Tampereen ammattikorkeakoulu

Luottoluokittajaraportin asiakaskohtainen räätälöinti

Hanna Tuominen

OPINNÄYTETYÖ
Marraskuu 2021

Tietojenkäsittelyn tutkinto-ohjelma
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn tutkinto-ohjelma
Ohjelmistotuotanto

TUOMINEN, HANNA:
Luottoluokittajaraportin asiakaskohtainen räätälöinti

Opinnäytetyö 34 sivua
Marraskuu 2021

Tässä opinnäytetyössä suunniteltiin ja toteutettiin mallisovellus, jolla pyrittiin löytämään ratkaisu luottoluokittajaraporttien asiakaskohtaiseen räätälöintiin. Työn toimeksiantajana toimi Profit Software Oy, joka on finanssialalle järjestelmäratkaisuja ja konsultointipalveluita toimittava yritys. Tavoitteena oli luoda mallisovellus, joka tarjoaisi mahdollisen ratkaisutavan asiakaskohtaisen räätälöinnin ongelmiin, jota toimeksiantajayritys pystyisi hyödyntämään taustatutkimuksena tuotteessaan.

Opinnäytetyön tavoite oli tutkia erilaisia tapoja ratkaista luottoluokittajaraporttien asiakaskohtaisen räätälöinnin keskeiset ongelmat. Työn tarkoituksena oli tutkimuksen perusteella valita parhaimmat ratkaisutavat, joita hyödyntäen suunniteltiin ja toteutettiin toimiva mallisovellus käyttäen C#-ohjelmistokieltä sekä ASP.NET Core Frameworkia.

Opinnäytetyön tuloksena rakennettiin toimiva mallisovellus, jossa hyödynnetiin Design Patterneja, eli käyttäytymismalleja, sekä tietokantatauluja ongelmien ratkaisemiseen. Tutkimuksen tuloksena havaittiin, että strategiamallit ovat hyvä ratkaisutapa räätälöinnin skaalattavuutta, ylläpidettävyyttä sekä hallittavuutta ratkaistaessa. Työssä todettiin myös, että tietokantataulut ovat tarkoin suunniteltuina hyvä tapa ratkaista asiakaskohtaisten asetuksien säilöminen API-rajapintaa hyödyntäen. Työssä huomattiin, että strategiamallit ja tietokantataulut toimivat hyvin yhteen, ja niitä pystyy hyödyntämään monipuolisesti.

Tutkimuksen tuloksena havaittiin, että jatkokehityksen kannalta strategiamallien sekä tietokantataulujen käyttö tulee suunnitella sekä jaotella selkeästi jo työn alkuvaiheissa, jotta niitä on helppo ylläpitää sekä hallita. Työssä todettiin myös, että toimeksiantajayrityksen tuotteen jatkokehityksen kannalta tietokantataulujen sekä käyttäytymismallien hyödyntäminen voisi tulevaisuudessa säästää henkilötyötunteja ja täten yritykseltä rahaa tuotteen muuhun kehittämiseen uusien asiakassuhteiden alkuvaiheissa.

Asiasanat: asiakaskohtainen räätälöinti, ASP.NET Core, strategiamalli, tietokannat

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Production

TUOMINEN, HANNA:
Customer Specific Customization of Credit Rating Agency Report

Bachelor's thesis 34 pages
November 2021

The goal of this thesis was to provide a model software application that solved the main problems in customer specific customization of credit rating agency reports. The project was commissioned by Profit Software Oy, a company that provides system solutions to the financial industry. The application was developed using ASP.NET core Framework, and its purpose was to be used as a research base for the commissioner’s product.

The objective of this thesis was to research different methods to help with the different core problems of credit agency report customer customization. Based on the research, the goal was to select the best methods and design and provide a model application that would convey the best solutions to the different problems.

The result was a functioning model application, that the company could use as a research base for their product. The application utilizes strategy patterns and database tables as solutions to its core problems such as scalability, manageability, and reusability.

Key words: customer specific customization, ASP.NET Core, strategy pattern, databases

SISÄLLYS

JOHDANTO 6
1 OPINNÄYTETYÖN LÄHTÖKOHDAT 7
1.1 Asiakaskohtaisen räätälöinnin tarve 7
1.2 Luottoluokittajaraportti 7
1.3 Käytetyt teknologiat 8
1.4 Opinnäytetyön tavoite ja tarkoitus 9
2 MICROSOFTIN ASP.NET-YMPÄRISTÖ JA DESIGN PATTERNIT ... 11
2.1 ASP.NET 11
2.2 Design Patternit 12
2.2.1 Strategy Pattern 13
2.2.2 Dependency Injection 14
3 TIETOKANTATAULUJEN TARVE 16
3.1 Miksi tietokantataulut? 16
3.2 Skaalattavuus ja uudelleenkäyttö 17
4 RAPORTOINTISOVELLUSMALLIN RAKENTAMINEN 18
4.1 Raporttisovelluskokonaisuus lyhyesti 18
4.1.1 ASP.NET Core Web App (MVC) 19
4.2 Asiakaskonfiguraatioiden saanti tietokannan avulla 21
4.3 Vakuudet 21
4.3.1 Tehdyn ratkaisun uudelleenkäytettävyys ja skaalattavuus . 26
4.4 Eri Excel-pohjien mahdollistaminen raportille 28
4.4.1 Tehdyn ratkaisun uudelleenkäytettävyys ja skaalattavuus . 31
5 POHDINTA 32
LÄHTEET 34

ERITYISSANASTO

Design Pattern	Yleinen, uudelleenkäytettävä suunnittelumalli tiettyihin ohjelmistosuunnittelun ongelmiin
Strategy Pattern	Strategiamalli koostuu kolmesta pääkomponentista: strategiasta, konkreettisesta strategiasta sekä kontekstista, joita käytetään algoritmien erottelussa
ASP.NET Core	Microsoftin kehittämä web-kehityskirjasto
MVC	Arkkitehtuurimalli, jota käytetään käyttöliittymän, datan sekä sovelluslogiikan erottamiseen sovelluksessa
C#	Microsoftin kehittämä ohjelmointikieli, joka on kehitetty .NET-kirjastoa varten
Luottoluokittajaraportti	Luotu Excel-raportti, jonka luottoluokittaja vaatii pankilta
Backend	Sovelluskerros, jossa verkkosovelluksissa on muun muassa palvelinlogiikka, ja jossa komentojen käsittely tapahtuu
Frontend	Verkkosovelluksissa frontend tarkoittaa selaimessa käyttäjälle esitettävää käyttöliittymää ja ulkoasua
API	Application Programming Interface on ohjelmointirajapinta, jota hyödynnetään kun halutaan, että kaksi sovellusta keskustelevat keskenään

JOHDANTO

Tämä opinnäytetyö asettuu finanssialan sekä web-ohjelmoinnin aihealueille. Tässä opinnäytetyössä on tarkoitus tutkia, kuinka toimeksiantajayrityksen tuotteeseen olisi mahdollista rakentaa asiakaskohtainen räätälöinti luottoluokittajaraporteille. Tarkempana tarkoituksena on löytää ratkaisu kolmeen ongelmaan. Ensimmäisenä ongelmana on se, miten asiakaskohtainen räätälöinti toteutetaan helposti skaalattavaksi ja hallittavaksi. Toisena ongelmana on, että missä asiakaskohtaisia asetuksia ja säännöksiä pidetään tallessa, ja kuinka ne saadaan sovelluksen käyttöön. Kolmantena ongelmana, jota tässä opinnäytetyössä lähdetään ratkaisemaan, on eri raportointipohjien versioerot ja kuinka niiden tarve voidaan parhaiten ottaa huomioon toimeksiantajan tuotesovelluksessa.

Tämän opinnäytetyön toimeksiantaja on Profit Software Oy. Profit Software on vuonna 1992 perustettu finanssialalle järjestelmäratkaisuja ja konsultointipalveluita toimittava yritys.

Työn pääpaino on sovelluksen logiikan eli backendin teossa, ja työssä tutustutaan tarkemmin Microsoftin ASP.NET Core Frameworkkiin, sekä Design Patterneihin, joista tarkemmin tutustutaan Strategia Patterniin.

Opinnäytetyön malliprojektiin valittuja ratkaisutapoja pyritään testaamaan skaalattavuuden, yleisen hallinnan ja ylläpidon osalta niin tietokantaratkaisun kuin sovelluksen asiakaskohtaisen räätälöintienkin kannalta.

Itse opinnäytetyöraportti on jaettu kahteen osaan. Ensimmäisessä osassa kerrotaan tarkemmin tutkituista ratkaisuvaihtoehdoista ja siitä miksi ne olisivat hyvät vaihtoehdot ratkaisemaan kyseiset ongelmat. Raportin toisessa osassa kerrotaan tarkemmin, kuinka rakennettu mallisovellus toimii, ja kuinka tutkittuja ratkaisuvaihtoehtoja päädyttiin soveltamaan toimivan lopputuloksen saamiseksi. Jälkimmäisessä osassa raporttia myös pohditaan, olivatko valitut ratkaisutavat hyvät skaalattavuuden sekä hallinnan kannalta, ja mitä pitää ottaa huomioon toimeksiantajan projektissa, jos näitä ratkaisutapoja päädytään hyödyntämään.

1 OPINNÄYTETYÖN LÄHTÖKOHDAT

Tämä luku on jaettu kolmeen osaan, joissa kerron tämän opinnäytetyön lähtökohdista, jotka johtivat työn tarpeeseen toimeksiantajan puolelta. Esittelen myös opinnäytetyön tavoitteen ja tarkoituksen.

1.1 Asiakaskohtaisen räätälöinnin tarve

Tarve asiakaskohtaiselle räätälöinnille luottoluokittajaraportilla syntyi toimeksiantajayrityksen tarpeesta kehittää monipuolinen luottoluokittajaraportointitapa, joka on kustomoitavissa eri asiakkaiden tarpeiden mukaan.

Tämän opinnäytetyön toimeksiantajana toimi Profit Software Oy. Profit Software Oy on vuonna 1992 perustettu yritys, joka on erikoistunut finanssialan järjestelmäratkaisuihin sekä konsultointiin. Yrityksellä on toimipaikkoja Suomessa, Virossa sekä Ruotsissa. Tämä opinnäytetyö sijoittui finanssialan sekä web-ohjelmoinnin aihealueille, ja tässä opinnäytetyössä keskityttiin ratkaisemaan yrityksen tarvetta pystyä raportoimaan ulkoisille sidosryhmille, kuten luottoluokittajalle, yrityksen tuotteen eri osa-alueita asiakaskohtaisesti räätälöitynä.

Toimeksiantajayrityksellä on oma tuote, jota kehitetään olemassa olevan asiakasprojektin pohjalta. Tähän tuotteeseen on tarkoitus ottaa huomioon asiakaskohtainen räätälöinti luottoluokittajaraportoinnin osalta, luoden siitä mahdollisimman skaalattavan, sekä helposti ylläpidettävän osa-alueen tuotteeseen. Tämän avulla pyritään helpottamaan tuotteen käyttöönottoa ja ylläpitoa eri asiakkaiden osalta.

1.2 Luottoluokittajaraportti

Luottoluokittajaraportti on nimensäkin perusteella yksinkertaisesti raportti, jonka pankki lähettää luottoluokittajalle. Tätä opinnäytetyöraporttia ymmärtääkseen ei tarvitse tietää luottoluokittajaraportoinnista muuta kuin sen, että malliratkaisun esimerkkinä käytettyjä Standards & Poor’s (S&P) -raportteja on toimeksiantajayrityksessä käytössä kaksi erilaista, Loan by Loan ja Covered Bond Monitor (CBM).

Näille raporteille raportoidaan muun muassa luottoja ja vakuuksia. Luottoluokittajaraporteilla on jokaisella oma Excel-pohjansa, jonka päälle sovellus tallentaa saadut datatiedot. Näitä pohjia voi olla eri versioita asiakkaista riippuen.

Tässä opinnäytetyössä hyödynnettiin yhtä kuvitteellista esimerkkiä sekä yhtä todellista esimerkkiä itse raportoitavien datojen erojen osoittamisessa, sekä yhtä kuvitteellista erimerkkiä eri raporttipohjien tarpeesta.

Todellisessa esimerkissä datan raportointieroissa raportille hyödynnettiin yhtä vakuustietoa yhdelle luotolle. Jokaisella asiakkaalla on oma tapansa valita, minkä yhden vakuuden tiedot tulevat näkyviin Excel tiedoston kyseisen luoton riville. Vaihtoehtoina voivat olla esimerkiksi isoin vakuus, ensimmäinen vakuus tai pienin vakuusnumerollinen vakuus, jota luotolta löytyy. Tätä esimerkkiä käytettiin myös niin sanottuna globaalina esimerkkinä, jonka tarkoituksena oli tutkia, kuinka haluttua vakuutta voisi käyttää uudelleen molemmissa esimerkkiraporteissa, jotka olivat kuvitteellisia esimerkkejä Loan by Loan- sekä CBM-raporteista.

Kuvitteellisena esimerkkinä datan raportointieroista käytettiin esimerkkiä, jossa asiakkaan tarpeen mukaan raportilla joko näytetään vakuuden käyttöarvo tai ei. Tätä esimerkkiä käytettiin tutkiessa, miten haluttu ratkaisu skaalautuu tarvittaessa uusien rajaustapausten lisääntyessä.

Raporttipohjien erojen esittämiseen käytettiin kuvitteellista esimerkkiä, jossa maksimi rivimäärä vaihtui pohjien välillä. Tämä tarkoittaa käytännössä sitä, että jos esimerkiksi CBM-raportissa olisi ollut alkuperäisessä pohjassa käytössä 180 riviä, joille on tarkoitus saada dataa, niin uudessa pohjassa käytössä olisikin 190 riviä. Tällä esimerkillä oli tarkoitus tutkia, kuinka tämänlaisia pohjien välisiä eroongelmia ratkaistaisiin parhaiten.

1.3 Käytetyt teknologiat

Opinnäytetyön sovellusmallin teossa hyödynnettiin Microsoftin Visual Studiota, sekä ASP.NET corea. Nämä kaksi pääteknologiaa valittiin toimeksiantajayrityk-

sen tuoteprojektin perusteella. Toimeksiantajan tuoteprojekti käyttää .NET ympäristöä, joten tämän opinnäytetyön malliprojektin kannalta ehkä tärkein teknologia oli C# ohjelmointi-kieli, jota ASP.NET hyödyntää.

Mallisovellusta tehdessä tarvittiin myös muita teknologioita. UML-diagrammien tekoon hyödynnettiin https://app.diagrams.net/ -sivustoa. Näiden diagrammien tarkoituksena on havainnollistaa esimerkkejä esimerkiksi valituista strategiamenetelmistä. Sovelluksen API rajapinnan testaukseen käytettiin Postman-sovellusta, jota hyödynnettiin tietokantayhteyksien ja raporttiasetuksien muutosten vaikutuksia testatessa.

1.4 Opinnäytetyön tavoite ja tarkoitus

Toimeksiantajayrityksen omaan tuotteeseen on tarkoitus hyödyntää asiakasprojektissa luotuja valmiita sovelluskoodeja. Liittämättä kyseistä asiakasprojektia opinnäytetyöhön, työssä oli tarkoitus pitää mielessä jo olemassa oleva sovelluspohja ja tutkia sekä suunnitella tapoja ratkaista asiakaskohtaisten raportointipäätöksien ongelmia. Tällä tutkinnalla pyrittiin saamaan taustatutkimusta siitä, kuinka tuoteprojektin luottoluokittajaraporttien asiakaskohtaiset räätälöintiongelmat saisi parhaiten ratkaistua.

Tarkempana tavoitteena oli löytää ratkaisu kolmeen ongelmaan. Ensimmäisenä ongelmana oli missä asiakaskohtaisia asetuksia ja säännöksiä pidettäisiin tallessa, ja kuinka ne saataisiin sovelluksen käyttöön. Toisena ongelmana oli, miten asiakaskohtainen räätälöinti toteutettaisiin helposti skaalattavaksi ja hallittavaksi. Kolmas ongelma, johon pyrittiin etsimään ratkaisu, oli eri raportointipohjien versioerot ja kuinka niiden tarve voidaan parhaiten ottaa huomioon sovelluksessa.

Opinnäytetyön toimeksiantajayrityksen olemassa olevan sovelluspohjan takia ongelmiin pyrittiin löytämään ratkaisu ensisijaisesti muulla tavalla kuin suunnittelemalla ja toteuttamalla koko raportointia uudelleen alusta asti. Tässä opinnäytetyössä pyrittiin keskittymään eri tapoihin parantaa olemassa olevaa tuotekoodia tutkimalla eri tyylejä ratkaista uudelleenkäytettävyys sekä skaalattavuusongelmat. Taustatutkimusten perusteella tässä opinnäytetyössä luotiin malliprojekti, joka hyödynsi valittuja ratkaisutapoja räätälöintiongelmiin.

Toteutetussa malliprojektissa oli tarkoitus käydä ilmi suositellut ratkaisutavat keskeisiin ongelmiin, ja tässä opinnäytetyöraportissa tarkoituksena on kertoa tarkemmin valituista ratkaisutavoista ja tutkia, olivatko nämä valitut tavat sopivat ongelmien ratkaisuun, ja kuinka ne voisivat hyödyntää asiakasta tulevaisuudessa.

Luotu mallisovellus on toimiva ratkaisu, jota pystytään hyödyntämään taustamateriaalina tuotteen luottoluokittajaraportointia rakentaessa. Tämän opinnäytetyön tarkoituksena on antaa mallia ja mahdollisia ratkaisuja asiakaskohtaiseen räätälöintiin. Opinnäytetyön tuloksista voi olla suurta hyötyä monille yrityksille tai yksilöille, jotka räätälöivät tuotteidensa osa-alueita asiakaskohtaisiksi.

2 MICROSOFTIN ASP.NET-YMPÄRISTÖ JA DESIGN PATTERNIT

Tämän opinnäytetyön mallisovelluksen päätyökaluksi valittiin ASP.NET, sekä tämän myötä Microsoft Visual Studio. Opinnäytetyön mallisovelluksen skaalattavuuden, hallittavuuden sekä eri Excel-versiopohjien vaatimisongelmien ratkaisemiseen valittiin Design Patternit.

Tässä luvussa tutustutaan yleisesti näihin opinnäytetyössä tutkittuihin ja valittuihin ratkaisutapoihin, ja siihen, miksi juuri ne olisivat hyvät ratkaisut tätä opinnäytetyötä ja sen ongelmia varten.

2.1 ASP.NET

.NET Framework on Microsoftin kehittämä ohjelmistokomponenttikirjasto, joka on ilmainen, avointa lähdekoodia hyödyntävä kehittäjäalusta (What is .NET? n.d.). .NET-kehittäjäalustan avulla ohjelmoija pystyy käyttämään monia kieliä, editoreja sekä kirjastoja rakentaessaan verkko-, mobiili-, työpöytä- sekä IoT -sovelluksia. Toimeksiantajan sovelluksen pääkieli on C#-ohjelmistokieli, jota .NET hyödyntää. C# on yksinkertainen, moderni, oliokeskeinen ja tyyppiturvallinen ohjelmointikieli. (What is .NET? n.d.)

ASP.NET laajentaa .NET-kehitysalustaa erityisesti verkkosovellusten tekemisessä työkaluilla, sekä kirjastoilla (ASP.NET A framework for building web apps and services with .NET and C# n.d.). Toimeksiantajan tuote on web-pohjainen sovellus, joka on rakennettu hyödyntäen .NET alustaa. Tämän opinnäytetyön tarkoitus oli tuoda esille mahdollisimman selkeästi ja yksinkertaisesti käytetyt ja suositellut teknologiat, sekä ratkaisut toimeksiantajalle, joka johti siihen, että opinnäytetyön malliprojekti rakennettiin toimeksiantajan tuotteen mukaisesti ASP.NET Frameworkilla.

ASP.NET Framework hyödyntää monipuolisesti Design Patterneja (MSDN, 2005), ja koska toimeksiantajan tuote perustuu olemassa olevaan asiakasprojektiin, tässä opinnäytetyössä pyrittiin keksimään tapoja ratkaista räätälöinnin ongelmat parantamalla olemassa olevaa koodia. Tällä tavoin pyrittäisiin säästämään

rahaa sekä resursseja tulevaisuudessa uusien asiakkaiden kanssa, kun tuoteratkaisu olisi helposti skaalattava sekä hallittava. Koska toimeksiantajan tuote hyödyntää vahvasti Design Patterneja ennestään, johti tämä päätökseen tutkia asiakaskohtaisen räätälöinnin mahdollista ratkaisemista niillä.

2.2 Design Patternit

Tutkinta aloitettiin asiakaskohtaisen räätälöinnin perusteista. If-else, sekä switch ovat ohjelmoinnin perusteita, joita pystyisi yksinään hyödyntämään räätälöinnissä. Jokaiselle asiakaskohtaiselle tarpeelle tulisi yksinkertaisesti valinta kyseisen asiakkaan kohdalla. Jos asiakas X, niin tee näin, jos asiakas Y, teekin näin.

Nämä olisivat ns. yksinkertaisin ja nopein ratkaisu asiakaskohtaisen räätälöinnin ongelmaan, jota on mahdollista hyödyntää erityisesti pienissä sovellusprojekteissa. Toimeksiantajan sovelluksesta kuitenkin näkee nopeasti, että if-else sekä switch eivät ole yksinään uudelleenkäytettäviä, eikä niitä ole helppoa ylläpitää, koska asiakaskohtaiset tarpeet jakautuvat useisiin eri paikkoihin. Suuressa sovelluskokonaisuudessa uusien vaatimusten lisääminen muuttuu nopeasti hankalaksi, ja asiakaskohtaisten räätälöintiratkaisujen muokkaaminen käy vaikeaksi ilman, että aikaisempaa koodia rikotaan tai joudutaan muokkaamaan tarpeettomasti.

Ohjelmistokehittäjä Christopher Lasater (2007) kuvailee kirjassaan Design Patternien, eli suunnittelumallien, olevan työkaluja olemassa olevan ohjelmistokoodin parantamiseksi. Hän sanoo niiden auttavan ohjelmistokehittäjää ylläpitämään, rakentamaan ja toteuttamaan koodia yksinkertaisesti ja helposti. (Lasater, 2007, Why Pattern?)

Erityisesti koodin ylläpitäminen, on asia, jota tulee huomioida asiakaskohtaisessa räätälöinnissä. Luottoluokittajaraportin asiakaskohtainen räätälöinnin tarve eri asiakkailla alkuperäisten päätösten jälkeen tapahtuu harvoin, jolloin asiakaskohtaisen koodin ylläpitäminen ja toteuttaminen on tärkeä siirtää erilliseen paikkaan, josta sitä on helppo hallita sekä muokata tarpeen mukaan.

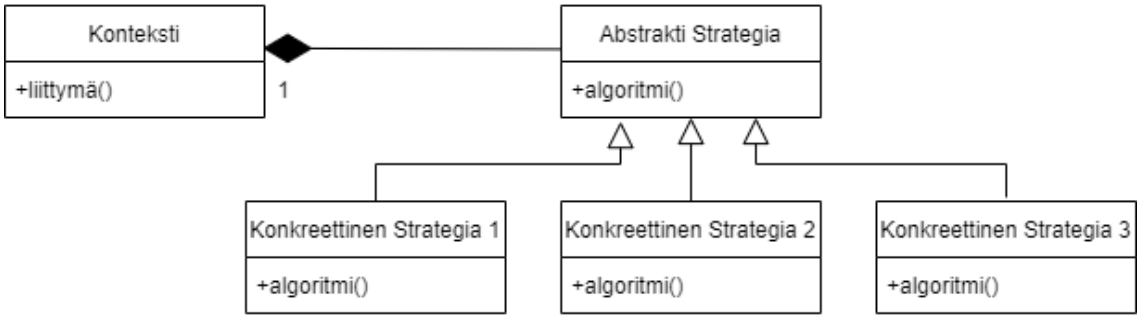
Suunnittelumallit jaetaan yleensä kolmeen eri kategoriaan, Creational Pattern, Behavioral Pattern sekä Structural Pattern (Gamma E., Helm R., Johnson R., Vlissides J., 1995, Organizing the Catalog). Alun tutkimusten perusteella tehtiin syvempää tutkimusta Behavioral Patterneihin, eli käyttäytymismalleihin, joiden tarkoitus on helpottaa ohjelmaluokkien sekä algoritmisten laskelmien välistä viestintää hyödyntäen inheritancea, eli perintää ohjelmakoodin hallintaan. (Lasater, 2007, 3 Behavioral Patterns)

Käyttäytymismalleja on monenlaisia. Opinnäytetyössä tarvittiin käyttäytymismalli, jolla pystyisi hallitsemaan mitä asiakkaan haluamaa luottoluokittajaraportin raportointisääntöä käytettäisiin, ilman, että tuotetta pitäisi muokata ja muutoksia asentaa uudelleen. Tämän tarpeen perusteella esiin nousi Strategy Pattern. Kyseistä käyttäytymismallia voi hyödyntää, kun sovelluksen ajon aikana pyritään tekemään päätöksiä milloin haluttu algoritmi otetaan käyttöön (Lasater, 2007, 3 Behavioral Patterns). Tämän perusteella kyseistä käyttäytymismallia pystyisi hyödyntämään asiakaskohtaisessa räätälöinnissä esimerkiksi tietokantataulujen kautta tehtyjen säännösmuutosten perusteella ilman, että asiakkaan ohjelmaa pitäisi asentaa uudelleen pienten muutostarpeiden päivittyessä.

2.2.1 Strategy Pattern

Strategy Patternin, eli strategiamallin, perusperiaate on koota tarvittavat algoritmit omiin luokkiinsa, joita voi vaihdella tarpeen mukaan keskenään. Malli koostuu kolmesta pääkomponentista, joiden avulla strategia luodaan: strategia, konkreettinen strategia sekä konteksti. (Lasater, 2007, 3 Behavioral Patterns)

Peruskäytössä strategia on yleensä abstrakti liittymä, joka yhdistää kaikki halutut algoritmit toisiinsa, kun taas konkreettiset strategiat ovat toteutettuja luokkia, jotka sisältävät jokaisen halutun algoritmin logiikat, ja konteksti on objekti, jota hyödynnetään liitettäessä strategia haluttuun koodiin (kuva 1).



KUVA 1. Strategiamallin peruskäyttö

Tämän esimerkkikuvion perusteella (kuva 1), pystyy hahmottamaan, että strategiat ovat siis hyödyllisiä, kun algoritmeja halutaan käyttää eri yhteyksissä yhden liittymän avulla. Strategian esimerkkinä voisi käyttää luottoluokittajaraportin asiakaskohtaista räätälöintiä, josta esimerkkiongelmana voisi käyttää jo aikaisemmin mainittua luoton yhtä vakuutta, joka valitaan eri perustein eri asiakkaan mukaan. Tämän esimerkkiperusteen mukaan tulisi yhteensä kolme eri konkreettista strategiaa, isoin vakuus, ensimmäinen vakuus sekä pienin vakuusnumerovakuus. Tästä esimerkistä kerrotaan tarkemmin myöhemmässä luvussa, jossa on kerrottu mallisovelluksen rakentamisesta.

Strategiamalli ei kuitenkaan yksin ratkaise koko asiakasräätälöinnin ongelmaa, vaan sen ratkaisemiseen tarvitaan myös tapa, jolla saadaan asiakasräätälöinnin asetukset itse strategioiden hyödynnettäviksi. Tätä varten tutkittiin ASP.NET sovelluksessa tietokantojen sekä API-rajapinnan tekoa.

2.2.2 Dependency Injection

Yksinkertaisesti sanottuna Dependency Injection (DI) on ohjelmistosuunnittelumalli, jolla pyritään saavuttamaan Inversion of Control luokkien ja niiden riippuvuuksien välillä (Microsoft, 2021). Dependency Injectionia hyödynnetään usein API rajapintojen rakentamiseen, jotta sovellus pystyy hyödyntämään saatua dataa. Toimeksiantajayrityksen tuotteessa hyödynnetään monipuolisesti Dependency Injectionia datan saamiseen tietokannoista, ja tähän opinnäytetyöhön rakennettiin pienikokoinen API-rajapinta simuloimaan sitä, sillä rajapinta ei tule muuttumaan tuotteessa, ja uudenlaisen API-rajapinnan teko ei olisi hyödyllinen opinnäytetyön osalta.

Opinnäytetyössä yhtenä ongelmana oli ratkaista se, mihin asiakaskohtaisen räätälöinnin asetukset saadaan talteen. Taustatutkimuksen jälkeen päädyttiin hyödyntämään Dependency Injectionia, API-rajapintaa, sekä tietokantoja asetuksien tallentamiseen, muokkaamiseen sekä ylläpitämiseen.

3 TIETOKANTATAULUJEN TARVE

Strategiamalli tarvitsi API-rajapintaa asetuksien saantia varten jokaisen asiakkaan kohdalla. API-rajapinta edellytti jonkun ratkaisun, josta se saa halutut asetukset. Tässä luvussa käydään läpi valittu ratkaisutapa, eli tietokantataulut, joka valittiin asiakaskohtaisen räätälöinnin asetuksien säilömiseen.

3.1 Miksi tietokantataulut?

Asiakaskohtaisia asetuksia tulisi pystyä lisäämään aina tarpeen mukaan, ja niitä on tarkoitus ylläpitää, ja vaihdella muutoin kuin käyttöliittymän kautta. Tämä rajaa asetuksien saannin joko konfiguraatiotiedostoihin tai tietokantoihin.

Toimeksiantajayrityksen tuotteesta pyritään tekemään Multi-tenant SaaS ratkaisu, jossa jokaisella asiakkaalla on omat tietokantansa. Tämä on yksi kolmesta eri tavasta eristää asiakastietoja pilviratkaisuissa (Yaish, H., Goyal, M. 2013, 870). Tuotteen ollessa Multi-tenant SaaS ratkaisu, antaa se hyvän syyn ylläpitää jokaisen asiakkaan omat asiakaskohtaiset räätälöinnit omissa tietokannoissaan.

Tietokannat tuovat varmuutta, sekä joustavuutta datan sekä asetuksien säilömiseen. Mikäli tietokantataulut suunnitellaan tarkoin sovelluksen kehitysvaiheessa, on niiden käyttö hyvä valinta sovellukselle. Tietokantataulut antavat mahdollisuuden tallentaa ja muokata asiakaskohtaisia sääntöjä niin, että esimerkiksi asiakkaan haluamista muutoksista jää jälki tietoihin. Tämän pystyisi ratkaisemaan esimerkiksi lisäämällä uuden data-rivin tauluun, jossa tallennetaan muokkauspäivä tarpeen mukaan. Tietokantataulut myös antavat sen edun, että tietokantaa pystyy päivittämään ilman, että tuotetta tarvitsisi asentaa ja päivittää uudelleen jokaisen pienen muutoksen yhteydessä.

Tietokantatauluja pitää tarvittaessa päivittää SQL-scriptien avulla, joka voi vaatia sovelluksen uudelleen päivittämisen, mutta tämä on silti yksinkertaisempi tapa ylläpitää tarvittavia asiakasasetuksia, sillä taulut ovat API-rajapinnan käytössä, josta uusia muutoksia ja lisäyksiä on tarpeen tullen helppo päivittää, ilman, että sovelluksen muuhun kokonaisuuteen kosketaan. Tietokantataulut antavat myös

sen vahvuuden, että niistä on helppo katsoa, mitkä asetukset ovat käytössä sen sijaan, että ne joutuisi etsimään sovelluksen konfiguraatioista.

3.2 Skaalattavuus ja uudelleenkäyttö

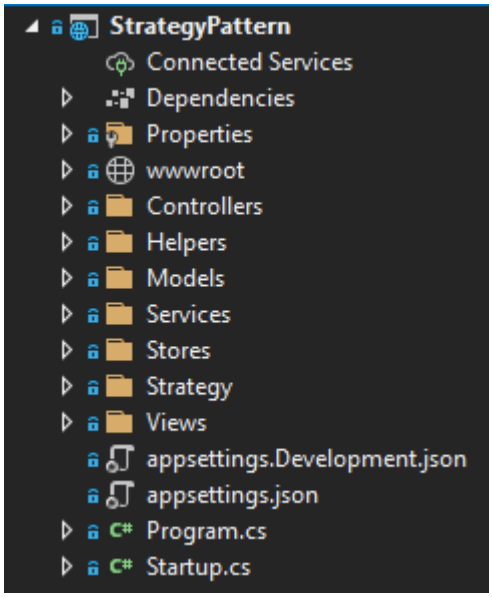
Tässä opinnäytetyössä keskityttiin asiakaskohtaisten asetusten skaalattavuuteen ja uudelleenkäytettävyyteen sekä tietokannassa että itse sovelluskoodissa. Luottoluokittajaraportointi on laaja kokonaisuus, jossa löytyy sekä yhteneväisyyksiä säännöksien välillä, että suuria eroja, joka tuo lisävaikeutta tietokantojen suunnitteluun, ja ylläpitoon. Asiakaskohtaisten sääntöjen tarkoitus on, että ensimmäisessä tuoteversiossa niitä on vain muutamia. Uusia sääntöjä lisätään tarvittaessa uusien asiakkaiden ja asiakastarpeiden mukaan, ja jotta taulut olisivat skaalattavia sekä uudelleenkäytettäviä, tarvitsee ne suunnitella huolella alun perin. Luottoluokittajaraportoinnin laajuuden takia tietokantojen käyttö vaatii kattavat dokumentaatiot, jotta niitä pystyy ylläpitämään, ja muokkaamaan helposti muutostarpeiden päivittyessä.

4 RAPORTOINTISOVELLUSMALLIN RAKENTAMINEN

Tässä opinnäytetyössä luotiin malliprojekti simuloimaan valittuja ratkaisutapoja eri asiakaskohtaisten räätälöintien ongelmiin. Tässä luvussa kuvaillaan tarkemmin toteutettua mallisovellusta, sen avulla saatuja ratkaisuehdotuksia, ja toteutuksen testausvaiheita. Tässä luvussa myös tutkitaan sekä pohditaan, olivatko valitut ratkaisutavat hyviä kyseisten ongelmien ratkaisemisessa.

4.1 Raporttisovelluskokonaisuus lyhyesti

Raportointisovellusmalli jaettiin kahteen eri osaan hyödyntäen ASP.NET Coren Web App (MVC) mahdollisuutta. Valitussa tavassa pohjaprojekti jaetaan kahteen osaan. Ensimmäinen osa on Views, jossa rakennetaan komponentit, jotka käyttäjä näkee (frontend). Toinen osa, johon tässä projektissa keskityttiin, on sovelluksen logiikka, joka tekee taustatyön (backend) käyttöliittymää varten. Tämä koostuu Controllers, Helpers, Models, Services, Stores sekä Strategy kokonaisuuksista, jotka yhdessä luovat tarvittavat taustalogiikat, joita Views komponentit hyödyntävät (kuva 2).



KUVA 2. Projektin rakenne

Raporttisovellukseen rakennettiin API-rajapinta tietokantayhteyttä ja datan saantia varten.

4.1.1 ASP.NET Core Web App (MVC)

Opinnäytetyömalliprojektin rakentamisessa keskityttiin raportin asiakaskohtaiseen räätälöintiin backend puolella.

Vaikka mallisovelluksessa keskityttiinkin sovelluksen taustalogiikkaan, sovellus tarvitsi myös yksinkertaisen käyttöliittymän, josta sovelluksen käyttäjä pystyi helposti näkemään tietokantojen datat. Käyttöliittymältä oli myös tarve pystyä lataamaan kaksi eri Excel-tiedostoa, jotka vastasivat kuvitteellisesti S&P raportoinnin Loan by Loan sekä CBM raportteja.

Näiden kahden päätarpeen pohjalta sovelluksen pohjaksi valittiin ASP.NET Core Web App (MVC). ASP.NET Core MVC on rikas kehys, jota käytetään verkkosovellusten rakentamisessa. MVC (Model-View-Controller) on suunnittelumalli, jota hyödyntämällä sovellukseen pystyy rakentamaan käyttöliittymän sekä API-rajapinnan helposti. (Overview of ASP.NET Core MVC. 2021.)

MVC kehyksen avulla rakennettiin yksinkertainen web- käyttöliittymä, josta käyttäjä näkee eri taulukoihin listattuna kaikki saatavilla olevat tietokantataulujen datat. Käyttöliittymällä on myös kaksi nappia, joita painamalla voi tallentaa eri asetuksilla rakennettuja Excel tiedostoja. (kuva 3) Näiden Excel-tiedostojen avulla pyrittiin testaamaan sovelluksen taustalogiikkaa ja sen toimivuutta.

StrategyPattern Home Privacy

Loans

LoanId	JaljellaolevaPaaoma	AlkupErapaiva	Erapaiva	NostamatonSaldo
11	123445,00	11.10.2022 0.00.00	11.10.2023 0.00.00	1222,00
2	1111,00	11.10.2023 0.00.00	11.10.2024 0.00.00	1222,00
3	3333,00	11.10.2020 0.00.00	11.10.2024 0.00.00	1222,00

Collaterals

CollateralId	LoanId	KaypaArvo	SopimusAika
1	11	33333,00	11.10.2021 0.00.00
2	11	22222,00	15.10.2021 0.00.00
3	11	11111,00	10.10.2021 0.00.00
4	2	444444,00	10.9.2021 0.00.00
5	2	555555,00	10.10.2021 0.00.00
6	3	12344444,00	10.10.2021 0.00.00

Export to CSV One Export to CSV Two

Customer Configurations

CustomerConfigurationId	CreateDate	WantedCollateralSelect	ShowKaypaArvoInRow
1	10.10.2021 0.00.00	BiggestCollateral	ShowNothing

Customer ConfigurationsVersions

CustomerConfigurationVersionId	CreateDate	HowManyRowsShownInExcel
1	10.10.2021 0.00.00	Two
2	15.10.2021 0.00.00	Three

Customer ConfigurationsTemplates

CustomerConfigurationTemplated	CustomerConfigurationVersionId	CreateDate
1	1	10.10.2021 0.00.00
2	2	15.10.2021 0.00.00

KUVA 3. Sovelluksen web-käyttöliittymänäkymä

Asiakaskohtaisia asetuksia ei pysty muokkaamaan käyttöliittymältä, vaan niitä pystyy muokkaamaan ainoastaan tietokantaa päivittämällä. Tähän ratkaisuun päädyttiin sen takia, että toimeksiantajayrityksen tuotteen raportointisääntöjä ei pysty muokkaamaan käyttöliittymältä missään vaiheessa, vaan asetuksia tullaan muokkaamaan ainoastaan asiakkaan erillisistä pyynnöistä, joita sovelluksen alkukäyttöönoton jälkeen tulee erittäin harvoin.

4.2 Asiakaskonfiguraatioiden saanti tietokannan avulla

Raportointisovellusmallin taustaprosessin suunnittelu aloitettiin suunnittelemalla tietokantakokonaisuus. Ensin suunniteltiin kaksi tietokantataulua käytettävälle datalle. Esimerkissä luottoluokittajaraporteilla raportoitiin luottoja ja vakuuksia, ja tässä opinnäytetyössä hyödynnettiin niitä yksinkertaistettuna esimerkkeinä datasta luomalla kummallekin omat taulunsa. Käytetyissä raporttiesimerkeissä tärkeintä oli demonstroida esimerkin mukaisesti luoton suhde vakuuteen, ja sitä, kuinka kyseistä dataa voitaisiin raportoida eri tavalla eri asiakkaiden ja raporttipohjien kohdalla.

Asiakaskonfiguraatiot päätettiin jakaa kolmeen eri tietokantatauluun. Ensimmäisen taulun idea oli kerätä kaikki eri raportointimahdollisuudet yhteen tietokantatauluun, toisessa tietokantataulussa oli tarkoitus ylläpitää jokaisen eri raporttipohja-Excelin tarpeet ja kolmannessa taulussa oli tarkoitus ylläpitää sen hetkistä valittua Excel-pohjaa.

4.3 Vakuudet

Raportointiprojektissa päädyttiin käyttämään pääesimerkkinä oikeaa luoton ja yhden vakuuden suhdetta. Tässä esimerkissä jokaisella luotolla voi olla nollasta moneen vakuutta. Jokaisen luoton tiedot raportoidaan riveittäin Loan by Loan raportille. Ongelmana on kuitenkin se, että yhden luoton riville saa vain yhden vakuuden tiedot näkyviin. Tämä on ongelma, jonka jokainen toimeksiantajayrityksen asiakas ratkaisee haluamallaan tavalla, ja usein tämän ongelman ratkaisu on tehty eri tavalla asiakkaasta riippuen.

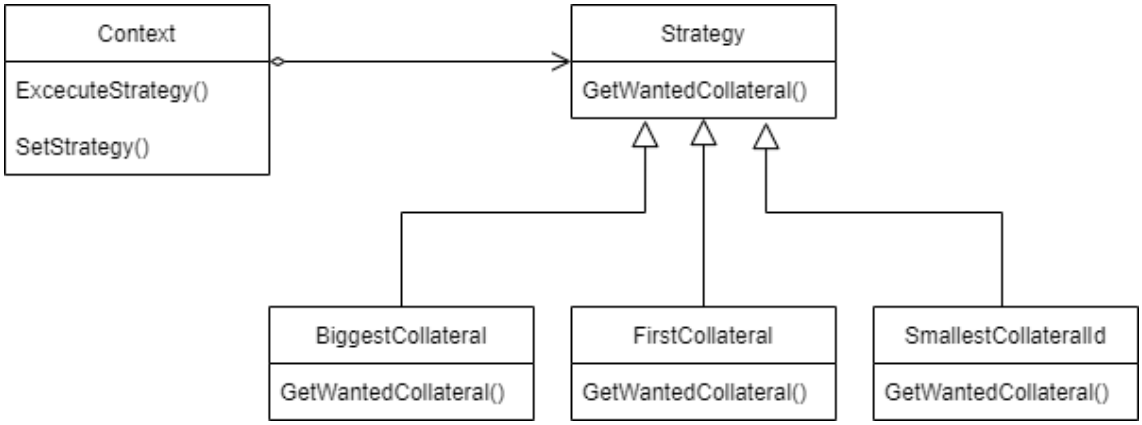
Tässä opinnäytetyössä päädyttiin käyttämään kolmea eri vaihtoehtoa halutun vakuuden raportointitapana. Esimerkkeinä käytettiin isointa vakuutta, ensimmäistä vakuutta sekä pienintä vakuusnumerollista vakuutta, ja ongelman ratkaisutapana käytettiin Strategy Patternia, jota hyödynnetään olio-ohjelmoinnissa koodin parantamisessa ja sen ylläpidon helpottamisessa.

Asiakaskohtaiset asetukset saatiin tietokannasta numeroiden avulla, jotka yhdistyivät enum tiedostossa luettavaan muotoon (kuva 4). Asetuksista löytyi alun perin vain BiggestCollateral, eli suurin vakuus, mutta strategianmallin skaalattavuutta testatessa, lisättiin kaksi uutta konkreettista strategiaa.

```
6 references
public enum WantedCollateralSelect
{
    BiggestCollateral = 1,
    FirstCollateral,
    FirstCollateralIdCollateral,
}
```

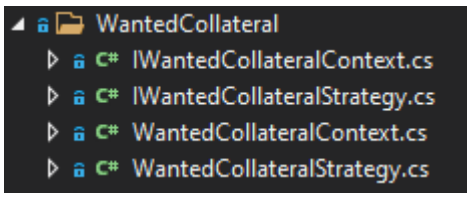
KUVA 4. Enum tiedosto kaikista mahdollisista vakuuden saantitavoista

Strategiamallista suunniteltiin yksinkertainen UML luokkakaavio, jota hyödynnettiin sovelluksen koodia rakennettaessa (kuva 5).



KUVA 5. UML kaavio strategiamallin käytöstä yhden vakuuden ongelman ratkaisussa

Työssä luotiin neljä eri tiedostoa yhtä strategiaa varten (kuva 6). Jokaisella tiedostolla oli omat tehtävänsä, ja tiedostonimet jaettiin konteksteihin sekä strategioihin. Sekä kontekstille että strategialle tehtiin interface luokka, joka erotettiin isolla i-alkukirjaimella tiedostonimen edessä. Interface luokka on luokka, jota käytetään periytymiseen.



KUVA 6. Kansiorakenne halutun vakuuden strategiaa varten

Jotta strategia saatiin toimimaan, luotiin halutun vakuuden konteksti. Kontekstissa oli kaksi eri metodia SetStrategy, sekä ExcecuteStrategy. Metodien nimistäkin voi päätellä, että ensimmäisenä mainittua metodia käytettiin uuden strategian asettamiseen, ja jälkimmäistä, ExecuteStrategyä, kutsuttiin kun haluttiin suorittaa asetettu strategia (kuva 7).

```
public class WantedCollateralContext : IWantedCollateralContext
{
    private IWantedCollateralStrategy _strategy;

    0 references
    public WantedCollateralContext() { }

    0 references
    public WantedCollateralContext(IWantedCollateralStrategy strategy)
    {
        _strategy = strategy;
    }

    5 references
    public void SetStrategy(IWantedCollateralStrategy strategy)
    {
        _strategy = strategy;
    }

    2 references
    public Collateral ExcecuteStrategy(List<Collateral> collateralList)
    {
        return _strategy.GetWantedCollateral(collateralList);
    }
}
```

KUVA 7. Halutun vakuuden konteksti strategian asettamista ja suorittamista varten

Konteksti kuitenkin tarvitsi vielä strategian, jota kutsua. Strategiamallin tapaan työssä luotiin abstrakti strategia (kuva 8), jonka tarkoitus oli yhdistää kaikki kolme haluttua konkreettista strategiaa toisiinsa, ja jota konteksti suorittaessaan kutsui.

Interface luokka, eli abstracti strategia tarvitsi yhden metodin, GetWantedCollateral, jonne lähetettiin lista kaikista vakuuksista. Lähetetyistä vakuuksista konkreettisten strategioiden perusteella valittiin yksi vakuus palautettavaksi.

```
public interface IWantedCollateralStrategy
{
    4 references
    Collateral GetWantedCollateral(List<Collateral> collateralList);
}
```

KUVA 8. Strategy Interface

Jotta strategiasta olisi hyötyä, luotiin ensin yksi konkreettinen strategia, BiggestCollateral, eli suurin vakuus, joka peri GetWantedCollateral metodin, ja toteutti sen logiikan, eli palautti vakuuslistan suurimman käypäarvon mukaan yhden vakuuden (kuva 9).

Opinnäytetyön yhtenä tavoitteena oli tutkia valitun ratkaisutavan skaalattavuutta ja uudelleen käytettävyyttä eri tavoilla, joten työ toteutettiin ensin lisäämällä yksi konkreettinen tapa strategiamallia hyödyntäen, jonka jälkeen lisättiin loput kaksi ratkaisua (kuva 9, ja kuva 12).

```
public class BiggestCollateral : IWantedCollateralStrategy
{
    2 references
    public Collateral GetWantedCollateral(List<Collateral> collateralList)
    {
        return collateralList.OrderByDescending(collateral => collateral.KaypaArvo).FirstOrDefault();
    }
}
```

KUVA 9. Konkreettinen strategia suurimman halutun vakuuden saantia varten

Yksinkertaisen strategian tekemisen jälkeen pyrittiin ratkaisemaan seuraava ongelma, jotta luodusta strategiasta saatiin kaikki irti. Haluttua vakuutta käytettäisiin todellisuudessa monessa eri raportissa, ja paikassa, ja luodussa mallisovelluksessa sitä hyödynnettiin molemmissa Excel-tiedostoissa.

Asiakasasetukset tarvitsivat siis uudelleenkäytettävän tavan, jolla katsoa mitä strategiaa tulisi käyttää. Tämä ratkaistiin tekemällä apuluokka CustomerConfigurationUsedInMultiplePlacesHelper (kuva 10). Apuluokan ideana oli, että konfigu-

raatiokutsuja, jotka löytyivät tästä luokasta, käytettiin uudelleen useassa paikassa. Luokan GetCollateral metodi asetti halutun strategian saadun asiakaskonfiguraation perusteella oikeaksi, ja sen perusteella suoritti vakuuksien rajauksen ja palautti halutun vakuuden.

```
2 references
public class CustomerConfigurationsUsedInMultiplePlacesHelper
{
    2 references
    public static Collateral GetCollateral(CustomerConfiguration customerConfiguration,
        List<Collateral> collaterals, IWantedCollateralContext context)
    {
        switch (customerConfiguration.WantedCollateralSelect)
        {
            case WantedCollateralSelect.BiggestCollateral:
                context.SetStrategy(new BiggestCollateral());
                break;
            case WantedCollateralSelect.FirstCollateral:
                context.SetStrategy(new FirstCollateral());
                break;
            case WantedCollateralSelect.FirstCollateralIdCollateral:
                context.SetStrategy(new FirstCollateralIdCollateral());
                break;
            default:
                context.SetStrategy(new BiggestCollateral());
                break;
        }

        return context.ExceceuteStrategy(collaterals);
    }
}
```

KUVA 10. Apuluokka asiakaskohtaisten asetusten saantia varten

Luodun apuluokan avulla pystyttiin erottamaan selkeästi globaalit sekä kerran käytetyt metodit omiin tiedostoihinsa, näin lisäten ylläpidettävyyttä sekä uudelleenkäytettävyyttä.

Globaalin GetCollateral metodin kutsua hyödynnettiin tiedostoissa, joissa kerättiin datat Excel-tiedostoja varten. Näitä tiedostoja oli yhteensä kaksi, kummallekin Excel-tiedostolle omansa. Kullakin tiedostolla oli omat Populate metodinsa, jotka keräsivät ensin kaikki luotot, ja tämän jälkeen luotoille valittiin haluttu vakuus hyödyntämällä globaalia CustomerConfiguration luokkaa. (kuva 11) Tämän jälkeen kerättiin kaikki halutut data oikeanlaisina Exceleitä varten, ja ne tallennettiin parametreihin, jotka sitten ExcelServicessä otettiin mukaan oikeille paikoilleen Excel-tiedostoissa.

```
public void Populate()
{
    var loansWithCollaterals = _loans.GroupJoin(_collaterals.AsParallel(),
    loan => loan.LoanId,
    collateral => collateral.LoanId,
    (loan, collateral) => {
        if (!collateral.Any())
        {
            return new LoanWithCollateralAndKayttoarvo { LoanData = loan };
        }

        var wantedCollateral =
            CustomerConfigurationsUsedInMultiplePlacesHelper
                .GetCollateral(_configuration, collateral.ToList(), _wantedCollateralContext);

        return new LoanWithCollateralAndKayttoarvo
        {
            LoanData = loan,
            CollateralData = wantedCollateral,
            KaypaArvo = CustomerConfigurationsSpecificHelper
                .GetKaypaArvo(_configuration, wantedCollateral, _showKaypaArvoInRowContext)
        };

    }).ToList();

    _parameters.LoanWithCollateralsOne = loansWithCollaterals;

}
```

KUVA 11. Property luokka, jota Report Service luokka käyttää Exce-tiedostojen täyttämiseen datojen avulla

4.3.1 Tehdyn ratkaisun uudelleenkäytettävyys ja skaalattavuus

Opinnäytetyössä haluttiin todentaa, että Design Patternien avulla luotua koodia pystyi ylläpitämään, rakentamaan ja toteuttamaan yksinkertaisesti ja helposti (Lasater, 2007, Why Pattern?).

Ylläpidettävyyttä ja hallittavuutta pyrittiin testaamaan lisäämällä kaksi uutta konkreettista strategiaa, FirstCollateral sekä FirstCollateralIdCollateral, jotka vastasivat malliesimerkin ensimmäistä vakuutta sekä pienintä vakuusnumero vakuutta. Tämän jälkeen päivitettiin kaikki niihin tarvittavat tiedostot toimimaan uusien strategioiden kanssa.

Uusia strategioita luodessa todettiin, että valittua strategiamallitapaa oli helppo ylläpitää, koska kaikki tiedostot olivat selkeästi nimetty, ja kaikki tarvittavat tiedot, joita piti päivittää, olivat selkeästi erillään muusta sovelluskoodista. Erillään olevan koodin ansiosta sovellusta päivittäessä ei ollut pelkoa, että jokin paikka, jonka pitäisi hyödyntää tarvittavaa strategiaa ei päivittyisikään uusien tarpeiden mukaan. Tämän avulla myös todettiin, että strategioita oli helppo skaalata, kun kaikki

tarvittavat tiedot löytyvät selkeästi jaoteltuina erillään, jolloin niitä oli helppo kopioida uusille paikoilleen samoihin tiedostoihin rikkomatta ja muokkaamatta olemassa olevia koodeja tarpeettomasti. (kuva 10 ja 12).

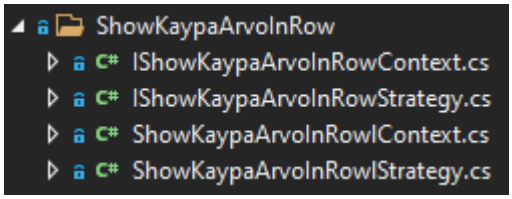
```
2 references
public class BiggestCollateral : IWantedCollateralStrategy
{
    2 references
    public Collateral GetWantedCollateral(List<Collateral> collateralList)
    {
        return collateralList.OrderByDescending(collateral => collateral.KaypaArvo).FirstOrDefault();
    }
}

1 reference
public class FirstCollateral : IWantedCollateralStrategy
{
    2 references
    public Collateral GetWantedCollateral(List<Collateral> collateralList)
    {
        return collateralList.FirstOrDefault();
    }
}

1 reference
public class FirstCollateralIdCollateral : IWantedCollateralStrategy
{
    2 references
    public Collateral GetWantedCollateral(List<Collateral> collateralList)
    {
        return collateralList.OrderBy(collateral => collateral.CollateralId).FirstOrDefault();
    }
}
```

KUVA 12. Kaikki halutut konkreettiset strategiat

Tämän jälkeen haluttiin testata strategian yksinkertaisuutta ja uudelleenkäytettävyyttä kokonaan uuden strategian luonnin avulla. Tätä pyrittiin testaamaan luomalla toinen perusstrategia, jossa joko näytettiin tai piilotettiin rivin lopusta vakuuden käypäarvotieto (kuva 11 ja 13).



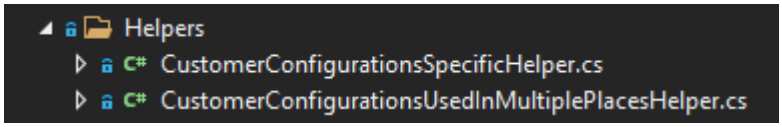
KUVA 13. Strategia käyttöarvon näyttämistä varten

Jotta strategia saatiin toimimaan, vaati se tietokantataulun päivittämistä ja siihen uuden sarakkeen lisäämistä. Uusi sarake kertoi, että näytetäänkö vai piilotetaanko haluttu käypäarvotieto numeron avulla. Uutta saraketta tietokantatauluun lisätessä huomattiin se, että taulut voivat kasvaa nopeasti suuriksi ja epäselviksi,

jos rivejä ja taulujen sarakkeita ei nimetä ja suunnitella alusta asti selkeästi. Jotta tietokantatauluista saa parhaimmat puolet esiin, niin skaalattavuuden kuin uudelleenkäytettävyydenkin osalta, tulisi siis tuotetyön alussa suunnitella selkeästi luottoluokittajaraportoinnin tietokantakokonaisuus.

Uuden strategian lisääminen tapahtui erittäin nopeasti kopioimalla jo luotu strategia pohjaksi ja sitä muokkaamalla. Uudelle strategialle luotiin uusi apuluokka, johon kerättiin kaikki strategiat, joita käytettiin vain kyseisessä Excel-tiedostossa. Tällä jaolla pyrittiin demonstroimaan ero useasti sekä kerran käytettävien strategioiden välillä (kuva 14).

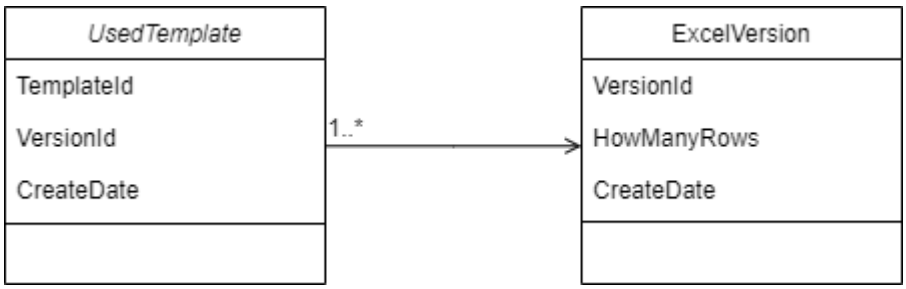
Apuluokat ovat kuitenkin asia, jota tuotteen kohdalla kannattaa tutkia tarkasti etukäteen, sillä strategioiden päivittäminen ja siirtäminen voi olla vaikea jälkikäteen. Mikäli tämä jako tehdään huolellisesti, ovat apuluokat erittäin hyviä ratkaisutapoja ylläpidettävyyttä sekä skaalattavuutta ajatellen.



KUVA 14. Kaikki halutut konkreettiset strategiat

4.4 Eri Excel-pohjien mahdollistaminen raportille

Opinnäytetyöprojektissa haluttiin rakentaa mahdollisuus hakea tiedot Excel-tiedostoihin eri pohja-asetusten perusteella. Malliprojektissa käytettiin esimerkkinä kahden Excel-pohjan maksimirivimäärän vaihtumista. Tätä varten rakennettiin kaksi eri tietokantataulua. Ensimmäisen taulun ideana oli sisällyttää kaikki Excel-pohjakohtaiset versioerot omissa riveissään. Tämän rivijakauman ideana oli, että rivejä ei päivitettäisi muuten, kuin uusien sarakkeiden erojen ilmestyessä seuraavien pohjien myötä. Toisen taulun idea oli ylläpitää kullakin hetkellä asiakkaan valitsemaa Excel-pohjaa (kuva 15).



KUVA 15. Excel-pohjan ja version suhde UML kaaviolla

Tämä tietokantataulujakauma tarkoitti käytännössä sitä, että jos Excel-pohjia olisi kaksi erilaista, niin ExcelVersion taulussa olisi kaksi datariviä, ja UsedTemplate taulussa kerrottaisiin kumpaa datariviä milläkin hetkellä käytettäisiin.

Tässä opinnäytetyössä ensimmäisessä ExcelVersion tietokantataulun rivissä ylläpidettiin tietoa siitä, että Excel-pohjaan mahtuisi 2 riviä dataa. Taulun toisessa rivissä kerrottiin, että toiseen Excel-pohjaversioon mahtuisikin 3 riviä dataa. Näitä tietoja pystyttiin hyödyntämään ottamalla UsedTemplate tietokantataulusta uusin rivi CreateDaten tiedon perusteella, joka sisälsi VersioId:n perusteella saadun tiedon, että mikä Excel-versiopohja oli käytössä milläkin hetkellä (kuva 15).

Jotta itse data pysyi oikeanalaisena, ja helposti hallittavana loppuun asti, ei rivien määrän rajaus saanut vaikuttaa itse datan määrän saantiin, vaan Excel-tiedostojen rivien määrää tuli karsia tarvittaessa vasta tiedostoa muodostaessa.

Tällä vaatimuksella päädyttiin hyödyntämään strategiamallia ExcelServicessä, jossa sallittu datamäärä vietiin Excel-tiedostoon. Malliprojektissa päädyttiin siihen, että Excel-tiedosto numero yhdellä oli kaksi eri pohjavaihtoehtoa, joissa tuli eri määrä tietoa asetuksien perusteella. Kun halutut datat oli yhdistetty ja ne oli annettu parametreinä ExcelServiceen, tutkittiin servicessä, että mikä Excel-pohja oli käytössä hakemalla viimeisin tieto tietokannasta (kuva 16). Tämän avulla turvattiin, että haluttu versio oli aina viimeisin versio, joka haluttiin olevan käytössä.

```
var template = await _customerConfigurationTemplateStore.GetLatestCustomerConfigurationTemplate();
```

KUVA 16. Uusimman tiedon saanti Excel-pohjasta

Tämän jälkeen saatua pohjaversiota hyödynnettiin Excel rivien määrän saantiin strategiamallia käyttäen (kuva 17).

```
var collateralLoans = CustomerConfigurationVersionsHelper
            .GetRowsInExcel(_wantedRowsInExcelContext, template, parameters.LoanWithCollateralsOne);
```

KUVA 17. Excel versio apuluokka strategiamallin apuna.

Strategiamallin periaate oli sama, kuin halutun vakuuden saannissa. GetRowsInExcel oli staattinen metodi, jolle asetettiin uusi strategia versioId:n perusteella, jonka jälkeen strategia suoritettiin. (kuva 18)

```
1 reference
public static IEnumerable<LoanWithCollateralAndKayttoarvo> GetRowsInExcel(IWantedRowsInExcelContext context,
    CustomerConfigurationTemplate template,
    IEnumerable<LoanWithCollateralAndKayttoarvo> loansWithCollaterals)
{
    switch (template.CustomerConfigurationVersionId)
    {
        case 1:
            context.SetStrategy(new ShowTwoRows());
            break;
        case 2:
            context.SetStrategy(new ShowThreeRows());
            break;
        default:
            context.SetStrategy(new ShowTwoRows());
            break;
    }

    return context.ExceceuteStrategy(loansWithCollaterals);
}
```

KUVA 18. Excel rivien määrän strategian vaihto ja tiedon saanti

Strategia hyödynsi kahta konkreettista strategiaa, jotka palauttivat saadun tiedon perusteella joko kaksi tai kolme riviä dataa, jotka se sitten tallensi Excel-tiedostoon (kuva 19).

```
2 references
public class ShowTwoRows : IWantedRowsInExcelStrategy
{
    2 references
    public IEnumerable<LoanWithCollateralAndKayttoarvo> GetWantedRowsInExcel(
        IEnumerable<LoanWithCollateralAndKayttoarvo> loansWithCollaterals)
    {
        return loansWithCollaterals.Take(2);
    }
}

1 reference
public class ShowThreeRows : IWantedRowsInExcelStrategy
{
    2 references
    public IEnumerable<LoanWithCollateralAndKayttoarvo> GetWantedRowsInExcel(
        IEnumerable<LoanWithCollateralAndKayttoarvo> loansWithCollaterals)
    {
        return loansWithCollaterals.Take(3);
    }
}
```

KUVA 19. Konkreettiset strategiat Excel rivien määrän saantia varten

4.4.1 Tehdyn ratkaisun uudelleenkäytettävyys ja skaalattavuus

Excel-pohjaratkaisun uudelleenkäytettävyys oli samalla tasolla halutun vakuuden kanssa, joka on hyvä. Kaksi erillistä tietokantataulua tuo joustavuutta, sekä selkeyttä tiedon hallintaan ja tallentamiseen. Rivien lopullisen määrän päättely vasta juuri ennen Excel-tiedoston tekemistä luo varmuutta datan oikeellisuuden pitämiseen loppuun asti. Selkeiden Excel-pohjaerojen erikseen siirtäminen voi myös auttaa pidemmän päälle koko luottoluokittajaraportoinnin ylläpidettävyyttä sekä uudelleenkäytettävyyttä selkeyttäen erikseen pohjien ja perus asiakaskohtaisten erojen tarpeet. Tässä on kuitenkin hyvä ottaa huomioon se, että eri Excel-pohjilla voi olla paljonkin pieniä tai suurempia eroja. Tämä lisää nopeasti koodimäärää ja siksi se pitää suunnitella alusta asti selkeäksi sekä tietokantarakenteessa että itse koodissa tehdyissä pohjien välisissä erotuksissa.

5 POHDINTA

Opinnäytetyössä pyrittiin etsimään ratkaisuja luottoluokittajaraportoinnin asiakaskohtaiseen räätälöintiin ja sen keskeisiin ongelmiin. Raportoinnilla oli kolme keskeistä ongelmaa. Ensimmäisenä ongelmana, jota lähdettiin selvittämään, oli missä asiakaskohtaisia asetuksia ja säännöksiä pidettiin tallessa, ja kuinka ne saatiin sovelluksen käyttöön. Toisena ongelmana oli, miten asiakaskohtainen räätälöinti toteutettaisiin helposti skaalattavaksi ja hallittavaksi. Kolmas ongelma, jota pyrittiin ratkaisemaan, oli eri raportointipohjien versioerot, ja kuinka niiden tarve voitaisiin parhaiten ottaa huomioon sovelluksessa. Kaikkiin ongelmiin pyrittiin löytämään ratkaisutavat, jotka olisivat mahdollisimman uudelleen käytettäviä sekä skaalattavia tarpeen mukaan. Näiden ongelmien ratkaisemiseen tehtiin taustatyötä, ja taustatyön perusteella rakennettiin malliprojekti, jossa esiteltiin valitut ratkaisutavat. Luotu malliprojekti on suuntaa antava, mutta sen perusteella toimeksiantajayritys pystyy ottamaan viitteitä tarvittaessa tuoteprojektin luottoluokittajaraportoinnin asiakaskohtaiseen räätälöintiin, kuinka sitä kannattaisi lähestyä, ja mitä kaikkea projektin suhteen tulisi ottaa huomioon.

Opinnäytetyössä luotiin toimiva malliprojekti käyttäen ASP.NET Frameworkia sekä C#-ohjelmointikieltä. Opinnäytetyön perusongelmia lähdettiin ratkaisemaan strategiamallia sekä tietokantatauluja käyttämällä. Luotuja ratkaisutapoja testattiin skaalautuvuuden sekä ylläpidon ja hallinnan kautta, ja testauksessa todettiin, että molemmat valitut ratkaisutavat tarvitsevat paljon taustatyötä, suunnittelua sekä ennakointia, jotta niistä saa kaiken mahdollisen irti.

Mikäli toimeksiantajayrityksen projektissa kuitenkin suunnitellaan näiden suositusratkaisujen käyttö tarkasti, ovat valitut ratkaisutavat helposti skaalattavissa sekä ylläpidettävissä tarpeen mukaan. Sekä tietokantataulut että suunnittelumallit voivat parhaimmillaan antaa toimeksiantajan tuotteelle suuren edun uusien asiakkaiden luottoluokittajaraporttitarpeiden kartoittamisessa, sekä mahdollisuuden helpottaa uusien asiakkaiden kohdalla tuotteen käyttöönottoa luottoluokittajaraporttien osalta. Tämän ratkaisun avulla toimeksiantajayritys pystyisi parhaimmillaan säästämään jopa satoja henkilötyötunteja uusien asiakkaiden kohdalla, ja

täten rahaa säästyisi asiakaskohtaisten räätälöintien hallinnassa sekä ylläpidossa. Tämä taas voisi parhaimmillaan tuoda lisäarvoa tuotteen myynnissä, sekä ylläpidossa asiakkaiden kanssa, kun muutosten teko olisi skaalattavan ja uudelleen käytettävän koodin avulla helppoa hallita ja päivittää asiakkaiden muutostarpeiden mukaan.

LÄHTEET

Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. 37. painos. Indianapolis: Addison-Wesley.

Larkin, K., Smith, S., Addie, S., Dahler, B. Microsoft. 2021. Viitattu 20.10.2021. https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0

Lasater, C. G. 2007. Design patterns. Plano, Texas: Wordware Pub.

Microsoft. n.d. ASP.NET A framework for building web apps and services with .NET and C#. Viitattu 6.11.2021. https://dotnet.microsoft.com/apps/aspnet

Microsoft. n.d. What is .NET?. Viitattu 6.11.2021. https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet

MSDN Magazine, 7/2005. Discover the Design Patterns You're Already Using in the .NET Framework. Viitattu 21.10.2021. https://docs.microsoft.com/en-us/archive/msdn-magazine/2005/july/discovering-the-design-patterns-you-re-already-using-in-net

Smith, S. 10.03.2021. Overview of ASP.NET Core MVC. Microsoft. Viitattu 7.11.2021. https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-6.0

Yaish, H., Goyal, M. 2013. Multi-tenant Database Access Control. IEEE 16th International Conference on Computational Science and Engineering. https://doi.org/10.1109/CSE.2013.131