ei ollut diplomityön kirjoittajan vastuulla. Lisäksi käyttöliittymän toteutuksessa koodiin uudelleenkäyttö on yksinkertaisempi ongelma kuin palvelinpään toteutuksessa.

6.1 Ostolaskuprosessin vaatimukset

Yleinen sähköinen ostolaskuprosessi on määritelty luvussa 4.2. Tässä luvussa käydään läpi toteutettavan ostolaskumoduulin ostolaskuprosessia vaatimusten kautta. Ostolaskumoduuli on alustavasti tarkoitettu lähinnä pienten yritysten käyttöön. Tällöin toiminnallisuudet voidaan pitää yksinkertaisina [29, s. 32]. Anittan muidenkin ominaisuuksien kehityksessä on havaittu hyväksi toimintamalliksi tehdä ensin yksinkertainen toteutus. Toiminnallisuuksia on mahdollista laajentaa suurempien yritysten käyttöön myöhemmin, jos tarvetta ilmenee.

Toteutettavaan moduuliin tulee ostolaskuja kahta reittiä. Ostolaskuja on mahdollista syöttää käyttöliittymän kautta. Suurin osa ostolaskuista tulee kuitenkin perintätoimiston yhteistyökumppanina toimivan verkkolaskuoperaattorin välittämänä Finvoice-formaatissa. Verkkolaskuoperaattori hoitaa suoraan verkkolaskuna tulevat ostolaskut sekä skannaa paperiset laskut sähköiseen muotoon.

Ostolaskujen tiliöintiin ei ole tulossa tässä vaiheessa mitään automatisointia, koska laskumäärät ovat pieniä. Käytännössä käyttäjä tiliöi jokaisen ostolaskun manuaalisesti. Koska Anittassa ei tehdä varsinaista kirjanpitoa, tiliöinnin ei tarvitse olla täsmällinen. Tiliöinnin perusteella luodaan raportti, joka lähetetään kirjanpito-ohjelmaan. Tiliöinnin automatisointi on iso työ ja vaatii asiantuntemusta, jota ei tällä hetkellä ole, joten sitä ei tässä vaiheessa otettu edes suunnittelussa huomioon. Tarkoituksena on kerätä ensin kokemuksia nykyisestä mallista ja miettiä automatisointia myöhemmin.

Ostolaskujen hyväksyntään ei tule alkuvaiheessa hyväksymisrooleja tai edes kaksiportaisuutta. Hyväksyntä on yksinkertainen ostolaskun tilan vaihto, jonka voi tehdä kuka tahansa käyttäjä, jolla on ylipäätään oikeus kyseisen ostolaskun käsittelyyn.

Hyväksytyt ostolaskut on mahdollista maksaa. Niistä muodostetaan maksuaineisto, joka lähetetään pankkiin. Pankista saadaan takaisin tiliote, jonka avulla maksujen onnistumi-

nen voidaan tarkistaa.

6.2 Osto- ja myyntilaskuprosessien yhteiset ominaisuudet

Anittan myyntilaskuprosessia ja toteutettavan ostolaskumoduulin prosessia tarkasteltaessa voidaan havaita, että niissä on paljon yhteistä. Ostolaskuprosessissa ylläpidetään toimittajarekisteriä ja myyntilaskuprosessissa ylläpidetään asiakasrekisteriä. Asiakkaan ja toimittajan tietomalli on lähes samanlainen. Asiakkaalla on toimittajalta puuttuvat tiedot laskun lähetysasetuksista ja perinnän tiedoista.

Osto- ja myyntilaskujen tietomallit ovat myös hyvin samanlaiset johtuen siitä, että reaalimaailmassa kysymys on samasta asiasta, laskusta. Osto- ja myyntilaskujen tiedonsiirrot perustuvat vahvasti Finvoice-formaattiin. Molempia tuodaan Anittaan Finvoice-formaatissa. Lisäksi myyntilaskuja viedään Finvoice-formaatissa. Ostolaskujen vienti Finvoice-formaatissa on myös tulevaisuudessa todennäköistä, vaikka sille ei ole tällä hetkellä tarvetta. Yhteisen laskukomponentin käyttö olisi järkevää, jotta myös Finvoice-konversioissa voidaan uudelleenkäyttää olemassa olevia toteutuksia.

Ostolaskun tiliöinti toimii Anittan näkökulmasta samalla tavalla kuin myyntilaskun luottotappioiden kirjaus. Molemmissa merkitään osa laskun summasta tai koko laskun summa tietylle kirjanpidon tilille. Tästä luodaan raportti kirjanpidon ohjelmaan vietäväksi.

Rahaliikenteen puolesta myynti- ja ostolaskut toimivat samalla tavalla. Molemmilla seurataan laskun avointa summaa. Laskuille kirjataan suorituksia. Myyntilaskujen suoritukset tulevat joko viitesuorituksina pankkiaineistosta tai erityyppisinä hyvityksinä. Ostolaskuille kirjataan suoritus, kun käyttäjä haluaa maksaa ostolaskun. Molemmissa tapauksissa laskulle kirjatuista suorituksista muodostetaan maksuaineisto, joka lähetetään pankkiin. Rahaliikenteen komponentit on jo suunniteltu yleiskäyttöisiksi.

Prosessien yhteiset ominaisuudet ovat vahva argumentti yhteisen laskukomponentin käytön puolesta. Taloushallinnon prosessit pysynevät samankaltaisina jatkossakin.

6.3 Koodin uudelleenkäyttö

Nykyisen myyntilaskutoteutuksen uudelleenkäyttö on pragmaattista uudelleenkäyttöä, koska sitä ei ole suunniteltu uudelleenkäytettäväksi. Pragmaattista uudelleenkäyttöä on käsitelty luvussa 2.1. Pragmaattiselle uudelleenkäytölle on kaksi vaihtoehtoa. Nykyistä toteutusta voidaan refaktoroida siten, että se toimii myös ostolaskuilla. Tällöin nykyisestä myyntilaskutoteutuksesta muokataan yleiskäyttöinen laskukomponentti, josta periytetään myyntilaskut ja ostolaskut. Toinen vaihtoehto on toteuttaa ostolaskumoduuli tyhjästä hyödyntäen myyntilaskutoteutusta kopioimalla ja muokkaamalla sitä.

Nykyinen myyntilaskutoteutus on tuotantokäytössä. Sen muokkaamisessa on riski aiheuttaa virheitä käytössä oleviin ominaisuuksiin. Lisäksi myyntilaskutoiminnallisuudet ovat järjestelmän ydintoimintaa, jolloin mahdollisten virheiden tapahtuminen saattaa pysäyt-

tää merkittävän osan järjestelmän toiminnasta. Myyntilaskupuolen toteutusyksityiskohdat ovat kuitenkin tarkasti tiedossa, joten riskit olemassa olevan koodin muokkaamisen suhteen ovat pienemmät.

Refaktorointi vaatii myös paljon resursseja. Jos ostolaskumoduuli toteutettaisiin kopioimalla ja muokkaamalla, saataisiin valmis tuote aikaiseksi nopeammin. Myyntilaskutoteutukselle on kuitenkin suunnitteilla myös muita isompia refaktorointitöitä. Tekemällä ne samalla kuin uudelleenkäyttöön liittyvät muutokset saadaan synergiahyötyjä ja näin säästettyä resursseja.

Yhtenäisen toteutuksen tavoitteena on kasvattaa tuotteen laatua vähentämällä virheitä ja helpottaa ylläpidettävyyttä myöhemmin. Laatua saadaan kasvatettua sekä ostolaskuettä myyntilaskutoiminnallisuuksiin. Varsinkin myyntilaskutoiminnallisuuksien laadun kasvattaminen on hyödyllistä toiminnallisuuksien tärkeyden vuoksi. Ylläpito nopeutuu, koska ylläpidettävää lähdekoodia on vähemmän. Yleiskäyttöisyysvaatimus kuitenkin vaatii ylläpitäjältä enemmän osaamista ja tuntemusta järjestelmästä.

Yleiskäyttöistä laskukomponenttia voitaisiin käyttää myös muun tyyppisille laskuille. Tällä hetkellä ei ole kuitenkaan tiedossa, että järjestelmään olisi tulevaisuudessa tulossa kolmatta tyyppiä. Osto- ja myyntilaskut kattavat tämänhetkiset käyttötarpeet.

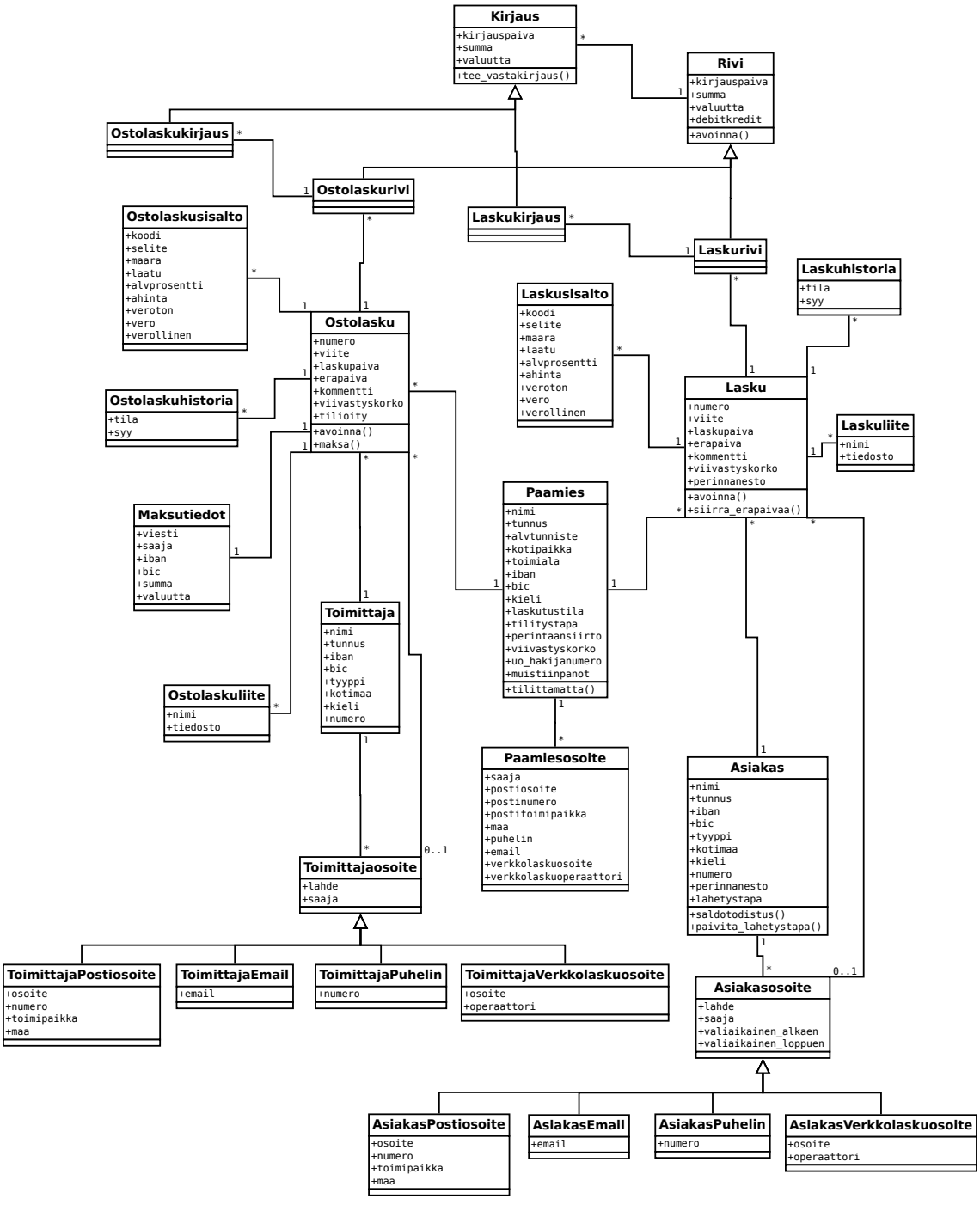
6.4 Django-mallit

Ostolaskumoduulin toteutuksessa merkittävin osuus on Django-mallien toteutus. Kaikki muu rakentuu mallien ja niiden metodien päälle. Tässä luvussa vertaillaan eri tapoja toteuttaa mallit. Ensin tarkastellaan mallien toteutusta, jos ne tehdään kopioimalla pohja olemassa olevista malleista. Tällöin ei siis muokata olemassa olevaa koodia ollenkaan. Sen jälkeen tarkastellaan mallien toteutusta, jos ne tehdään refaktoroimalla olemassa olevia malleja.

6.4.1 Kopioidaan olemassa olevista malleista

Kuvassa 6.1 esitetään yksinkertaistettu kuvaus Django-malleista, kun ostolaskumoduulin django-mallit toteutetaan erikseen ilman nykyisten mallien refaktorointia. Uusista Django-malleista merkittävimpiä ovat Ostolasku ja Toimittaja. Niiden tietosisällöt ovat samankaltaisia Lasku- ja Asiakas-mallien kanssa. Lisäksi osa Ostolasku-malliin viittaavista malleista on identtisiä myyntilaskupuolen mallien kanssa.

Riskiä olemassa olevien mallien tai niihin liittyvien ominaisuuksien hajoamiselle ei ole, koska niihin ei tule muutoksia. Toisaalta osassa uusia malleja toteutuksen lähdekoodi olisi suora kopio, mikä rikkoo ohjelmoinnin DRY-periaatetta.



Kuva 6.1. *Yksinkertaistettu kuvaus osto- ja myyntilaskuun liittyvistä Django-malleista ostolaskumoduuli toteutettaessa kopioimalla ja muokkaamalla*

6.4.2 Refaktoroidaan olemassa olevia malleja

Django-mallien periytyksen toteuttaminen järkevästi on mahdollista kolmella eri tavalla. Periyttämiseen voidaan käyttää abstraktia kantaluokkaa, Djangon proxy-mallia tai konkreettista kantaluokkaa. Eri periyttämistyyppien eroja on käsitelty tarkemmin luvussa 5.1.

Kaikissa tapauksissa Asiakas- ja Toimittaja-mallin kantaluokka nimetään LaskunVastapuoli-nimellä. Vastaavasti Ostolasku- ja Lasku-mallin kantaluokka nimetään BaseLasku-nimellä. Selkeämpää olisi käyttää kantaluokasta nimeä Lasku. Tällöin alaluokat olisivat Ostolasku ja Myyntilasku. Näin nimet saataisiin vastaamaan reaalimaailmassa käytettyjä termejä. Näin ei kuitenkaan tehty, koska Lasku-mallin nimen vaihtaminen Myyntilaskuksi kaikkialle olemassa olevaan koodiin olisi vaatinut huomattavan määrän lisätyötä ja riskejä.

Abstraktit kantaluokat

Kuvassa 6.2 esitetään yksinkertaistettu kuvaus Django-malleista, kun periyttäminen tehdään abstraktien kantaluokkien avulla. Tällöin BaseLasku- ja LaskunVastapuoli-mallit ovat abstrakteja eli niillä ei ole tietokannassa tauluja.

Koska Lasku- ja Ostolasku-malleille syntyy tietokantaan omat taulut, riski mallien sekoittumiselle koodissa vähenee huomattavasti. Lisäksi suurimmalta osin vältetään Djangon ongelmat mallien polymorfismin kanssa, koska viittaavat mallit palauttavat suoraan oikean alaluokan. Heikkoutena on, että sekä osto- että myyntilaskuihin viittaaviin malleihin on lisättävä erilliset vierasavaimet molemmille tauluille. Uusien vierasavainten lisäämisen myötä tietokannan eheyden varmistaminen vaatii rajoitteiden (engl. constraint) lisäämistä tietokantaan. Lisäksi sekä myyntilaskuihin että ostolaskuihin kohdistuvat tietokantaoperaatiot hankaloituvat, koska tietokannassa ei ole taulujen välillä mitään yhdistävää tekijää.

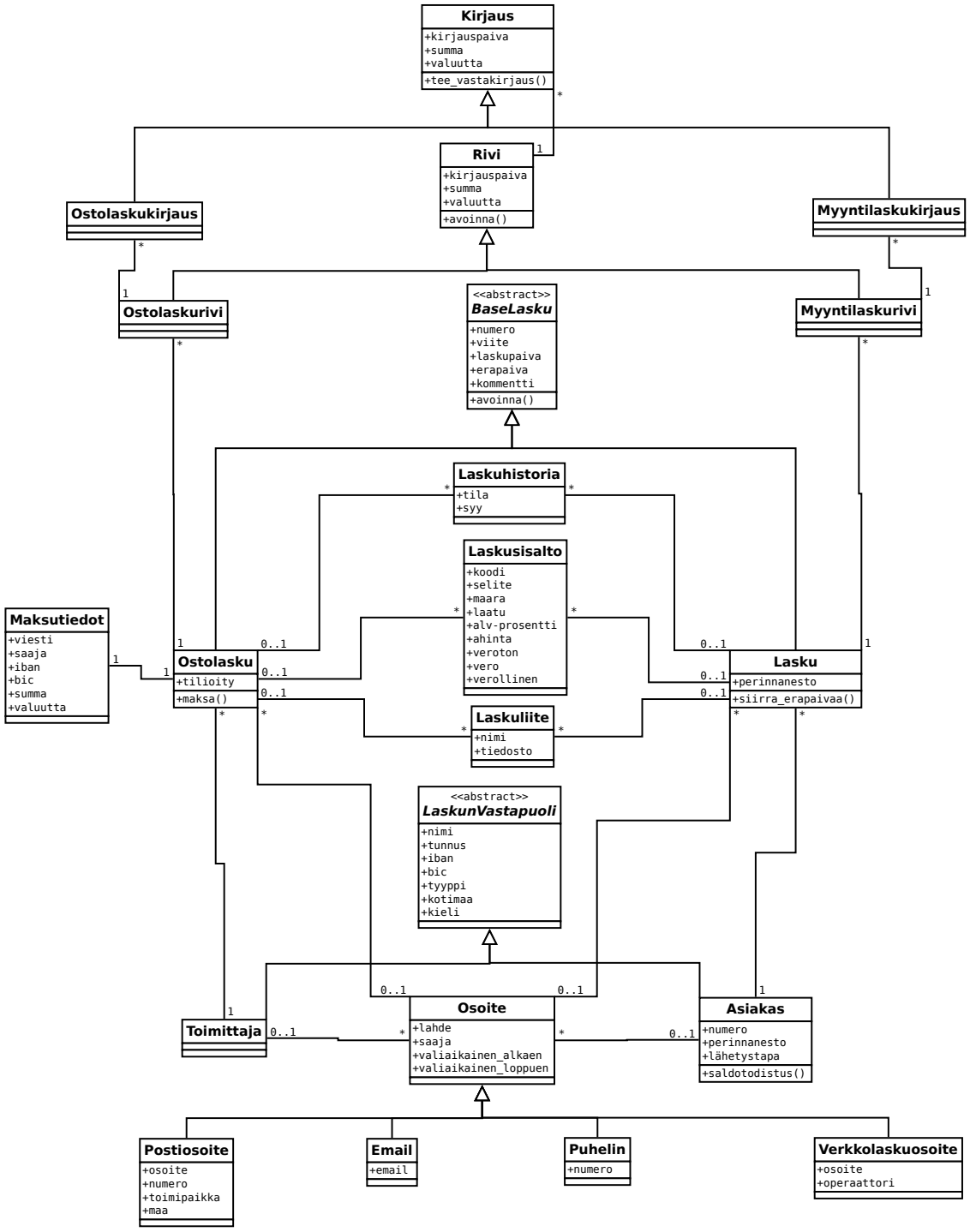
Proxy-mallit

Kuvassa 6.3 esitetään yksinkertaistettu kuvaus Django-malleista, kun periyttäminen tehdään proxy-mallien avulla. Ostolasku, Lasku, Toimittaja ja Asiakas ovat proxy-malleja eli niillä ei ole tietokannassa omia tauluja. BaseLasku- ja LaskunVastapuoli-malliin on lisätty tyyppi-kenttä erottamaan alaluokat tietokannassa toisistaan.

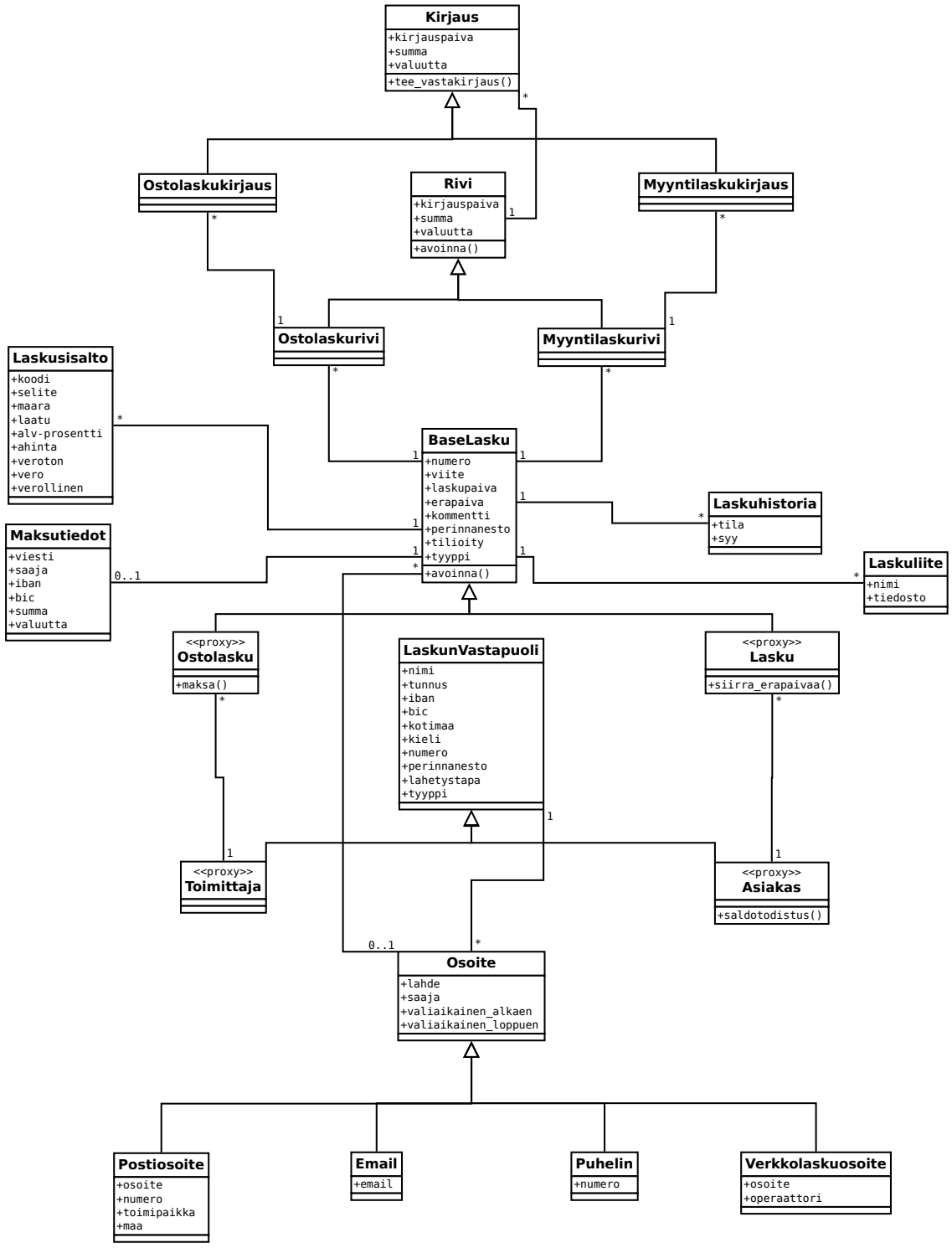
Proxy-mallien käyttö mahdollistaa monien myyntilaskuihin viittaavien mallien (kuvassa Laskusisalto, Laskuhistoria ja Laskuliite) käytön ostolaskuilla suoraan ilman muutoksia. Proxy-mallien käytön suurin haaste on Djangon ongelmat polymorfismin kanssa. Olemassa oleva koodi hajoaa monilta osin, koska Lasku-mallin sijasta käsittelyyn tuleekin BaseLasku-malli. Lisäksi kaikkien kenttien ollessa samassa taulussa pelkkien ostolaskuihin tai myyntilaskuihin liittyvien kenttien erottaminen yhteisistä kentistä on hankalaa. Saman taulun käyttäminen lisää myös riskiä sille, että mallit menevät keskenään sekaisin.

Monitauluperiytyminen

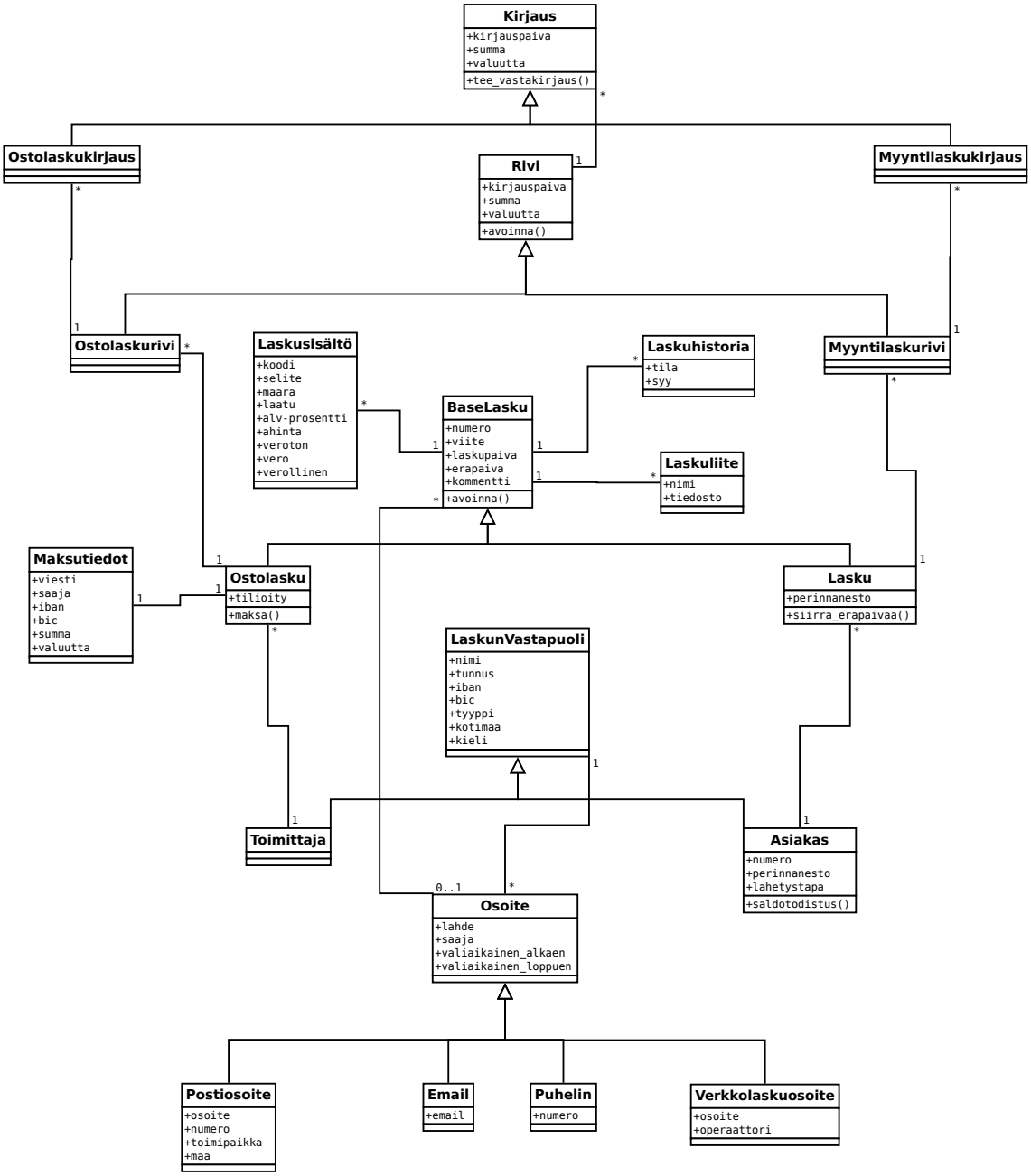
Kuvassa 6.4 esitetään yksinkertaistettu kuvaus Django-malleista, kun periyttäminen tehdään monitauluperiytymisen avulla. BaseLasku ja LaskunVastapuoli ovat konkreettisia



Kuva 6.2. *Yksinkertaistettu kuvaus osto- ja myyntilaskuun liittyvistä Django-malleista periyttäminen tehdessä abstrakteilla kantaluokilla*



Kuva 6.3. *Yksinkertaistettu kuvaus osto- ja myyntilaskuun liittyvistä Django-malleista periryttäminen tehdessä proxy-malleilla*



Kuva 6.4. *Yksinkertaistettu kuvaus osto- ja myyntilaskuun liittyvistä Django-malleista periyttäminen tehdessä monitauluperiyttämisellä*

kantamalleja eli niillä on tietokannassa omat taulut ja lapsimallit viittaavat kyseisiin tauluihin vierasavaimella.

Monitauluperiytymisen käyttö yhdistää abstraktien kantaluokkien ja proxy-mallien hyvät puolet. Riski Lasku- ja Ostolasku-mallien sekoittumiselle on vähäinen, koska tietokannassa on malleille omat taulut. Vierasavainviittaukset voidaan määrittää joko kantamalliin tai lapsimalliin. Tämä mahdollistaa viittaavien mallien käytön joustavasti. Merkittävä ongelma monitauluperiytymisen käytössä on suorituskyky. Myyntilaskun tai ostolaskun hakeminen vaatii aina ylimääräisen liitoskyselyn, kun kantamallin tiedot pitää yhdistää lapsimalliin. Vastaavasti käy toimittajan ja asiakkaan haun yhteydessä.

Varsinaiseen refaktorointiin periytysmalliksi valittiin abstraktit kantaluokat. Vierasavaimien lisääminen todettiin pienimmäksi lisätyöksi, sillä niiden lisääminen ei vaadi muutoksia olemassa oleviin toiminnallisuuksiin. Sekä myyntilaskuihin että ostolaskuihin kohdistuvien tietokantaoperaatioiden tekemisen hankaluus ei myöskään ole merkittävä ongelma. Tällä hetkellä kyseisenlaisille operaatioille ei ole tarvetta. Tulevaisuudessa tarpeita saattaa tulla, kun lasketaan kehittyneempiä tilastoja, esimerkiksi kassavirtaennusteita. Erilaisia molempiin tauluihin kohdistuvia operaatioita ei pitäisi kuitenkaan tulla kovinkaan montaa.

Proxy-malleja ei saatu toimimaan olemassa olevien toiminnallisuuksien kanssa luotettavasti polymorfismiongelmien vuoksi. Ratkaisuksi kokeiltiin myös avoimen lähdekoodin django-polymorphic python-pakettia [34]. Sen käyttöönoton todettiin kuitenkin vaativan enemmän töitä kuin abstraktien kantaluokkien käyttö. Lisäksi django-polymorphic paketin käyttö aiheuttaa ylimääräisen liitoskyselyn, mikä alentaa suorituskykyä. Monitauluperiytymisen käyttöä ei myöskään harkittu mahdollisten suorituskykyongelmien vuoksi. Myyntilaskuihin, ostolaskuihin, asiakkaisiin ja toimittajiin liittyy paljon käsittelyä ja toiminnallisuuksia, joiden pitää toimia mahdollisimman nopeasti. Tämän vuoksi suorituskykyä alentavaa ratkaisua ei voida ottaa käyttöön.

6.5 Komponenttipohjaisen kehityksen hyödyntäminen

Django-mallien periytyksen toteutustavan päättämisen jälkeen yhteistä laskukomponenttia lähdettiin kehittämään komponenttipohjaista kehitystä hyväksi käyttäen. Tarkoituksena ei ollut tehdä komponentista täydellistä mustaa laatikkoa käytännön rajoitteiden vuoksi. Laskukomponentti onkin komponentti enemmän teollisuuden näkökulmasta kuin akateemisesta näkökulmasta.

Laskukomponentin rajapintana toimii Django-mallien metodit. Myyntilaskutoteutuksesta erotettiin yleiseen laskukomponenttiin myös ostolaskumoduulin tarvitsemat yhteiset metodit. Yhteisten metodien määrittäminen perustui asiantuntija-arvioon siitä, että mitkä operaatiot tehdään sekä osto- että myyntilaskuille. Yhteisiä operaatioita ovat muun muassa laskun tilan vaihdot ja laskun avoimeen summaan liittyvät laskennat. Yhteisiin metodeihin lisättiin tuki molemmille alaluokille.

Yhteisiä metodeita parannettiin sopimussuunnittelun periaatteita hyödyntämällä. Python ei suoraan tue sopimussuunnittelua eli käytännössä esi- ja jälkiehtoja tai invariantteja. Avoimen lähdekoodin kirjastoja tuen lisäämiselle on olemassa. Tällaista kirjastoa ei kuitenkaan haluta ottaa käyttöön, jotta toteutus ei eroaisi merkittävästi muusta järjestelmästä. Sopimussuunnittelun ideaa sovellettiin kuitenkin automaatiotestien, assert-komentojen ja koodikommenttien avulla, vaikka se ei olekaan mitä sopimussuunnittelulla alun perin ajateltiin. Huomioitavaa on myös, että invariantteja ja esi- ja jälkiehtoja ei määritelty täydellisesti. Osa toiminnallisuuksista ei ole kriittisiä tai usein käytettyjä. Erityisesti näiden kohdalla määrittelyä karsittiin tai se jätettiin kokonaan tekemättä. Täydellinen määrittely olisi vienyt kohtuuttoman paljon aikaa verrattuna siitä saatavaan hyötyyn.

Metodi sisältää samalla toteutuksen, joten toteutusta ei voi suoraan vaihtaa ilman raja-

pinnan muuttamista. Tämän ei pitäisi olla ongelma, koska on vaikea kuvitella tilannetta, jossa toteutus haluttaisiin vaihtaa toiseen. Nykyisen toteutuksen muokkaaminen on paljon todennäköisempää.

Django-mallien kenttien käsittely on mahdollista suoraan ilman metodien käyttöä, mikä mahdollistaa sisäisen toteutuksen käsittelyn metodien ohi. Jokaisen kentän käsittelyyn voisi tehdä oman metodin. Missään muualla järjestelmässä ei ole kuitenkaan tehty näin, joten tässä ei haluta poiketa yleisestä mallista. Kentät voidaan myös ajatella osana rajapintaa.

6.6 Ostolaskumoduuliin tehty toteutus

Lähes kaikki ostolaskumoduulin logiikka sijaitsee Django-mallien metodeissa. Yhteisen laskukomponentin metodien lisäksi ostolaskun omiksi metodeiksi toteutettiin muun muassa ostolaskujen luonti, maksaminen ja tiliöintiominaisuudet.

Merkittävää metodien ulkopuolista logiikkaa ovat raportointi ja REST-rajapinnan toteutus. Raportoinnissa hyödynnetään metodeita, mutta ne sisältävät myös omaa logiikka liittyen tietojen laskentaan ja esittämiseen. REST-rajapinta sisältää käyttöoikeuksien tarkistuksen. Muutoin REST-rajapinnan toteutus on hyvin yksinkertainen. Tietoa haettaessa REST-rajapinta palauttaa mallin sisältämät kentät ja mahdollisesti jotain metodin laskemaa lisätietoa. Muutoksia tehdessä REST-rajapinnasta kutsutaan jotain metodia ja mahdollisesti välitetään sille käyttöliittymästä tulleet metodin vaatimat kentät.

7 ARVIOINTI

Pragmaattisen uudelleenkäytön säästettyjä kustannuksia mitattiin Irshad et al. [46] kehittämällä yksinkertaisella kaavalla:

$$C = (O - I) \times H \tag{7.1}$$

Jossa C on säästetyt kustannukset, O on sisäisesti kehitettyyn ohjelmistoartefaktiin käytetyt työtunnit, I on artefaktin uudelleenkäyttöön käytetyt työtunnit kuten artefaktin tunnistus, ymmärtäminen ja integrointi ja H on yhden työtunnin kustannus. Laskuissa työtunnin kustannus H voidaan jättää huomioimatta, koska kustannus on vakio ja tämän työn kannalta kiinnostavaa on pelkästään käytetty aika. Kaava ei ota huomioon mahdollisia artefaktin ylläpidossa vältettyjä kustannuksia.

Ostolaskumoduulin kehityksessä uudelleenkäytettyjen myyntilaskuominaisuuksien käytetyiksi työtunneiksi arvioitiin 500 tuntia. Kyseessä on myyntilaskuominaisuudet toteuttaneen henkilön asiantuntija-arvio, sillä historiallista dataa ei ole saatavilla. Tutkimusten perusteella asiantuntija-arvion käyttö on ohjelmistokehityksessä luotettava menetelmä, jos dataa ei ole saatavilla [46][47][48][49]. Myyntilaskuominaisuuksien uudelleenkäyttöön käytetyt työtunnit olivat 100 tuntia. Säästetyiksi työtunneiksi saadaan täten 400 tuntia. Jos tätä verrataan koko projektiin käytettyyn aikaan, joka oli 450 tuntia, voidaan todeta, että pragmaattinen uudelleenkäyttö oli ajankäytön suhteen järkevää. On huomioitava, että tässä verrataan ostolaskumoduulin toteutusta vaihtoehtoon, jossa nykyistä lähdekoodia ei olisi hyödynnetty millään tavalla.

Ostolaskumoduulin toteutukseen käytetyksi ajaksi arvioitiin 370 tuntia, jos ostolaskumoduuli olisi toteutettu ilman nykyisen lähdekoodin refaktorointia. Aikaa olisi siis säästetty 80 tuntia verrattuna tehtyyn toteutukseen. Pitkällä aikavälillä 80 tunnin erotus pitäisi kuroutua umpeen, kun otetaan huomioon ylläpidossa ja jatkokehityksessä säästettävä aika. Suurin hyöty olisi ollut nopeammin valmis tuote. Käytännössä 80 tunnin ajansäästö olisi tarkoittanut, että ostolaskumoduuli olisi ollut valmiina noin kuukautta aikaisemmin. Tässä tapauksessa kuukauden aikaisemmalla valmistumisella ei olisi ollut suurta merkitystä, koska aikataulu valmistumisen suhteen ei ollut tiukka. Anittaan on kuitenkin tehty vastaavankokoisia projekteja, joissa kuukauden viive olisi aiheuttanut merkittäviä taloudellisia tappioita. Tällöin kannattaa valita nopeampi toteutus, vaikka laatu ei olisi yhtä hyvä.

Eniten työtä refaktoroinnissa aiheutti oletetusti jo käytössä olevan toiminnallisuuden pitäminen ehjänä. Tässä onnistuttiin ennakoitua paremmin, sillä ostolaskumoduulin käyttöön-

otto ei ollut aiheuttanut yhtään virhettä jo käytössä olleisiin toiminnallisuuksiin diplomityön kirjoitushetkellä kuukausi käyttöönoton jälkeen. Virheettömyys yllätti sekä diplomityön kirjoittajan että toisen asiantuntijan. Molemmilla on kokemusta useista samankaltaisista päivityksistä Anittaan ja niistä ei ole ennen tätä selvitty ilman virheitä. Syy virheettömyyteen on luultavasti aiempaa parempi suunnittelu. Soveltamalla hyväksy todettuja periaatteita, kuten komponenttipohjaista kehitystä, saadaan aikaiseksi laadukkaampaa lähdekoodia. Osaselittäjänä voi olla myös kokemuksen kautta kehittynyt parempi ohjelmointitaito.

Django-mallien periytyksen eri vaihtoehtojen tutkiminen, vertailu ja testaaminen vei selkeästi arvioitua enemmän aikaa. Toisaalta tähän käytettyä aikaa ei voida suoraan pitää pelkästään ostolaskumoduuliin käytettynä aikana, sillä kerättyä tietämystä voidaan hyödyntää myös tulevissa projekteissa.

Myyntilaskuominaisuuksien lähdekoodin laatua saatiin kasvatettua. Erityisesti koodikommenttien ja testien lisääminen oli hyödyllistä, sillä niitä ei osassa koodia ollut juuri ollenkaan. Koodiin tehtiin myös paljon muita pieniä parannuksia. Tämän tyyppisten muutosten teko on yleensä mielekkäintä jonkun ison muutoksen yhteydessä. Jos isompaa refaktorointia ei olisi tehty, niin myös pienemmät muutokset olisivat jääneet tekemättä ja odottamaan jotain muuta isoa muutosta. Refaktoroinnin yhteydessä myös löydettiin ja korjattiin yksi bugi.

Osto- ja myyntilaskujen ylläpidossa ja jatkokehityksessä on tulevaisuudessa oltava tarkkana sen suhteen, että minne muutokset tehdään. Muutokset on mahdollista tehdä kolmeen eri paikkaan, ostolaskutoteutukseen, myyntilaskutoteutukseen tai yhteiseen komponenttiin. Täten kehittäjän on tiedettävä, liittyykö muutos pelkästään ostolaskuihin tai myyntilaskuihin vai onko se yhteinen molemmille. Erityisen ongelmallista on, jos vain toiseen liittyvä uusi ominaisuus lisätään yhteiseen komponenttiin. Tällöin ulospäin kaikki toimii oikein, mutta yhteiseen komponenttiin jää hämmentävästi ominaisuus, jota ei oikeasti molemmat käytä. Ylläpitäjiltä vaaditaan siis jatkossa enemmän osaamista.

Yleisesti ylläpidon suhteen ratkaisun onnistumista on liian aikaista arvioida. Aikaisempien kokemusten perusteella ratkaisun hyvyys mitataan siinä vaiheessa, kun tulee muutostarve, jota ei tässä vaiheessa ole ollut mahdollista ennakoida.

8 YHTEENVETO

Tämän työn tavoitteena oli tutkia miten lähdekoodia kannattaa uudelleenkäyttää valmiista järjestelmästä uuden moduulin teossa ajankäytön, ylläpidettävyyden ja virheherkkyyden suhteen. Työssä toteutettiin laskutus- ja perintäjärjestelmään ostolaskumoduuli uudelleenkäyttämällä pragmaattisesti jo tuotantokäytössä olleen myyntilaskutoteutuksen lähdekoodia.

Uudelleenkäytön suunnittelu aloitettiin valitsemalla pragmaattisen uudelleenkäytön tapa kahdesta mahdollisesta vaihtoehdosta. Olemassa olevan lähdekoodin hyödyntämistä kopioimalla ja muokkaamalla verrattiin lähdekoodiin refaktorointiin. Ratkaisuksi valittiin lähdekoodin refaktorointi. Merkittävänä syynä valinnalle oli, että osto- ja myyntilaskuprosessien havaittiin olevan suurilta osin samanlaisia. Tällöin niissä kannattaa käyttää yhteistä toteutusta.

Myyntilaskutoteutuksen lähdekoodia refaktoroitiin erottamalla siitä yleiskäyttöinen laskukomponentti, jota hyödynnettiin ostolaskumoduulin toteutuksessa. Tärkeä osa refaktoroinnin toteutuksessa oli Django-mallien periyttämistavan valinta. Abstrakteja kantaluokkia, proxy-malleja ja monitauluperiyttämistä vertailtiin keskenään. Varsinaiseksi periyttämistavaksi valittiin abstraktit kantaluokat. Muussa refaktoroinnissa hyödynnettiin mahdollisuuksien mukaan komponenttipohjaisen kehityksen periaatteita.

Ostolaskumoduulin toteuttaminen olemassa olevaa lähdekoodia refaktoroimalla laskettiin vieneen 80 tuntia enemmän aikaa kuin se olisi vienyt lähdekoodia kopioimalla ja muokkaamalla. Koko projektiin käytetty aika oli 450 tuntia. Aikaa kuitenkin säästetään tulevaisuuden ylläpidossa ja jatkokehityksessä. Ylläpidon arvioitiin olevan jatkossa helpompaa, vaikka ylläpitäjältä vaaditaankin enemmän ymmärrystä järjestelmästä. Tuotantokäytössä olleen lähdekoodin refaktorointi ei aiheuttanut virheitä jo käytössä olleisiin myyntilaskuominaisuuksiin, vaikka se arvioitiin suurimmaksi riskiksi ennen muutosten tekoa. Refaktoroinnin myötä myyntilaskuominaisuuksien lähdekoodin laatua saatiin kasvatettua, mikä nähtiin erityisen hyödyllisenä.

Ostolaskumoduulin kehitys jatkuu tulevaisuudessa. Työssä tehty toteutus oli tarkoituksella mahdollisimman yksinkertainen ja siihen onkin suunnitteilla uusia ominaisuuksia. Jatkokehityksen yhteydessä saadaan lisää tietoa siitä, oliko valittu toteutustapa oikea.

LÄHTEET

[1] McIlroym, M., Buxton, J., Naur, P. ja Randell, B. Mass-produced software components. *International Conference on Software Engineering* (1968), 88–98.

[2] Barros-Justo, J. L., Benitti, F. B. ja Matalonga, S. Trends in software reuse research: A tertiary study. *Computer Standards & Interfaces* 66 (2019), 103352. ISSN: 0920-5489.

[3] de Almeida, E. S., Alvaro, A., Lucredio, D., Garcia, V. C. ja de Lemos Meira, S. R. A survey on software reuse processes. *IRI -2005 IEEE International Conference on Information Reuse and Integration, Conf, 2005.* Elokuu 2005, 66–71. DOI: 10.1109/IRI-05.2005.1506451.

[4] Barros-Justo, J. L., Pinciroli, F., Matalonga, S. ja Martínez-Araujo, N. What software reuse benefits have been transferred to the industry? A systematic mapping study. *Information and Software Technology* 103 (2018), 1–21.

[5] IEEE Standard for Information Technology–System and Software Life Cycle Processes–Reuse Processes. *IEEE Std 1517-2010 (Revision of IEEE Std 1517-1999)* (elokuu 2010), 1–51. DOI: 10.1109/IEEESTD.2010.5551093.

[6] Szyperski, C., Gruntz, D. ja Murer, S. *Component Software: Beyond Object-oriented Programming*. ACM Press Series. ACM Press, 2002. ISBN: 9780201745726.

[7] Holmes, R. ja Walker, R. J. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21.4 (2013), 1–44.

[8] Prieto-Díaz, R. Status report: Software reusability. *IEEE software* 10.3 (1993), 61–66.

[9] Ravichandran, T. ja Rothenberger, M. Software reuse strategies and component markets. *Commun. ACM* 46 (elokuu 2003), 109–114. DOI: 10.1145/859670.859678.

[10] Casanave, C. Business-Object Architectures and Standards. *Business Object Design and Implementation*. Toim. J. Sutherland, C. Casanave, J. Miller, P. Patel ja G. Hollowell. London: Springer London, 1997, 7–28. ISBN: 978-1-4471-0947-1.

[11] Crnkovic, I. ja Larsson, M. P. H. *Building reliable component-based software systems*. Artech House, 2002.

[12] Caldiera, G. ja Basili, V. R. Identifying and qualifying reusable software components. *Computer* 24.2 (1991), 61–70.

[13] Pfister, C. ja Szyperski, C. Why objects are not enough. *CUC96: Component Based Software Engineering* (1998), 141.

[14] Bosch, J. ja Bengtsson, P. Component Evolution in Product-Line Architectures. *Proceedings of International Workshop on Component Based Software Engineering*. 1999.

[15] Crnkovic, I. ja Larsson, M. A case study: demands on component-based development. *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. 2000, 23–31.

[16] Meyer, B. Applying 'design by contract'. *Computer* 25.10 (1992), 40–51.

[17] Gamma, E., Helm, R., Johnson, R. ja Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[18] Johnson, R. E. Frameworks = (Components + Patterns). *Commun. ACM* 40.10 (lokakuu 1997), 39–42. ISSN: 0001-0782. DOI: 10.1145/262793.262799. URL: https://doi.org/10.1145/262793.262799.

[19] Jacobson, I., Griss, M. ja Jonsson, P. *Software Reuse: Architecture Process and Organization for Business Success*. ACM Press Books. ACM Press, 1997. ISBN: 9780201924763.

[20] *Laravel*. URL: https://laravel.com/ (viitattu 20. 04. 2021).

[21] *Django Web Framework*. Django Software Foundation. URL: https://www.djangoproject.com (viitattu 23. 02. 2021).

[22] Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R. ja Wallnau, K. *Volume II: Technical concepts of component-based software engineering*. Tekninen raportti. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon Software Engineering, 2000.

[23] Weck, W. Independently extensible component frameworks. *Special Issues in Object-Oriented Programming, M. Mühlhäuser (ed.), dpunkt Verlag* (1997), 177–183.

[24] Tripathy, P. ja Naik, K. *Software evolution and maintenance: a practitioner's approach*. John Wiley & Sons, 2014. ISBN: 1-118-96463-2.

[25] Brown, A. W. *Large-scale, component-based development*. Vol. 1. Prentice Hall PTR Englewood Cliffs, 2000.

[26] Maiden, N. A. ja Ncube, C. Acquiring COTS software selection requirements. *IEEE Software* 15.2 (1998), 46–56.

[27] Larsson, M. *Applying Configuration Management Techniques to Component-based Systems*. Tekninen raportti ISSN 1404-3041 ISRN MDH-MRTC-24/2000-1-SE. Joulukuu 2000. URL: http://www.es.mdh.se/publications/425-.

[28] Pohl, K., Böckle, G., Linden, F. van der ja Bc6ckle, G. *Software Product Line Engineering: Foundations, Principles, and Techniques*. eng. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2005. ISBN: 3642063640.

[29] Kaarlejärvi, S. ja Salminen, T. *Älykäs taloushallinto : automaation aika*. fin. Helsinki: Alma Talent, 2018. ISBN: 978-952-14-3431-0.

[30] Tomperi, S. *Käytännön kirjanpito*. fin. 28., uudistettu painos. Helsinki: Edita, 2020. ISBN: 978-951-37-7827-9.

[31] *Laki saatavien perinnästä, 22.4.1999/513*. 1999. URL: https://www.finlex.fi/fi/laki/ajantasa/1999/19990513 (viitattu 17. 01. 2020).

[32] Schutt, K. ja Balci, O. Cloud software development platforms: A comparative overview. *2016 IEEE 14th International Conference on Software Engineering Research,*

Management and Applications (SERA). Kesäkuu 2016, 3–13. DOI: 10.1109/SERA.2016.7516122.

[33] Ravindran, A. *Django Design Patterns and Best Practices: Industry-standard web development techniques and solutions using Python, 2nd Edition*. Packt Publishing, 2018. ISBN: 9781788834971. URL: https://books.google.fi/books?id=FnxeDwAAQBAJ.

[34] *Django Polymorphic package*. URL: https://github.com/django-polymorphic/django-polymorphic (viitattu 24. 02. 2021).

[35] Fielding, R. T. ja Taylor, R. N. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)* 2.2 (2002), 115–150.

[36] Kendall, S. C., Waldo, J., Wollrath, A. ja Wyant, G. *A note on distributed computing*. 1994.

[37] Clark, D. D. ja Tennenhouse, D. L. Architectural considerations for a new generation of protocols. *ACM SIGCOMM Computer Communication Review* 20.4 (1990), 200–208.

[38] Doglio, F. *REST API Development with Node.js : Manage and Understand the Full Capabilities of Successful REST Development*. eng. 2. painos. For professionals by professionals. Apress, 2018. ISBN: 1484237153.

[39] *Korkolaki, 20.8.1982/633*. 1982. URL: https://www.finlex.fi/fi/laki/ajantasa/1982/19820633 (viitattu 17. 01. 2020).

[40] *Laskutusvaatimukset arvonlisäverotuksessa, diaarinumero VH/1780/00.01.00/2019*. Verohallinto. 27. syyskuuta 2019. URL: https://www.vero.fi/syventavat-vero-ohjeet/ohje-hakusivu/48090/laskutusvaatimukset-arvonlis%C3%A4verotuksessa/ (viitattu 24. 01. 2020).

[41] Directive 2014/55/EU of the European Parliament and of the Council of 16 April 2014 on electronic invoicing in public procurement. *OJ* L 133 (6. toukokuuta 2014), 1–11. URL: https://eur-lex.europa.eu/eli/dir/2014/55/oj.

[42] *Finvoice-soveltamisohje*. Versio 3.0. Finanssiala ry. 26. kesäkuuta 2018. URL: https://www.finanssiala.fi/finvoice/dokumentit/Finvoice_3_0_soveltamisohje.pdf (viitattu 24. 01. 2020).

[43] *Finvoice-välityspalvelun kuvaus*. Finanssiala ry. 2. tammikuuta 2018. URL: https://www.finanssiala.fi/finvoice/dokumentit/Finvoice-valityspalvelun_kuvaus.pdf (viitattu 24. 01. 2020).

[44] *Django REST framework*. URL: https://www.django-rest-framework.org/ (viitattu 12. 04. 2021).

[45] *VueJS*. URL: https://v3.vuejs.org/ (viitattu 13. 04. 2021).

[46] Irshad, M., Torkar, R., Petersen, K. ja Afzal, W. Capturing cost avoidance through reuse: systematic literature review and industrial evaluation. *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. 2016, 1–12.

[47] Jørgensen, M. A review of studies on expert estimation of software development effort. *Journal of Systems and Software* 70.1-2 (2004), 37–60.

[48] Hughes, R. T. Expert judgement as an estimating method. *Information and software technology* 38.2 (1996), 67–75.

[49] Rush, C. ja Roy, R. Expert judgement in cost estimating: Modelling the reasoning process. *Concurrent Engineering* 9.4 (2001), 271–284.