Jukka Rantanen

Sidosryhmäkohtaisten funktionaalisuuksien generalisoimoinen - toteutus ja evaluointi

Tietotekniikan pro gradu -tutkielma

15. elokuuta 2018

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Jukka Rantanen

Yhteystiedot: jutarant@student.jyu.fi

Ohjaaja: Ville Isomöttönen

Työn nimi: Sidosryhmäkohtaisten funktionaalisuuksien generalisoimoinen - toteutus ja evaluointi

Title in English: Generalization of client specific functionality - implementation and evaluation

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Sivumäärä: 57+29

Tiivistelmä: Tämä tutkielma tarkastelee yhden spesifin liiketoimintadataa jakavan sovelluksen yleistämistä toimimaan useamman liiketoimintakumppanin tietojärjestelmien kanssa, yleistämiseen käytettyjä ohjelmointitekniikoita ja suunnittelutapoja, sekä yleistämisellä saatuja työaikasäästöjä.

Tutkielma pohjautuu anonymisoituun koodiin, sekä anonymisoituun aikatauluraporttiin, sillä taho jonka tarpeisiin sovellus kehitettiin haluaa pysyä nimettömänä.

Tutkielma vertailee ohjelman koodia kahdessa vaiheessa; ennen ja jälkeen refaktoroinnin. Vertailun tarkoituksena on etsiä yleisesti tunnistettuja ohjelmointitekniikoita ja -tapoja, sekä pohtia näitten vaikutusta koodin yleiseen laatuun.

Koska tarkastelun kohteena on vain yksi ohjelmisto, jonka muokkaukseen tutkija osallistui, ohjelmiston muutoksia, muutosten syitä, sekä niiden vaikutuksia analysoidaan konstruktiivisesti.

Avainsanat: Refaktorointi, Evaluointi, Koodin yleistäminen, Koodin uudelleenkäyttö, Data Transformation, Process Automation, Data Acquisition, Data Warehouse

Abstract: This thesis looks at the generalization of a specific enterprise application provi-

ding data acquisition automation, the utilized programming techniques or design patterns, and the potentially reduced amount of work resulting from the generalization.

The thesis is based on anonymized code and an anonymized time table report, because the party, for whose needs the application was developed, wishes to remain anonymous.

The thesis compares the code of a single program that was modified by the researcher. The comparison is performed between the program code before the refactoring started, and after it was completed. No in-between steps are considered. The purpose of the comparison is to locate general design patterns and techniques utilized in the refactoring. These findings are the basis of comteplation on the effects they have on the general code quality of the software.

As the thesis is about investigating a single program, with the analyst being one of the people implementing the changes, the analysis of the changes and their effects is done with a viewpoint of constructive research.

Keywords: Refactoring, Evaluation, Code Generalization, Code Re-Use, Data Transformation, Process Automation, Data Acquisition, Data Warehouse

Esipuhe

Jotenkin Paavo sai vakuutettua minulle, että “työelämästäkin saa graduja aikaan”, ja Ville hieroi konstruktiivisen tutkimuksen konseptia naamaani kunnes ymmärsin miten “ihan normipäivä töissä”-tyyppinen tilanne voisi olla tieteellisesti kiinnostavaa tutkittavaa. Hommassa olen oudosti löytänyt useita perusteita asioille, jotka olen ennen tätä sisäistänyt kohtuullisen nimettömästi. Tässä tutkielmassa olen kuitenkin ollut pakotettu etsimään niille nimet ja miettimään niiden konsepteja tarkemmin. Koska gradujen tieteellinen arvo on usein varsin nimellinen, näenkin että tämä tuotos on auttanut enemmän minua vahvistamaan ja selkeyttämään omaa kykyäni kommunikoida työstäni toimialan kielellä.

Lisäksi, tätä ei olisi kirjoitettu ilman Teroa, Anttia, Hokkia, Leilaa, Annaa, Mikkoa, Tuomasta, Alexandraa, Tonia, Sannaa, tai Liehua. Kiitos.

-Rantanen

Termiluettelo

Abstrakti luokka	Luokka, joka määrittelee mahdollisesti vajavaisen luokan ja voi vaatia, että siitä perittävät luokat toteuttavat tietyt abstraktiksi esitellyt metodikuvaukset. Naiivisti määriteltynä luokka, jossa osa metodeista voidaan määritellä ilman toteutusta, samalla tapaa kuin rajapinnoissa.
Asiakas	Taho jonka tarpeisiin tutkielman tarkastelukohteena oleva sovellus alunperin kehitettiin.
Data Acquisition	Suomeksi “Tiedonkeruu”. Prosessi, jossa luetaan dataa yhdestä lähteestä, muunnetaan se spesifin kohteen ymmärtämään muotoon ja lopuksi tuodaan se kohdejärjestelmään. Suomenkielinen vastine on helposti sekoitettavissa yleiseen käsitteeseen tiedon keräämisestä, joten tutkielma käyttää englanninkielistä termiä.
Data Warehouse	Yleinen tallennuskohde, joka integroi tietoa useammasta lähteestä. Suomenkielinen vastine “tietovarasto” on helposti sekoitettavissa ohjelmistoteknisiin abstraktioihin esim. tietokannoista, joten tutkielma käyttää englanninkielistä termiä. Data Acquisition-menetelmillä populoidaan Data Warehouse-järjestelmiä.
GoF	Gang of Four, “Design Patterns: Elements of Reusable Object-Oriented Software”-kirjan kirjoittajat, eli Erich Gamma, Richard Helm, Ralph Johnson ja John Vlissides. (ks. viitteet)
Kapselointi	Ohjelman komponenttien, niiden osien, tai niiden toiminnallisuuden näkyvyyden rajoittaminen. ks. Metodin näkyvyys ja Muuttujan näkyvyys.
Käyttäjä	Henkilö. Käyttää Asiakkaan järjestelmiä ja luo tietueita.
Kolmas osapuoli	Taho jolle Asiakas välittää Käyttäjän luomia tietueita. Esimerkiksi liiketoimintakumppani.
Luokka	Ohjelman osa-alue. Javassa luokka voi määritellä olioita ja metodeja.

Metodi	(Javassa) luokan tai olion osa. Metodi sisältää itsenäisen osan ohjelmakoodia. Metodeja käytetään koodin rakenteelliseen järjestelyyn ja toistuvan koodin eliminoimiseen.
Metodin tunniste	Englanniksi “Method Signature”. Metodin tunniste on uniikki kooste, johon sisältyy metodin nimi, sen parametrien tyypit, sekä sen paluuarvon tyyppi. Javassa määrite on suppeampi, sillä metodin paluuarvon tyyppi ei ole osa metodin tunnistetta.
Metodin näkyvyys	Rajoittaa mistä kaikkialta metodia voidaan kutsua. Tämän tutkielman puitteissa ei käytetä kuin kahta määrittelyä. Julkisia metodeja voi kutsua mistä tahansa. Yksityisiä metodeja voi kutsua vain luokan sisältä.
Muuttujan näkyvyys	Kuten metodin näkyvyys, mutta olion muuttujille. Javassa on yleisenä käytäntönä määritellä kaikki olion muuttujan yksityisiksi ja hallita näkyvyyttä luku- ja kirjoitusmetodeilla.
Olio	Luokassa määritelty kokonaisuus. Luokka voi sisältää metodeja ja tilaa kuvaavia muuttujia.
Olioinstanssi	Olion ajonaikainen versio. Luokasta voidaan luoda useampi instanssi joilla voi toteutuksesta riippuen olla oma tila. Esimerkiksi Javassa kaikki tekstit ovat luokan “String” instansseja, mutta niiden sisältö vaihtelee.
Rajapinta	Määrittely joka kertoo mitä metodeja rajapintaa toteuttavan luokan pitää sisältää. Olioita jotka toteuttavat rajapinnan, voidaan käyttää sekä itse oliona, että sen toteuttamien rajapintojen mukaisesti.
Rajapintatoteutus	Koodi, joka toteuttaa tietyn rajapinnan. Yksi luokka voi toteuttaa useampia rajapintoja. Javassa rajapintatoteutusten ei tarvitse noudattaa kuin metodien parametrejä, paluuarvoa ja poikkeuksia. Itse toteutuksen sisältöä rajapinta ei voi määritellä.

Kuviot

Sisältö

1 Johdanto

Tieto ja sen analysointi on osoittautunut arvokkaaksi, jopa siinä määrin, että jättimäiset yritykset kuten Google käyttävät ansaintamallinsa raaka-aineena käyttäjiltään kerättyä tietoa. Tietokoneiden tehon ja tallennuskapasiteetin huomattavan kasvun myötä tiedon tallentaminen ei enää ole niin suuri ongelma, kuin sen kerääminen. Tiedosta onkin tullut kauppatavaraa.

Valitettavasti tiedon tallentamiseen ei ole universaaleja sääntöjä. Siinä missä yksi järjestelmä tallentaa henkilön etunimet ja sukunimen eri tietueisiin, voi toinen järjestelmä tallentaa koko nimen yhteen tietueeseen. Syyt tämänlaiseen toimintaan kumpuavat monista tekijöistä, mutta pelkästään nimitietoa säilövän tietueen tunnisteena voitaisiin helposti kuvitella olevan “nimi”, “nimet”, “koko_nimi”, “names”, “last name”, “apellido”, “family_name”, “givenName” tai “familyname”.

Edellämainittu ongelma ratkaistaan yleensä totettamalla tai ostamalla Data Acquisition-prosessin toteuttava järjestelmä. Tällaisten järjestelmien toiminta on kaksitahoista. Ne määrittelevät jonkinlaisen mekanismin siirtää tietoa järjestelmästä toiseen, sekä jonkinlaisen funktion, jolla järjestelmän A tieto saadaan muotoon jota järjestelmä B ymmärtää.

Kyseisen prosessin automatisointi on ollut aiheellista sen verran pitkään, että Microsoft haki ongelman ratkaisevan yleiskäyttöisen työkalun patenttia alunperin jo 1998 (MacLeod ja Kieman 2002). Microsoftin DTS (Lawton ja Awalt 1999) hoitikin useasta lähteestä lukemisen, tiedon muuntamisen järjestelmä A:n ymmärtämästä muodosta järjestelmä B:n ymmärtämään muotoon, sekä jonkinlaisen prosessin, jolla voitiin tehdä ehdollista suoritusta tiedon siirrossa.

Tutkielmassa tarkastellaan miten kustomoitua Data Acquisition-palvelua tarjoava järjestelmä yleistettiin toimimaan useamman kuin yhden kolmannen osapuolen järjestelmän kanssa. Yleistys toteutettiin yleistämällä kohdejärjestelmäspesifeistä osioista rajapinnat ja muokkaamalla kaikille yhteinen toiminnallisuus käyttämään kyseisiä rajapintoja, spesifien toteutusten sijaan.

Yleistyksen päämääränä oli hyödyntää mahdollisimman paljon olemassaolevaa ja testattua ohjelmakoodia, jotta uusien kohdejärjestelmien lisääminen automaattisen tiedonsiirron piiriin ei vaatisi niin paljon resursseja, kuin ensimmäisen implementaation toteutus vaati.

Tutkielman käsitellessä tasan yhtä ohjelmistoa, jonka muutoksia olin toteuttamassa, on hyödyllisintä analysoida refaktorointia kahtena artefaktina; ohjelmiston lähdekoodi ennen refaktorointia, sekä lähdekoodi refaktoroinnin jälkeen. Tämä lähestymistapa myöskin edesauttaa muutosten analysointia konstruktiivisesti, sillä vertailu koitetaan suorittaa kahden selkeän artefaktin välillä. Vaikka objektiivisuutta ei tällä tavalla täysin saavutetakaan, on mielestäni kokonaisuuksien vertailu järkevämpää kuin keskeneräisten muutosten.

Tutkielman ensimmäisessä osassa käydään läpi tarkkoja teknisiä sekä arkkitehtuurisia käsitteitä, joita käytetään analysoitavan sovelluksen yleisluontoisessa kuvauksessa. Yleiskuvauksen jälkeen tutkielman toisessa osassa tarkastellaan sovelluksen alkuperäiseen koodiin pohjautuvaa anonymisoitua versiota luokka- ja koodilistaustasolla. Toisen osan tarkoituksena on havainnollistaa kuinka ohjelmistoa muutettiin, sekä esittää konkreettisia esimerkkejä erilaisten ohjelmistotekniikan metodien sopivuudesta ohjelmistojen yleistämiseen ja yleistettäväksi suunniteluun. Kolmannessa osassa arvioidaan sovelluksen refaktoroinnista saatuja koodihyötyjä, sekä tutkitaan sovelluksen omistajan raportin pohjalta sovelluksen refaktoroinnin kannattavuutta resurssimielessä.

2 Suunnittelumallit ja ohjelmointiperiaatteet

Luvussa esitellyt konseptit ovat vahvasti esillä joko käsiteltävän ohjelman toiminnassa, niitä hyödynnetään refaktoroinnissa, tai niiden avulla voidaan kuvata potentiaalisia parannuksia ohjelmaan. Näitten syiden takia niiden periaatteet käydään läpi.

Kehittämäni esimerkit eivät välttämättä ole täydellisiä läpikäyntejä niiden teemoista, mutta ne mielestäni kuvaavat idean tarpeeksi selvästi, että suunnittelumallien ja ohjelmointiperiaatteiden käyttötarkoitus käy ilmi ainakin tutkielman tarvitsemissa rajoissa.

2.1 Yleiset ohjelmistotekniikan käsitteet

2.1.1 Tietomalli

Tietomallit määrittelevät, miten tietoa käsitellään ja siihen viitataan ohjelmakoodissa (West 2010). Yksinkertaisena esimerkkinä voi toimia vaikka kordinaatiston piste, joka koostuu X- ja Y-akseleita kuvaavista arvoista.

Koska vain ohjelmointikielen tekniset ominaisuudet rajoittavat miltä tietomalli voi näyttää, tiedon siirtäminen tietomallista toiseen ei ole triviaali ongelma. “Nimi” ja “Name” voivat tarkoittaa samaa asiaa, mutta koska tietueet eivät ole saman nimisiä, niiden ristikkäinen käyttö ei onnistu suoraan. Lisäksi tilannetta hankaloittaa myös se, että vaikka tietueet hyväksyisivät samanmuotoista tietoa ja olisivat saman nimisiä, niiden kontekstit eivät välttämättä ole yhteensopivia. Esimerkiksi ihmisillä ja laivoilla voi olla tietue “Nimi”, mutta näitä ei välttämättä ole haluttavaa käsitellä samalla tavalla.

2.1.2 Data Acquisition-sovellus

Koska tietomallit voivat olla erilaisia, yhden tietomallin käyttö useammassa palvelussa pitää hoitaa jaetulla tietomallin määrittelyllä. Koska tämä kuitenkin vaatisi, että tietomallin määrittely olisi valmis ennen tilannetta jossa ohjelmisto ottaa tietomallin käyttöönsä, on tällainen vaatimus usein ongelmallinen. Asiasta tulee hankalampaa, kun tarve yhteiselle tietomallille havaitaan vasta kun ohjelmisto on toteutettu.

Näistä syistä johtuen, onkin usein helpompaa miettiä tapa jolla yhden tietomallin sisältö voidaan muuntaa toisen tietomallin mukaiseen muotoon. Ongelma ei itsessään ole mitenkään uusi, sillä jopa tiedon kopiointi puhelinluettelosta osoitekortistoon voidaan mieltää tällaisena operaationa. Tietotekniikka on lähinnä mahdollistanut prosessin automaation ja internetin myötä luonut valmiudet suorittaa tiedonvälitystä ympäri maailmaa.

Data Aqcuisition-sovellukset yksinkertaisimmillaan lukevat tietoa yhdessä muodossa ja tuottavat sitä toisessa muodossa. Tapa millä kommunikaatio hoidetaan lähde- ja kohdejärjestelmien välillä on täysin toteuttajien päätettävissä.

Tällaiset sovellukset toimivat erittäin yksinkertaisen prosessin mukaan:

1. Lue dataa lähteestä
2. Muuta lähteen data toiseen muotoon
3. Kirjoita muutettu data kohteeseen

Prosessi voi olla kokonaisuudessaan automatisoitu, tai se voi vaatia manuaalisia operaatioita. Jos prosessi tuottaa tera- tai petatavumäärissä dataa, voi kohde olla nippu kovalevyjä, joka täytyttyään pitää fyysisesti postittaa jonnekin. Jos taas muunnos ei ole selkeästi automatisoitavissa, voidaan prosessi integroida järjestelmään, jossa ihmiset saavat tutkiakseen dataa ja päättävät sen muutoksesta. Päätöksen jälkeen prosessi jatkuu automaattisesti.

Prosessi ei myöskään aseta vaatimuksia lähteiden tyypille tai prosessin ohjaajalle. Kaikki seuraavat ovat täysin valideja esimerkkejä kyseisenlaisille automatisoiduille järjestelmille:

- Sovellus kysyy verkko-osoitteesta seuraavan prosessoitavan tietueen, lukee ja konvertoi sen, sen jälkeen muunnettu tieto kirjoitetaan tietokantaan. Tämän jälkeen sovellus odottaa viisi minuuttia ja alkaa alusta.
- Manuaalisesti käynnistettävä sovellus lukee dataa tietokannasta rivi kerrallaan, konvertoi ne tiettyyn muotoon ja lähettää jokaisen sähköpostilla eteenpäin. Kun koko tietokanta on käyty läpi, sovellus sammuu, eikä sitä enää tarvita.
- Sovellus kuuntelee ulkoista signaalia, jonka mukaisen tietueen se lataa, konvertoi ja lähettää jollekin toiselle taholle. Tämän jälkeen se jää odottamaan uutta signaalia.

Viimeinen esimerkeistä on yleinen kuvaus periaatteesta jolle tutkielmassa käsiteltävä sovel-

lus perustuu.

2.1.3 Synkroninen ja asynkroninen kommunikaatio

Prosessien välisen kommunikaation korkeimman tason kaksi paradigmaa ovat synkronisuus ja asynkronisuus (Andrews ja Schneider 1983). Synkronisuuden perustoiminnallisuus on, että pyynnön tai viestin lähettäjä jää odottamaan vastausta, eikä tee muuta kunnes sellaisen saa. Asynkronisuudessa taas lähettäjä antaa viestin jonkin toisen tahon (kuten säikeen tai ohjelman) vastuulle ja jatkaa muita mahdollisia operaatioita suoraan.

Synkronisuuden etuja ovat selkeä toimintamalli, joka mahdollistaa sekä vastauksen, että viestin jatkokäsittelyn helposti samassa kontekstissa. Lisäksi operaatioden sekvenssi on aina sama, joka lisää toiminnan ymmärrettävyyttä. Haittapuoleksi taas voidaan lukea odottamisen varaamat järjestelmäresurssit. Asynkronisuus taas mahdollistaa pyyntöjen ja viestien vastausten käsittelyn huomattavasti pienemmillä resursseilla, mutta monimutkaisemmalla mekaniikalla. Esimerkkeinä tästä voi olla semaforien pollaus, jaettujen muistialueiden lukeminen, tai tapahtumakuuntelijat. Lisäksi on otettava huomioon, että alkuperäisen viestin lähettäjäkonteksti täytyy joko sisällyttää paluuviestiin, tai antaa siihen jonkinlainen latausviite, mikäli operaatioita halutaan jatkaa lähetyshetkeä vastaavalla tilalla. Molemmat vaatimukset kasvattavat ohjelmistojen kompleksisuutta.

Asiasta tulee monimutkaisempaa, kun kommunikaatio tapahtuu erillisten tietokoneiden välillä, jolloin jaettu muisti ei ole mahdollisuus, vaan kommunikaatio täytyy toteuttaa verkkoyhteyden tai vastaavan teknisen mekanismin yli.

Asynkroniseen viestinvälitykseen on Javassa mahdollista käyttää JMS-viestinvälitystä joka tarjoaa sekä jonokäyttäytymisen (queue), että otsikkokäyttäytymisen (topic). Ensimmäisessä viestit kerätään jonoon ja otetaan sieltä käsiteltäväksi. Jälkimmäisessä viesti välitetään juuri viestin käsittelyhetkellä otsikkoon rekisteröityneenä oleville tahoille. (“How does a Queue compare to a Topic” 2011)

Tutkimuksen kohteena oleva ohjelmisto käytti viestijonoja toteuttaakseen asynkronista viestinvälitystä useiden tietokoneiden välillä.

2.1.4 Viestijonot

Viestijonot ovat yksi tapa hoitaa asynkronista viestinvälitystä. Viestijonon periaate on, että yhteen "säiliöön" voidaan lähettää viestejä, josta ne otetaan käsittelyyn jossain vaiheessa. Viestijonoon lähettävien ja sieltä käsittelyyn ottavien tahojen ei ole tarpeellista tietää toisistaan mitään, eikä niiden välillä tarvitse olla viestin muodon lisäksi mitään sopimuksia. Jonoa voi kuunnella useampi taho, jotka voivat pyytää seuraavaa viestiä käsittelyyn. Koska useampi taho voi käsitellä viestejä ja tahot voivat käsitellä viestejä eri vauhdilla tai tavoin, ei käsittelyjärjestyksestä voida antaa takuita.

Viestijonoja voidaan käyttää lyhentämään sovelluksen sisäistä logiikkaa, tai pilkkomalla peräkkäisten operaatioiden ketjutus erillisiin kokonaisuuksiin. Viestien uudelleenlähetys ja seuranta voidaan siirtää viestejä tuottavasta sovelluksesta viestijonosovelluksen vastuulle. Suorista kutsuista viestinvälitykseen siirtyminen selkeyttääkin itsenäisiä ohjelmistoja, mutta vastapainoisesti monimutkaistaa sovelluskokonaisuuksia.

Myöskin kyky varastoida viestejä kunnes ne käsitellään (tai viestin elinaika loppuu) on yksinkertainen tapa hoitaa kuormantasausta jakamalla viestit useammille käsittelijöille, tai toimimalla käsittelijän ulkopuolisena puskurina. Tästä syystä viestijonojen sanotaan noudattavan kuormantasaajan määritelmää("JMS Load Balancing (Using the JMS Binding Component)" 2010).

Viestijonot poistavat viestin lähettäjältä mahdollisuuden tietää milloin viestit käsitellään. Täten niiden käyttö viestinvälityksessä jossa prosessoinnin täytyy jatkua tietyn aikaikkunan sisällä on riskialtista. Ongelmaa "Viesti tulee käsitellä ajanhetkeen X mennessä" ei ole lähtökohtaisesti mahdollista ratkaista viestijonoilla. Tämän ongelman välttäminen toisesta päästä, eli "Viestiä ei saa käsitellä X ajan jälkeen" voidaan ratkaista joidenkin viestijonototeutusten tarjoaman elinajan kautta (esim. viestijonototetus RabbitMQ tukee tätä ("RabbitMQ - Time-To-Live Extensions" 2017)). Epävarmempi vaihtoehto on lisätä viestiin Time To Live-kenttä ja toivoa että vastaanottava sovellus ei käsittele viestejä, joiden elinaika on umpeutunut.

Viestijonot myös muodostavat jaetun resurssivaatimuksen prosessien tai sovellusten välille, jonka pitää olla aina saatavilla. Mikäli viestijono ei toimi, viestin lähettäjät eivät saa mitään lähetettyä ja kuuntelijat odottavat viestejä tekemättä mitään.

Tutkimuksen kohteena oleva ohjelmisto käytti viestijonoja pilkkoakseen pitkäkestoisia operaatioita lyhytkestoisempiin osiin, sekä uudelleenyrittääkseen tietyistä syistä epäonnistuneita vaiheita. Näitä erikoistilanteita ja niiden toiminnallisuutta ei kuitenkaan tutkielmassa käsitellä, koska refaktorointi ei vaikuttanut sovelluksen kyseiseen osaan.

2.2 Refaktoroinnissa esiintyvät ohjelmistotekniikan käsitteet

Dijkstra selitti jo vuonna 1972 konetehon kasvun mahdollistaneen sekä ohjelmistot että ohjelmointikielet, joiden monimutkaisuuden taso on sen verran huomattava, ettei niiden ymmärtäminen kokonaisuutena ole inhimillisesti mahdollista (Dijkstra 1972).

Näenkin konetehon sekä ohjelmistojen ominaisuuksien kasvaneen monumentaalisesti sitten 70-luvun. Ongelman tästä tekee se harmittava fakta, että ominaisuuksien määrän kasvaessa, ohjelmointityökalut eivät ole kehittyneet samaan räjähdysmäiseen tahtiin. Pahemman tästä tilanteesta saa aikaan se, että useimpiin ongelmiin ei ole vain yhtä oikeaa ratkaisua, vaan vaihteleva määrä eri tilanteisiin sopivia, tai vähemmän sopivia ideoita.

Triviaalina esimerkkinä ratkaisujen monimuotoisuudesta esitän niinkin yksinkertaisen asian kuin lajittelualgortimit. Kyseisiä algoritmeja on huomattava määrä, kuten kuplalajittelu, kekolajittelu, lisäyslajittelu, pikalajittelu ja niin edelleen.

Koska ongelmia voidaan tietotekniikassa yleensä ratkaista usealla eri tavalla, ohjelmistojen potentiaalinen kompleksisuus voi olla huomattava. Tässä kappaleessa käsittelen keinoja, joilla ohjelmistoja voidaan yksinkertaistaa, helpottaen niiden toiminnan ymmärtämistä sekä muokkaamista.

Kappaleessa läpikäyn konsepteja, joita hyödynnetään modulaarisessa ohjelmoinnissa, sekä olio-ohjelmoinnissa, sillä tutkielmassa tarkasteltu ohjelmakoodi on kirjoitettu Java-kielellä. Java on perintää ja rajapintoja tukeva luokkapohjainen olio-ohjelmointikieli, joka mahdollistaa modulaarisuuden.

Kaikkia luvussa käsiteltyjä konsepteja on hyödynnetty joko alkuperäisessä ohjelmassa tai refaktoroidussa ohjelmassa.

2.2.1 Vastuualueiden erottelu

Dijkstra esitteli 1982 käsitteen *"Separation of Concerns"*, jota hän käytti havainnollistamaan 60- ja 70-lukujen taitteessa näkemiään ongelmia tietotekniikan alalla. Tämän lisäksi Dijkstra onnistui tarjoamaan selkeitä esimerkkejä siitä, miten helposti kokonaisuuksiksi miellettävät asiat voivat vaikuttaa olevansa ristiriidassa vaatimuksiensa kanssa, mikäli niitä ei tarkastella osa-alueinaan. (Dijkstra 1982)

Parnas taas ehdotti 1972 julkaistussa artikkelissaan ohjelmien modularisaation suunnittelun aloittamista monimutkaisista tai todennäköisesti muuttuvista kohdista (Parnas 1972).

Sekä Dijkstra että Parnas siis ehdottavat ohjelmistojen pilkkomista selkeästi rajattuihin moduuleihin. Moduulit joiden toiminnallisuus on tarkkaan rajattu, mahdollistavat usean moduulin rinnakkaisen kehittämisen, yksinkertaistavat ongelmien löytämistä, sekä voivat muuttua itsenäisesti ja ilman tarvetta koko ohjelmiston muuttua.

2.2.2 Yhden vastuun periaate

Idea tunnetaan englanniksi nimellä Single Responsibility Principle, ja se jalostaa vastuualueiden erottelun konseptia edelleen rajaamalla ohjelmiston osien vastuita kokonaisuuksiin, joilla on 'tasan yksi syy muuttua' (Martin 2003).

Moduulirajausten jatkoksi moduulin komponenteillekin määritellään yksittäiset vastuut, jotta koodi olisi yksinkertaisempaa ja selkeämpää. Vastuut voidaan määritellä useilla tavoilla, eikä absoluuttista ohjetta tähän metodiin olekaan. Esimerkkinä määrittelyn hankaluudesta toimivat ideat kuten "Tämä komponentti vastaa tietueiden lataamisesta ja tallentamisesta tietokantaan", "Tämä komponentti varmistaa tietueen oikeellisuuden", sekä näitä hyödyntävä "Tämä komponentti tallentaa oikeelliseksi varmistetut tietueet kantaan". Kyse on siis selkeän komposition mahdollistamisesta, eikä ohjelman osien teknisestä luokittelusta.

Vastuun eristäminen on tärkeä asia, kun mietitään ohjelmiston kokoamista, sillä eristämällä implementaatioyksityiskohdat omiin selkeisiin osioihinsa, voidaan korottaa ohjelmiston abstraktiotasoa. Ohjelmakoodista tulee huomattavan paljon luettavampaa, kun voidaan sanoa esim. "Varmista että sisääntuleva tilaus on täytetty oikein ja tallenna se.", sen sijaan että sa-

nottaisi “Varmista että sisääntulevan tilauksen asiakas on asetettu ja siinä on ainakin yksi tuote ja sitten tallenna se JDBC-kutsulla PostgreSQL-kantaan”. Molemmat konseptit voivat toimia täysin identtisesti ja tekniset yksityiskohdat voivat jopa olla identtisiä, mutta luomalla validaatiokomponentti tilauksille, tallennuskomponentti JDBC-kutsujen lähettämiselle tietokantaan, sekä komponentti joka määrittelee operaatiojärjestyksen, voidaan koko asia abstrahoida helpommin ymmärrettäväksi.

Lisähyötynä tämä toiminnallisuus antaa mahdollisuuden muuttaa implementaatioita sekä logiikkaa, vaikuttamatta muun ohjelmiston toimintaan. Kun vastuualueet on jaettu eri komponentteihin, muutokset tapahtuvat vain yhteen paikkaan. Edellisessä esimerkissä käytetyn tiedontallennuskomponentin sisäinen toteutus voi muuttua täysin, mutta muutokset ohjelmakooditasolla pysyvät itse toteutuksen sisällä. Jopa tiedontallennustavan muuttaminen PostgreSQL-kannasta johonkin täysin erilaiseen tallennusratkaisuun on mahdollista toteuttaa täysin komponentin sisäisillä muutoksilla, mikäli komponentin rajapinnat eivät toteutuksen takia muutu.

Edellinen esimerkki havainnollistaa triviaalilla tavalla kolmen erilaisen tehtävän jakamista omiin komponentteihinsa joilla on rajattu vastuu. Kaksi komponenttia mahdollistavat yksittäisen operaation ja kolmas komponentti määrittelee operaatioiden järjestyksen. Tästä edelleen yleistämällä voidaan huomata, että operaatioiden toiminnallisten ykstyiskohtien muuttuessa, koostamistavan ei tarvitse muuttua. Tämä voi myös toimia toisen suuntaan erinäisillä tavoilla, peräkkäisiä operaatioita voi tulla uusia, niitä voi poistua, osasta voi tulla ehdollisia jne.

Liiketoimintavastuualueittain käsitettä havainnollistava esimerkki löytyy Robert Martinin blogista (Martin 2014).

2.2.3 Demeterin laki ja Template Method-suunnittelumalli

Bock (Bock 2003) määrittelee Demeterin lain seuraavasti:

A method of an object should invoke only the methods of the following kinds of objects:

1. itself

2. its parameters
3. any objects it creates/instantiates
4. its direct component objects

Lain keskeinen teema on määrittellä rajoituksia luokkien keskenäisille interaktioille niiden osiensa suhteen.

Esittäisin tämän olion tavoitteena delegoida työnsä toisille olioille, mikäli kyseessä ei ole kompositio. Bockin (Bock 2003) esimerkkiä lainaten, kun asiakas ostaa lehdenjakajalta lehden, on asiakkaan vastuulla kaivaa rahat lompakosta, sohvatyynyjen välistä, tai mistä vaan. Toisaalta, asiakkaan vastulla ei ole päättää laittaako lehdenjakaja saamansa rahat lompakkoon vai vyölakkun. Asiakkaan ja lehdenjakajan ei tarvitse sopia kuin tapa jolla toinen antaa toiselle rahaa. Koodimuodossa tämä voitaisiin ilmaista asiakkaan näkökulmasta vaikka seuraavasti:

```
1  void payForPaper(Paperboy paperboy) {
2    Amount price = paperboy.askPrice();
3    Money payment = this.searchMoneyFrom(price, WALLET, SOFA, PIGGYBANK);
4    paperboy.acceptMoney(payment);
5  }
```

Demeterin Laki siis käsittelee pääasiallisesti olioiden sisältämiä attribuutteja, sekä niiden näkyvyyttä ja käsittelyä. Läheistä sukua sille on GoF:n suunnittelumalli Template Method, joka käsittelee, hieman samaan tapaan, algoritmien tai liiketoimintalogiikan toteuttamista (Gamma ym. 1995).

Template Method-suunnittelumallia noudattamalla voidaan rakentaa korkeamman tason abstraktioita matalan tason implementaatioiden päälle. Malli antaa mahdollisuuden olla välittämättä mitä alemmissa kerroksissa tapahtuu, sekä olla paljastamatta komposition kompleksisuutta tai vaatimuksia ylemmille kerroksille.

Kapselointi on yksinkertaisin esimerkki, jonka avulla Template Method-mallia voidaan havainnollistaa Java-luokissa. Luokka voi määritellä julkisen metodin, joka kutsuu useampia luokan sisäisiä metodeja, tai sen luokka-attribuuttien metodeja, on täydellinen esimerkki. Luokan käyttäjien ei tarvitse tietää kuin julkisesti näkyvä metodi, joka hoitaa toiminnalli-

suutensa täysin piilossa muilta.

Tällaisilla metodeilla voidaan helposti kuvitella olevan useita syitä pienentää julkisesti näkyvää rajapintaansa:

- Metodi muokkaa luokkamuuttujia muutenkin kuin korvaamalla yhden arvoa suoraan toisella
- Metodi suorittaa tiettyä järjestystä vaativaa kompositioita
- Metodin kompositio delegoi operaatioita luokkamuuttujiksi määritellyille oliolleen
- Metodi tekee paljon asioita

Sekä Demeterin Laki että Template Method-suunnittelumalli tähtäävät sinällään samaan asiaan: abstrahoimaan operaatioiden yksityiskohtia piiloon niiden käyttäjiltä.

2.2.4 Inversion of Control ja Dependency Injection

"Dependency injection means giving an object its instance variables. Really. That's it." - James Shore (Shore 2006).

Periaatteena *"Inversion of Control"*, suomennettuna täysin ideaa kuvaamaton *"kontrollin kääntäminen"*, on olio-ohjelmoinnissa käytetty tekniikka, jossa yksinkertaisimmillaan asetetaan luokan staattisiin kenttiin tai olioinstanssin luokkamuuttujiin arvoja. Tässä tutkielmassa rajataan Inversion of Control-periaate vain Javan olioinstanssin arvojen asettamiseen, sillä muita mekaniikkoja tai kieliä ei tukimuksen kohteena olevassa ohjelmassa käytetty.

Olio voi sisältää toisen olioinstanssin, sisällytetty olioinstanssi voidaan joko luoda olion itsensä toimesta tai asettaa olioinstanssille ulkoapäin. Yksinkertaisena Java-koodiesimerkkinä seuraavasti:

Olion toimesta

```
1   public Foo() {
2     this.bar = new Bar();
3   }
```

Konstruktorissa ulkoapäin

```
1   public Foo(Bar bar) {
```

```
2       this.bar = bar;
3   }
```

Erillisellä metodilla

```
1   public void setBar(Bar bar) {
2       this.bar = bar;
3   }
```

Olion itsensä toimesta riippuvuuksien asettaminen on Inversion of Control-periaatteen vastaista, ja sitä noudattavat luokat yleensä tarjoavatkin yhden jälkimmäisistä tavoista asettaa olioinstansseja toisen olioinstanssin sisään.

Mekaanisella tasolla riippuvuksien asettamisen (engl. *Dependency Injection*) suorittaa ohjelmakoodi. Yksinkertaisin esimerkki tästä on Factory Method Pattern (Gamma ym. 1995), kun taas voimakkaampia esimerkkejä on Googlen Guice-kirjasto. Kyseisiä työkaluja konfiguroidaan tuottamaan niiltä halutut instanssit käyttötapaukseen sopivilla tavoilla. Tämän konfiguraation mukaisesti sitten työkalut voivat tarjota mahdollisesti kymmeniä, satoja, tai vielä useampia yksittäisiä olioviitteitä käyttävälle sovellukselle tavan vähentää toistoa koko olioketjun koostamisessa. Dependency Injection-työkalu siis ottaa parhaimmillaan vastuulle koko sovelluksen olioinstanssien vaadittavan luomisjärjestyksen selvittämisen, instanssien luomisen, sekä näiden paikoilleen asettamisen niitä käyttäviin toisiin instansseihin.

Esimerkki Guice-kirjasto osaisi tarvittaessa asettaa Foo-luokan Bar-luokkamuuttujan:

```
1   public class Foo {
2     @javax.inject.Inject
3     private Bar;
4   }
```

Mikäli Bar-luokka olisi rajapinta tai sillä olisi vaihtoehtoisia toteutuksia, jotka noudattavat Liskovin korvattavusperiaatetta, jonka Martin avaa pikaisesti sivuillaan (Martin 2005) ja tarkemmin kirjassaan (Martin 2003), voitaisiin Dependency Injection-työkalu konfiguroida asettamaan mikä tahansa Bar-luokkaa perivä tai toteuttava luokka esimerkissä käytettyyn Foo-olioon sen instantaatiovaiheessa.

Edellisessä esimerkissä Guice-kirjaston työkaluilta pyydettäessä uutta Foo-luokkaa, Guice

etsisi luokan @com.google.guice.Inject, tai @javax.inject.Inject-annotaatiolla merkityt luokkamuuttujat, ja asettaisi siihen Bar-luokan instanssin. Teknisesti erilaisia mahdollisuuksia Dependency Injection-työkalujen toteutukseen ja toiminnallisuuteen on useita, mutta niitä ei tutkielmassa ole tarpeen käsitellä pintaraapaisua syvemmälti.

Ohjelmistokehitystyössä olen huomannut tekniikan olevan hyödyllinen ainakin seuraavissa tilanteissa:

- Luokkia rakennetaan käyttämään rajapintoja tai abstrakteja luokkia.
- Oliosta halutaan asettaa sama olioinstanssi useampaan olioon.
- Oliosta halutaan asettaa eri olioinstanssit kaikkiin sitä käyttäviin olioihin.
- Joku sekoitus kahdesta edellisestä.
- Koko ohjelman olioiden instantoiminen manuaalisesti on huomattavasti laajempi työ kuin Dependency Injection-työkalun vaatima lisätyö.
- Luokat toteutetaan noudattaen Template Method-suunnittelumallia (Gamma ym. 1995), jolloin pitkä tai monimutkainen tehtävä voi abstraktoitua useampiin sisäkkäisiin luokkiin.

Kyseisten tilanteiden manuaalinen toteutus ilman Dependency Injection-työkaluja voi vaatia huomattavasti useamman rivin koodia, kuin työkalujen konfiguraatio. Tosin kyseiset työkalut eivät ole mikään välttämättömyys. Pienissä ohjelmissa muutama koodirivi, joissa riippuvuudet asetetaan muutamaan luokkaan käsin, on hyvinkin tehokas lähestymistapa, sillä riippuvuudet lisäävät aina koodia tai konfiguraatiota, joka taas lisää tarvetta ylläpitotyölle, joka pahimmillaan vie aikaa ohjelmoijien päätavoitteilta (Shaw 2017).

Tarkempia teknisiä selityksiä löytyy esimerkiksi Martin Fowlerin kirjoittamana (Fowler 2004).

2.2.5 Luokkatason yksikkötestaus Mock-olioilla

Template Method-mallia tai Demeterin lakia noudattavia ohjelmistoja voidaan yksikkötestata luokkatasolla, käyttämällä nk. *Mock*-olioita. Mock-oliot voidaan luoda joko perimällä alkuperäinen luokka, tai käyttämällä työkalua joka tekee sen dynaamisesti. Mock-oliot (tai dynaamisesti konfiguroitavat instanssit) asetetaan käyttäytymään ennallamäärätyllä tavalla. Nämä Mock-instanssit injektoidaan testattavaan olioinstanssiin, jonka jälkeen testat-

tavan olioinstanssin käyttäessä ulkoisia riippuvuuksiaan, voidaan testattavan olioinstanssin suorittamien kutsujen vastauksiin vaikuttaa sekä niiden kutsukertoja kirjata ylös. Tämä antaa laajoja mahdollisuuksia suorittaa White-box testausta(Nidhra 2012) järjestelmän rajatuille osioille.

Luokkatason yksikkötestaus Mock-olioilla on yleensä kompleksisuudeltaan vähäisempää kuin luokkatason yksikkötestaus koko toteutuksella. Havainnollistan tätä mukaellen töissä useasti törmäämääni ongelmaan monivaiheisen operaation mahdollisten ongelmien kertautumisesta.

Esimerkkiin kuulukoon olio, joka sisältä useita metodeja, joista jokainen hyväksyy eri tavalla enkoodatun viestin. Jokainen näistä metodeista tekee seuraavat asiat:

1. Dekoodaa spesifin muotoinen viesti yleiseen viestimuotoon
2. Palauta virhe, mikäli viestin sisältö ei ole määrittelyn mukainen
3. Tallenna viesti

Laskettaessa mahdollisia syitä, miksi jokainen vaihe voisi epäonnistua, päästään triviaalilla tavalla seuraavaan kaavaan:

[viestin dekoodamisen epäonnistumisen mahdollisuudet] * [viestin sisällön tavat poiketa määrittelystä] * [tallennus onnistui tai ei]

Yhtälöstä muodostuva luku on aina toteutuksen mukaan yksilöllinen, mutta kyseessä on aina kertolasku, esimerkiksi 7 * 14 * 2 = 196.

Jos ohjelma taas eriytetään Yhden Vastuun periaatteella useampaan yksinkertaiseen komponenttiin ja implementaatiota korvataan Mock-oliolla, tilanne yksinkertaistuu huomattavasti. Edeliseltä, täysin hypoteettiselta, kertolaskulta voidaan jokaisen askeleen sijaan kysyä “monta vaihtoehtoista lopputulosta sinulla on?”. Esimerkkinä voisi olla vaikka seuraava lista:

1. Viestimuodon konversio: 7 tapaa epäonnistua, mutta erilaisia vastauksia on yleisellä tasolla vain kaksi: “Tässä on muutettu viesti”, tai “Muunnosvirhe, hylkää viesti.”
2. Validaatio: 14 tapaa epäonnistua, mutta vastauksia vain “validi” ja “ei validi”
3. Tallenna viesti: “onnistui” ja “ei onnistunut”

Esimerkissämme on kuvitteellisen tilanteen 196 mahdollista tilannetta yksinkertaistettu 2*2*2 = 8 tilanteeseen. Tämä määrä on huomattavasti yksinkertaisempi testata.

Lähestymistavassa on omat ongelmansa, eivätkä kaikki tilanteet ole näin yksinkertaisia. Kompositioita tehtäessä voidaan kuitenkin tällä menetelmällä kirjoittaa testit pienempiä koodinpaloja vasten.

Henkilökohtaisen kokemukseni mukaan luokkatason yksikkötestauksen suurimmat edut ovat olleet mahdollisuus paikallistaa bugit tiettyyn kerrokseen, sekä mahdollisuus kirjoittaa vain yhtä kerrosta koskettavia testejä. Näitä testejä voidaan parhaimmillaan ajaa ilman ulkoisten resurssien kuten tietokantojen määrittelyä ja jokaisen koodimuutoksen jälkeen muutamassa sekunnissa, kunnes koodi toimii halutulla tavalla.

Lähestymistavassa on kuitenkin huomioitava, että mikäli useampi testi udelleenmäärittelee samat Mock-oliot, huomattavana riskinä on määritellä toiminnallisuus ristiin. Lisäksi, mikäli testissä alustetaan Mock-olio toimimaan tietyllä tavalla, mutta alustus ei muualla koodissa tehtyjen muutosten takia enää vastaakaan todellisuutta, on tuloksena helposti valheellisesti positiivisia tuloksia palauttava testi.

Ellei siis koko järjestelmä määrittele yhtenäisesti ulkoisten rajapintojen Mock-toteutuksia, riskinä on että kaikkia viitteitä ei muuteta uuden tilanteen vaatimuksiin.

Tarkemmin asiasta ovat kirjoittaneet Mackinnon, Freeman, sekä Craig (Mackinnon, Freeman ja Craig 2001).

Itse olen myös huomannut Mock-olioiden käyttämisen olevan erinomainen tapa TDD:tä (Torchiano, Morisio ja Erdogmus 2005) harjoittaessani ilmoittaa milloin luokan kompleksisuus alkaa kasvaa liian suureksi. Kun testien kirjoittamisessa suurin osa ajasta alkaa mennä Mock-olioiden tilan säätämiseen halutun laiseksi, on aika pilkkoa toteutusta vähemmän kompleksisiin osioihin. Tämä lähestymistapa auttaa minua mielestäni kirjoittamaan selkeämpää koodia, sillä jos itse en voi kirjoittaa formaalia varmistusta koodin toiminnasta järkevästi samalla kun kirjoitan kyseisen toiminnallisuuden, ei muiden missään nimessä ole helppoa ymmärtää mitä koodi tekee.

3 Tutkimusmetodi

Kuten March ja Smith linjaavat artikkelissaan (March ja Smith 1995), tietotekniikan ja tietojärjestelmien tutkimus on selkeästi ei-luonnollinen tieteenala. Täten mitään absoluuttisesti todeksi näytettävää johtopäätöstä ei tutkimuksessa voida muodostaa. Tutkimuksesta ei tulekaan vetää suoria johtopäätöksiä, tai kuvitella että tutkimuksessa suoritetun refaktoroinnin jäljitteleminen tuottaisi vastaavanlaisia vaikutuksia muissa ohjelmistoissa. Ohjelmistot ovat kuitenkin niiden toteuttajiensa luomuksia, eikä niiden muotoa lähtökohtaisesti rajoita kuin ohjelmointikielen tekniset rajoitteet.

3.1 Tutkimuksen tavoite ja evaluointi

Crnkovicin kuvaus konstruktiivisestä tutkimuksesta (Crnkovic 2010) on ehkä selkein ilmaus mitä tutkimuksen lähtökohdista voidaan antaa: “Constructive research method implies building of an artifact (practical, theoretical or both) that solves a domain specific problem in order to create knowledge about how the problem can be solved (or understood, explained or modeled) in principle.”.

Tavoitteenani oli verrata kahta eri ajankohdan konstruktiota samasta ohjelmistosta; lähdekoodia ennen refaktorointia, sekä lähdekoodia refaktoroinnin jälkeen. Refaktoroinnin lähtökohtana oli yleistää käytettävän sovelluksen rakennetta ja maksimoida olemassaolevan koodin uudelleenkäytettävyyttä, kun lähdetään tarjoamaan Data Acquisition-palveluita uusille kolmansille osapuolille. Näiden kahden konstruktion, tai version, eroja vertailemalla koitin tunnistaa erilaisia ohjelmistotekniikan tunnistamia käsitteitä ja malleja, sekä miettiä yleisesti muutosten vaikutusta ohjelmiston yleiseen laatuun.

Tutkielman alaa on kuitenkin rajattu tietoisesti kolmesta syystä. Ensiksikin analysoitava ohjelmisto on oikean koodin sijaan anonymisoitu reproduktio, joten siitä puuttuu huomattava määrä tuotanto-ohjelmiston vaatimia metriikka-, raportointi-, ja virheenhallintaominaisuuksia. Toiseksi se ei keskity useampaan kuin yhteen sovellukseen eri ajankohdissa. Kolmanneksi ohjelmistoon suoritettuja muutoksia rajoitettiin liiketoimintavaatimusten (raportissa ilmenevä “minimoikaa muutokset”, sekä itsestäänselvä “älkää käyttäkö asian absoluuttiseen

hiomiseen kolmea vuotta”-mentaliteetti) takia, eikä refaktoroinnin lopullinen tuote tuo esille kuin “tarpeeksi hyvän” version refaktoroidusta ohjelmistosta.

Näistä syistä tutkielmassa ei yritetä metsästää absoluuttisen yleispäteviä periaatteita ohjelmatoteutusten uudelleenkäytettävyydestä, eikä näiden suhteesta ohjelmistossa esiintyvien suunnittelutieteellisten käsitteiden määrään.

Tämän sijaan tutkimus keskittyi arvioimaan muutosten vaikutusta kyseiseen ohjelmistoon kolmella kriteerillä:

- Erilaisten ohjelmointiteknisten periaatteiden ja mallien hyödyntämisestä saavutettu uudelleenkäytettävyys
- Uuden liiketoimintakumppanin kanssa toimimiseen vaaditun lisäkoodin määrän minimointi, joka saatetaan saavuttaa ohjelmiston jaottelulla kahteen kategoriaan, eli yhteiseen sekä uudelleenkäytettävään koodiin
- Oliko ohjelmiston yleistäminen ja osien uudelleenkäyttö liiketalodellisesti kannattavaa

Näiden lisäksi refaktoroinnin läpikäynnissä koitettiin huomioida asioita jotka parantivat koodin luettavuutta, tai tunnistaa rajoitteita jotka estivät luettavuuden parantamista.

3.2 Tutkimuksen kulku

Kuten teknisten käsitteiden selityksen yhteydessä on annettu ymmärtää, on tutkimuksen kohteena Data Acquisition-palvelu, joka kuuntelee viestijonoa, tarkoituksenaan saada tietoonsa uusien tietueiden saapuminen. Tietueet ladataan tietovarastosta, muutetaan muotoon, joka käy kolmannelle osapuolelle, sekä lähetetään eteenpäin. Ohjelmiston alkuperäisen luomisen päämääränä on ollut automatisoida manuaalinen prosessi. Jatkopäämääränä, ja sitä myöten ohjelmiston refaktoroinnin tavoitteena, on ollut laajentaa prosessin automaatio useammille kohdejärjestelmille, joiden käytössä on mahdollisesti erilaisia tietomalleja.

Tutkimuksen pohja luodaan selittämällä mahdollisimman monta suunnittelutieteen konseptia jotka esiintyvät alkuperäisessä tai refaktoroidussa ohjelmakoodissa.

Tämän jälkeen pyritään tunnistamaan mitkä konsepteista ovat vaikuttaneet refaktorointiin,

jotta voidaan erottaa olivatko konseptien hyödyt jo valmiiksi mukana ohjelmassa. Tämä analyysi suoritetaan mahdollisen jatkotutkimuksen takia, jolla saatettaisiin selvittää eroja alkuperäisen toteutuksen ja refaktoroinnin tuloksen suhteen.

Lopuksi tarkastellaan kyseisen refaktoroinnin laatuvaikutuksia ja aikasäästöjä jatkototeutuksen kannalta, jotta voidaan arvioida millaisia vaikutuksia refaktorointityöllä oli resursointimielessä ja koodin laadun suhteen.

4 Sovelluksen kuvaus sekä yleistyksen liiketoimintamotiivi

Kappaleessa tarkastellaan sovelluksen toimintaa, käydään läpi sovelluksen toimintalogiikka, sekä selitetään mitkä liiketoiminnnan tavoitteet johtivat sovelluksen yleistämiseen.

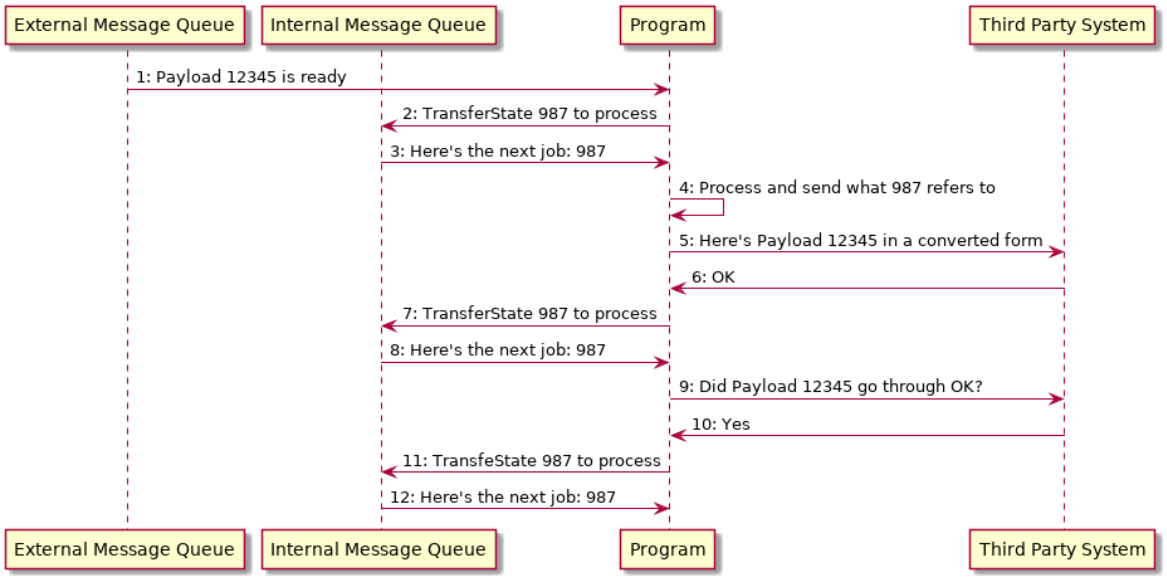
4.1 Yleistettävän sovelluksen toiminnan kuvaus

Tutkielman aiheena olevan sovelluksen alkuperäinen tarkoitus oli toteuttaa käyttäjägeneroidun datan automaattinen siirto asiakkaan ja kolmannen osapuolen välillä. Sovellus kytkettiin kiinni tapahtumalähteeseen, joka ilmoitti siirrettävän tietueen olevan valmis. Tällöin sovellus latasi kyseisen tietueen, varmisti automaattisen tiedon edelleenlähetyksen olevan luvallinen kyseiselle tietueelle, muutti tiedon vastaanottajan odottamaan muotoon, sekä lähetti muutetun tietueen kolmennelle osapuolelle. Kun tieto oli saatu lähetettyä, varmistettiin erillisessä vaiheessa, että lähetys oli saatu onnistuneesti tallennettua.

Sovelluksen alkuperäinen toteutus koostui kahdesta viestijonokuuntelijasta, sekä yhdestä prosessointimekanismista.

Viestijonokuuntelijoista ensimmäinen toimi tapahtumakuntelijana, joka sai tiedon aina kun tietueen tila muuttui. Toinen taas kuunteli sovelluksen sisäistä viestijonoa, jota käytettiin kuormantasaukseen ja prosessointimekaniikan yksinkertaistamiseen. Suunnitteluvaiheessa nähtiin helpommaksi pitää työjonon tila viestijonoissa, kuin rakentaa useamman sovellusinstanssin välistä lukitus- ja uudelleenyritysmekanismia.

Alkuperäinen prosessointimekanismi sisälsi lähes kaiken toimintalogiikan, kun taas refaktoroitu prosessointimekanismi delegoi vastuita muille luokille, jotka koittivat toteuttaa Yhden Vastuun Periaatetta. Molemmissa mekanismeissa riippuvuudet oli liitetty toisiinsa Dependency Injection-mekaniikoilla.



Kuvio 1. Ohjelman yleistason toiminta

Kuvio 1 selittää ohjelman toiminnan yleisellä tasolla. Kuviossa on mukana viestien lopullinen kohde, molemmat viestijonokuuntelijat, sisäinen ja ulkoinen, sekä itse ohelma. Kuvion esimerkissä Payload 12345 (tietue pääjärjestelmässä valmistuu lähetettäväksi, josta luodaan uusi TransferState 987 (siirtotilanteen tietue, viittaa pääjärjestelmän tietueeseen 12345).

Viestijonojen avulla hallitaan kuormantasausta ja ulkoistetaan ohjelman vastuu jokaisen prosessoitavan tilan säilyttämisestä viestojonon vastulle.

Samalla kun viestit odottavat viestijonossa, voi itse ohjelma prosessoida viestejä vain halutun määrän kerrallaan. Tällä tavoin ohjelma suorittaa operaatioita määrättyyn tahtiin, ottaen uuden operaation käsittelyyn vasta kun edellisiä on suoritettu tarpeeksi.

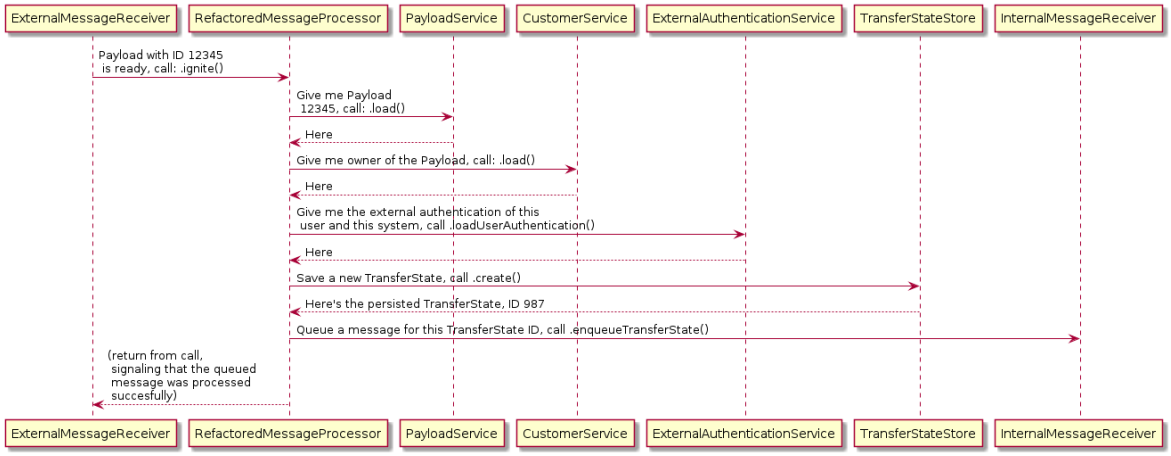
Kuoviossa mainittu “Third Party System” on minkä tahansa asiakkaan järjestelmä, jolle Payloadeja lopulta välitetään.

4.2 Vaiheiden sekvenssit

Sovelluksen sekvenssikaaviot aikaan ennen refaktorointia ja sen jälkeen muistuttavat sen verran paljon toisiaan, että niiden erillinen tarkastelu ei sinällään ole hyödyllistä. Refaktoroinnin

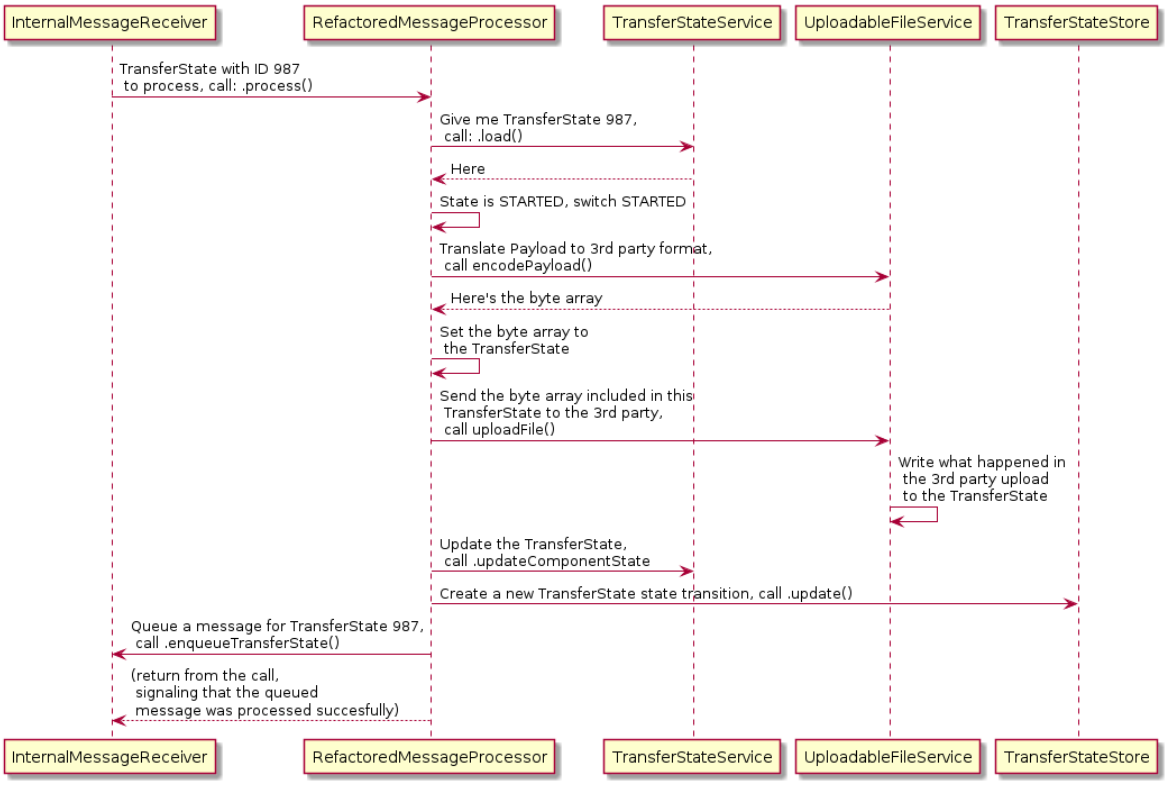
jälkeen kuitenkin ohjelman suoritus vaihtelee useamman luokan välillä, joka selkeyttää toimintojen kontekstia. Tämän takia koko sovelluksen 1 vaiheet 1-12 esitetään tarkemmin vain refaktoroidun version avulla.

Vaiheet 1-2. Aluksi tieto Payload 12345:n valmistumisesta saapuu ohjelmaan. Ohjelma varmistaa että Payload 12345:n omistaja on sallinut siirron kolmannen osapuolen järjestelmään. Tämä tieto tallennetaan järjestelmän tietovarastoon ja TransferState 987:n olemassaolosta viestitään sisäiseen viestijonoon.



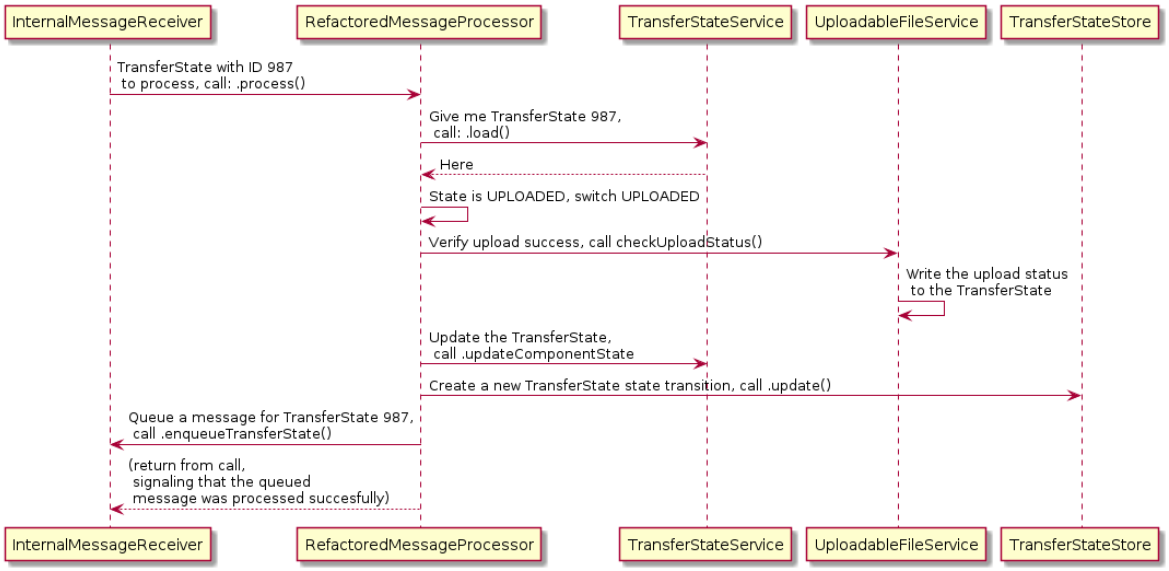
Kuvio 2. Vaiheet 1-2. Payload on valmis käsiteltäväksi

Vaiheet 3-7. Payload 987 on tässä vaiheessa kannassa tilassa STARTED. Kun vastaava viesti luetaan sovelluksen sisäisestä viestijonosta, TransferState ladataan kannasta, sen viittama Payload konvertoidaan, sekä lähetetään kolmannen osapuolen järjestelmään. TransferState 987:n tilaa päivitetään tietovarastossa ja se lähetetään takaisin sisäiseen viestijonoon.



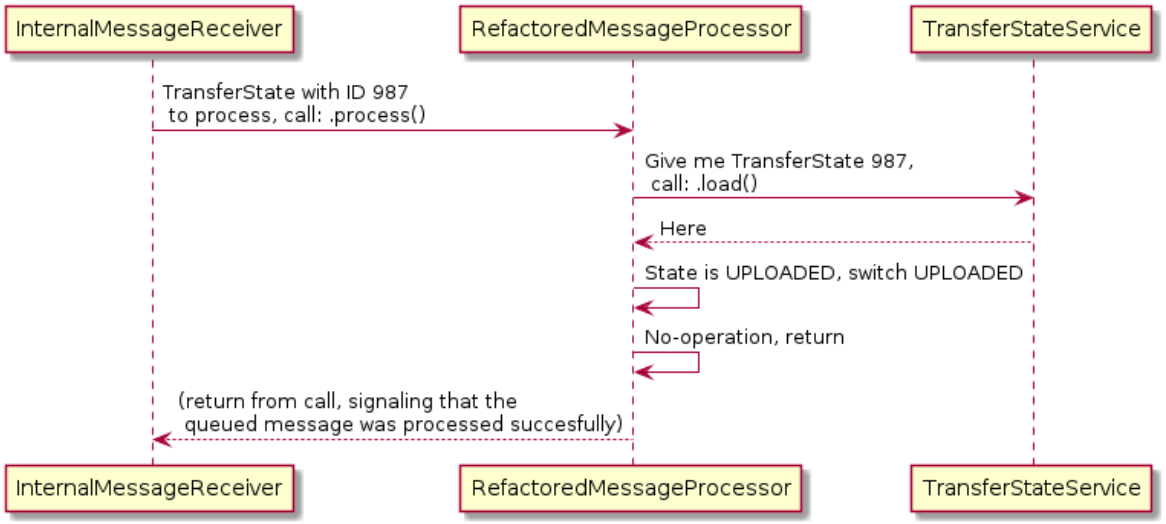
Kuvio 3. Vaiheet 3-7. Viesti lähetetään kolmannen osapuolen järjestelmään

Vaiheet 8-11. Payload 987 on tässä vaiheessa kannassa tilassa UPLOADED. Kun vastaava viesti luetaan sisäisestä viestijonosta, TransferState ladataan kannasta, sekä sen viittaaman Payloadin tilanne tarkastetaan kolmannen osapuolen järjestelmästä. TransferState 987:n tilaa päivitetään tietovarastossa ja se lähetetään takaisin sisäiseen viestijonoon.



Kuvio 4. Vaiheet 8-11. Varmistetaan viestin onnistunut siirto kolmannen osapuolen järjestelmään

Vaihe 12. Payload 987 on tässä vaiheessa kannassa tilassa FINISHED. Kun vastaava viesti luetaan sisäisestä viestijonosta, voidaan todeta Payload 987:n käsittelyn päättyneen ja suorittaa halutut raportointitoimet.



Kuvio 5. Vaihe 12. Viesti siirretty ja kuitattu onnistuneesti.

4.3 Yleistämisen liiketoimintatarve

Sovelluksen julkaisun jälkeen asiakas teki liiketoimintapäätöksen, jonka tavoite oli koittaa hioda sovellusta ja muokata sitä, mahdollistaen palvelun tarjoaminen ensimmäisen yhteistyökumppanin lisäksi muillekin potentiaalisille kumppaneille. Tavoitteet refaktoroinnissa olivat yleisen toiminnallisuuden ja kumppanispesifin toiminnallisuuden erottaminen toisistaan, sekä kumppanispesifien toiminnalisuuksien eriyttäminen rajapintojen taakse. Tämän erottamisen lopulllinen päämäärä oli luoda useita ohjelmistoja, jotka hyödyntävät suurilta osin jaettua ohjelmakoodia, mutta jokainen ohjelmisto toimisi itsenäisesti.

Yleistämiseen päädyttiin asiakkaan teknisten asiantuntijoiden arvioiden pohjalta. He näkivät että:

1. Yleistäminen vaikuttaa mahdolliselta
2. Koodin laatu paranisi
3. Voitaisi mahdollisesti saavuttaa lähinnä refaktoroimalla

Koska asiakas halusi minimoida riskit, refaktoroinnille asetettiin tavoite pitää koodin täydellisen uudelleenkirjoituksen määrä mahdollisimman pienenä. Yleistäminen tulisi siis tehdä pääasiassa refaktoroimalla nykyistä koodia.

5 Refaktorointi

Refaktorointi saatiin loppujen lopuksi kohdennettua OldMessageProcessor.java-luokan pilkkomiseen. Kyseinen luokka toteuttaa kahden julkisen metodin rajapintaa, joka ei refaktoroinnin yhteydessä muuttunut. Sekä vanha että uusi toteutus siis voidaan asettaa mihin tahansa, missä käytetään MessageProcessor-rajapintaa.

Ensimmäinen metodi, ignite(int payloadId), suorittaa seuraavat operaatiot:

1. Vastaanottaa jonokuuntelijalta saadun tiedon uudesta viestistä
2. Varmistaa että automaattinen datamigraatio oli päällä viestin luoneelle käyttäjälle
3. Luo kantaan rivin josta käy ilmi migraation tila kyseiselle tietueelle
4. Lähettää sovelluksen sisäiseen viestijonoon viestin joka sisältää migraation tunnisteen.

Toinen metodi, process(int transferStateId), on idealtaan selkeä. Sitä kutsutaan ohjelman sisäisen viestijonon kuuntelijan toimesta ja sille toimitetaan migraation tunniste. Tämä tunniste ladataan tietovarastosta, tarkastetaan migraation nykyinen tila ja suoritetaan migraatioprosessin mukaisesti järjestyksessä seuraava operaatio.

Luokka RefactoredMessageProcessor paljastaa täsmälleen samat julkiset metodit, kuin OldMessageProcessor. Tämä toteutus mahdollisti koko ohjelmistoon vaikuttavien muutosten minimoimisen. Lisähyötynä oli selkeä rajaus ohjelmiston yleisistä osista, verrattuna kolmansien osapuolten eroavuuksien vaatimiin spesifeihin toteutuksiin.

5.1 Esimerkkiohjelmakoodin rajoitteet

Alkuperäinen ohjelmakoodi on salaista, joten refaktoroinnin analyysi suoritetaan anonymisoidusta ohjelmakoodista, joka on toiminnallisesti identtistä. Anonymisoinnissa toimialakohtaiset termit on korvattu termeillä, jotka kuvaavat yleisellä tasolla mitä ohjelmakoodissa tapahtuu.

Anonymisoinnin lisäksi, analysoitavassa ohjelmakoodissa on oiottu muutamissa kohdin. Lähdekoodi on muodostettu alkuperäisen lähdekoodin pohjalta, noudattaen muutamaa yleistä

sääntöä. Tavoitteena oli yksinkertaistaa koodilistauksia eriyttämällä mahdollisimman paljon refaktoroinnissa ennalleen jäänyttä toiminnallista koodia rajapinnoiksi, sekä karsimalla koodin käsittelemiä ei-toiminnallisia luokkia sisältämään vain ne tietueet, jotka ovat toimintalogiikan takia tarpeellisia.

Anonymisointiin ja koodin määrän vähentämiseen käytetyt periaatteet olivat:

1. Viestit sisältävät aina oikeellisia arvoja.
2. Metodit joilla on paluuarvo ilmoittavat virheestä NULL-arvolla.
3. Metodit eivät määrittele heittävänsä poikkeuksia.
4. Viestijonokuuntelijat eivät tee minkäänlaista retry-vai-abort -jaottelua RuntimeException-yksinkertaistuksen takia.

Näiden periaatteiden jälkeen, anonymisoitu versio lähdekoodista on koitettu pitää rakenteeltaan mahdollisimman paljon alkuperäistä toteutusta vastaavana, mutta metodien nimiä ja viittauksia on koitettu muuttaa luettavammaksi.

Valitettavasti kyseinen prosessi on jättänyt muutamia arkkitehtuurillisia sekavuuksia koodiin. Näistä pahimmat koitetaan mainita, kun niitä käsitellään.

Liiketoimintavaatimus refaktoroinnille oli pitää funktionaaliset muutokset minimissä, sillä sovellus oli refaktoroinnin aikaan jo aktiivisessa liiketoimintakäytössä.

5.2 Ohjelmakoodin muutokset refaktoroinnissa

Edellämainitun muutosten minimoinnin vaatimuksen takia, refaktorointi jättää koskematta muutamaan kohtaan, joihin olisi saanut tehtyä isojakin muutoksia selkeyden ja toimintavarmuuden parantamiseksi.

Viittaukset tässä kappaleessa suoritetaan lähtökohtaisesti yleistason 1 vuokaavioon, sillä refaktorointia ei tarkastella järjestyksessä, vaan matkaamalla loogisesti korkeamman abstraktion tasolta aina matalammalle abstraktion tasolle. Notaatio on valittu luettavuuden takia, jotta lukijalla olisi vain yksi kaavio referoitavana.

5.2.1 ignite(int)

Ainoa muutos tässä osassa on autentikaatiotunnisteen lataaminen injektoidun konfiguraatio-objektin tilan mukaan, kovakoodatun vakion sijaan. Muutoksen pieni koko johtui siitä, että autentikaatioita käsittelevä osio oli asiakkaan olemassaolevan infrastruktuurin takia yleistetty tukemaan 1-N-tyyppistä suhdetta loppukäyttäjien ja potentiaalisten autentikaatiokohteiden välillä.

OldMessageProcessor.java

```
1  public void ignite(int payloadId) {
2    Payload payload = payloadService.load(payloadId);
3    Customer customer = customerService.load(payload.getOwner());
4    ThirdPartyAuthentication authentication =
         externalAuthenticationService.loadUserAuthentication(
5      customer.getId(), THIRD_PARTY_SERVICE_NAME);
6
7    if (authentication != null) {
8      TransferState transferState = this.createNewTransferState(payloadId, payload.getOwner(),
           EntityType.ORDER);
9
10     messagingInterface.enqueueTransferState(transferState.getDatabaseId());
11   }
12 }
```

RefactoredMessageProcessor.java

```
1  public void ignite(int payloadId) {
2    Payload payload = payloadService.load(payloadId);
3    Customer customer = customerService.load(payload.getOwner());
4    ThirdPartyAuthentication authentication =
         externalAuthenticationService.loadUserAuthentication(
5      customer.getId(), config.getThirdPartyServiceName());
6
7    if (authentication != null) {
8      TransferState transferState = this.createNewTransferState(payloadId, payload.getOwner(),
           EntityType.ORDER);
9
10     messagingInterface.enqueueTransferState(transferState.getDatabaseId());
```

```
11        }
12    }
```

Tämä metodi vastaanottaa koko prosessin aloituksen, eli kuviossa 1 vaiheet 1 ja 2.

5.2.2 process(int) - switch-casen ulkopuolinen osuus

Vertailemalla OldMessageProcessor- ja RefactoredMessageProcessor-luokkien koodia ennen switch-lauseketta huomataan että muutoksia ei ole tapahtunut.

Tämä kohta ohjelmistoa on yleinen kontrollikohta switch-case-rakenteensa takia. Se valitsee mihin ohjelman suoritus seuraavaksi siirtyy 1 kohdissa 3, 8, ja 12.

OldMessageProcessor.java

```
1   public void process(int transferStateId ) {
2       TransferState state = transferStateStore .load( transferStateId );
3       UploadStatus status = state .getUploadStatus();
4
5       switch ( status ) {
```

RefactoredMessageProcessor.java

```
1   public void process(int transferStateId ) {
2       TransferState state = transferStateStore .load( transferStateId );
3       UploadStatus status = state .getUploadStatus();
4
5       switch ( status ) {
```

Koska switch-rakenne jatkuu metodin loppun asti, on hyödyllisempää tarkastella rakenteen osia valinta kerrallaan, kuin koko metodia.

5.2.3 process(int) - switch STARTED

Jos vertaillaan metodin osia ennen refaktorointia:

OldMessageProcessor.java

```
1           case STARTED:
```

```
2            byte[] encodedPayload = this.encodePayload(state);
3            state.setEncodedPayload(encodedPayload);
4            this.uploadFile(state);
5            this.updateComponentState(state);
6            this.saveComponent(state);
7
8            if (UploadStatus.ERROR != state.getUploadStatus()) {
9              messagingInterface.enqueueTransferState(state.getDatabaseId());
10           }
11           break;
```

ja sen jälkeen:

RefactoredMessageProcessor.java

```
1      case STARTED:
2        byte[] encodedPayload = uploadableFileService.encodePayload(state);
3        state.setEncodedPayload(encodedPayload);
4        uploadableFileService.uploadFile(state);
5        transferStateService.updateComponentState(state);
6        transferStateStore.update(state);
7
8        if (UploadStatus.ERROR != state.getUploadStatus()) {
9          messagingInterface.enqueueTransferState(state.getDatabaseId());
10       }
11       break;
```

nähdään selvästi, että muutokset itse metodissa koostuvat pelkästään luokan sisäisten metodien siirtämisestä omiin luokkiinsa. Alkuperäisessä toteutuksessa OldMessageProcessor-riksi anonymisoitu luokka joko vastasi itse kaikesta logiikasta, tai piilotti toisten luokkien kutsumisen omiin alimetodeihinsa.

Alkuperäinen ratkaisu oli ongelmallinen kahdesta syystä. Ensiksikin koodin määrä näennäisesti vähenee yhdessä paikkaa, mutta tämä on hyvä asia vain mikäli ohjelman sisäiset tilamuutokset viestitään näkyvästi. Tässä tapauksessa updateComponentState-metodi tekee näkymättömiä tilamuutoksia state-oliolle, sekä tallentaa tiedon “state muuttui tilasta A tilaan B” persistenttiin tietovarastoon. Voidaankin väittää, että pelkän “updateComponentState”-

nimeämisen perusteella ei ole mahdollista ymmärtää metodin mahdollisia sivuvaikutuksia, tai ongelmatilanteita. Toinen ongelma on Yhden Vastuun Periaatteen rikkominen, joka refaktoroimattomassa toteutuksessa hankaloittaa luokkatason yksikkötestaamista, sekä kasvattaa luokan monimutkaisuutta.

Yleistasolla 1 tämä metodi vastaanottaa vaiheen 3 viestin, suorittaa vaiheet 4-6, sekä lähettää vaiheen 7 viestin.

5.2.4 process(int) - switch UPLOADED

Tässäkään metodin osassa ei ole muita muutoksia kuin luokan sisäisten metodien siirtämistä alkuperäisen luokan ulkopuolelle. Ennen refaktorointia toteutus näytti tältä:

OldMessageProcessor.java

```
1        case UPLOADED:
2          this.checkUploadStatus( state );
3          this.updateComponentState(state);
4          this.saveComponent(state);
5
6          if (UploadStatus.ERROR != state.getUploadStatus()) {
7            messagingInterface.enqueueTransferState( state.getDatabaseId());
8          }
```

ja sen jälkeen tältä:

RefactoredMessageProcessor.java

```
1      case UPLOADED:
2        uploadableFileService.checkUploadStatus( state );
3        transferStateService.updateComponentState(state);
4        transferStateStore.update( state );
5
6        if (UploadStatus.ERROR != state.getUploadStatus()) {
7          messagingInterface.enqueueTransferState( state.getDatabaseId());
8        }
```

Koska julkisten metodien sisällössä ei löydy suurempia eroja, siirytään vertailussa käymään OldMessageProcessor-luokkaa läpi metodien ilmenemisjärjestyksessä ja verrataan niitä re-

faktoroidussa toteutuksessa olevaan sisältöön, joka niitä toiminnallisesti vastaa.

Vaihe 8 kuviossa 1 laukaisee tämän prosessin, joka jatkuu kunnes operaatio lähettää prosessin 11. kohdassa viestin onnistumisesta.

5.2.5 encodePayload(TransferState) ja uploadFile(TransferState)

RefactoredMessageProcessor-luokan metodi encodePayload siirrettiin refaktoroinnin tuloksena ilman muutoksia JyuUploadableFileService-luokkaan, joka toteuttaa abstraktin luokan AbstractUploadableFileService. Abstrakti yläluokka toteutti lisäksi metodin uploadFile(TransferState).

Metodi uploadFile(TransferState) ennen refaktorointia:

OldMessageProcessor.java:encodePayload

```
1  private byte[] encodePayload( TransferState state ) {
2    byte[] encodedPayload = encodedPayloadService.load( state .getOwner(), state .getPayloadId () ) ;
3
4    if (encodedPayload == null) {
5      TransferStateTransition transition = new TransferStateTransition () ;
6      transition . setTransferStateId ( state .getDatabaseId () ) ;
7      transition . setTransitionReason ("Could not encode payload for state " +
           state .getDatabaseId () ) ;
8      transition . setOriginalStatus ( state .getUploadStatus () ) ;
9      transition .setNewStatus(UploadStatus.ERROR);
10     transferStateTransitionStore . create ( transition ) ;
11
12     throw new RuntimeException( transition . getTransitionReason () ) ;
13   }
14   return encodedPayload;
15 }
```

Metodi encodePayload(TransferState) ennen refaktorointia:

OldMessageProcessor.java:uploadFile

```
1  private void uploadFile ( TransferState state ) {
2    Payload payload = payloadService .load( state .getPayloadId () ) ;
```

```
3       Customer customer = customerService.load(payload.getOwner());
4       ThirdPartyAuthentication authentication =
                externalAuthenticationService.loadUserAuthentication(
5           customer.getId(), THIRD_PARTY_SERVICE_NAME);
6
7       try {
8         uploadService.uploadToThirdParty(state.getEncodedPayload(), authentication);
9       } catch (Exception e) {
10          TransferStateTransition transition = new TransferStateTransition();
11          transition.setNewStatus(UploadStatus.ERROR);
12          transition.setOriginalStatus(state.getUploadStatus());
13          transition.setTransferStateId(state.getDatabaseId());
14          transition.setTransitionReason("Error uploading to third party service for state " +
                state.getDatabaseId
15              ());
16
17          state.setUploadStatus(UploadStatus.ERROR);
18
19          transferStateStore.update(state);
20          transferStateTransitionStore.update(transition);
21      }
22  }
```

Abstrakti AbstractUploadableFileService refaktoroinnin jälkeen:

AbstractUploadableFileService.java

```
1   /**
2    * Load a translated {@link Payload} as a byte array, that is suitable for
3    * uploading to a specific third party service.
4    * @param state
5    * @return
6    */
7   public abstract byte[] encodePayload(TransferState state);
8
9   /**
10   * Upload an encoded {@link Payload} within a {@link TransferState} to
11   * a third party service.
12   * @param state
```

```
13      */
14     public void uploadFile( TransferState state ) {
15       Payload payload = payloadService.load( state.getPayloadId() );
16       Customer customer = customerService.load(payload.getOwner());
17        ThirdPartyAuthentication authentication =
               externalAuthenticationService .loadUserAuthentication (
18         customer.getId(), config.getThirdPartyServiceName());
19
20       try {
21         uploadService.uploadToThirdParty( state.getEncodedPayload(), authentication );
22       } catch (Exception e) {
23          TransferStateTransition transition = new TransferStateTransition ();
24         transition.setNewStatus(UploadStatus.ERROR);
25         transition. setOriginalStatus ( state.getUploadStatus() );
26         transition. setTransferStateId ( state.getDatabaseId() );
27         transition.setTransitionReason ("Error uploading to third party service for state " +
               state.getDatabaseId() );
28
29         state.setUploadStatus (UploadStatus.ERROR);
30
31          transferStateStore .update( state );
32          transferStateTransitionStore .update( transition );
33       }
34     }
```

AbstractUploadableFileServiceä perivän luokan encodePayload-toteutus refaktoroinnin jälkeen:

JyuUploadableFileService.java

```
1      public byte[] encodePayload( TransferState state ) {
2        byte[] encodedPayload = encodedPayloadService.load( state.getOwner(), state.getPayloadId() );
3
4        if (encodedPayload == null) {
5          TransferStateTransition transition = new TransferStateTransition ();
6          transition. setTransferStateId ( state.getDatabaseId() );
7          transition.setTransitionReason ("Could not encode payload for state " +
               state.getDatabaseId() );
8          transition. setOriginalStatus ( state.getUploadStatus() );
```

```
9           transition .setNewStatus(UploadStatus.ERROR);
10          transferStateTransitionStore . create ( transition );
11
12        throw new RuntimeException( transition . getTransitionReason () );
13      }
14      return encodedPayload;
```

Tähän osioon tehdyt muutokset ovat huomattavan vähäisiä koodin mekaniikassa, mutta vastapainoisesti huomattavan laajoja koko ohjelman lähdekoodin rakenteessa.

Muutoksia itse koodin mekanikkaan on tullut vain riveillä OldMessageProcessor.uploadFile:5 ja AbstractUploadableFileService:18 näkyvä tapa ladata kolmannen osapuolen autentikaatiotieto. Vanha toteutus on kovakoodannut tunnisteen, kun refaktoroidussa toteutuksessa se injektoidaan konfiguraatio-olion avulla. Tämä mahdollistaa konfiguraatio-olion luomisen muualla ohjelmassa. Ideaalitilanteessa tämä tehtäisi yhdessä paikassa, joka vastaa ohjelmiston yleisestä konfiguraatiosta.

Rakenteellisia muutoksia taas on yleisellä tasolla kaksi. Näistä molemmat ovat huomattavan suuria, järjestäen huomattavia osia koodista uusiksi.

Ensimmäiseksi Kun katsellaan mitä AbstractUploadableFileService-luokan riveillä 14-21 tapahtuu, havaitaan että metodi käyttää kolmea injektoitua palvelua ja yhtä injektoitua konfiguraatio-oliota, abstrahoiden näiden väliset suhteet piiloon itse RefactoredMessageProcess-luokalta kyseisen prosessin suhteen.

Seuraava osa rakenteellisia muutoksia on ollut Dijkstran ideoiden (Dijkstra 1982) mukainen tapa siirtää AbstractUploadableFileService-luokan vastuulle vain se toiminnallisuus, joka liittyy ulkoiseen palveluun. AbstractUploadableFileService sisältää kaksi metodia: konkreettinen uploadFile data lähettämiseen, sekä abstrakti encodePayload datan enkoodaamiseen kolmannen osapuolen ymmärtämään muotoon. Koska tässä vaiheessa ei oltu vielä nähty tarvetta muuttaa itse uploadFile-metodin toteutusta, luokasta tehtiin abstrakti luokka, eikä rajapintaa. JyuUploadableFileServicen anonymisoitu toteutus ei valitettavasti sisällä mitään indikaatiota miksi abstraktiota edes tarvittaisi, mutta syynä tähän on koodin anonymisoitu luonne.

Vaikka rakenteellisesti koodi on sijoittunut huomattavissa määrin uudelleen kyseisen toiminnallisuuden refaktoroinnissa, ei muutos ole vaikuttanut MessageProcessor-rajapintaan mitenkään. Operaatio sisältää samat vaiheet 4-6 kuviosta 1 ennen refaktorointia, sekä sen jälkeen.

5.2.6 updateComponentState(TransferState) ja createNewTransition(UploadStatus, UploadStatus, TransferState)

Metodit ennen refaktorointia:

OldMessageProcessor.java:updateComponentState

```
1  private void updateComponentState( TransferState state ) {
2    switch ( state . getUploadStatus () ) {
3      case STARTED:
4        state . setUploadStatus (UploadStatus . UPLOADED);
5        this . createNewTransition (UploadStatus . STARTED, UploadStatus.UPLOADED, state);
6        break;
7      case UPLOADED:
8        state . setUploadStatus (UploadStatus . FINISHED);
9        this . createNewTransition (UploadStatus . UPLOADED, UploadStatus.FINISHED, state);
10       break;
11     default :
12       break;
13   }
14
15 }
```

OldMessageProcessor.java:createNewTransition

```
1  private void createNewTransition (UploadStatus originalStatus , UploadStatus newStatus,
       TransferState state ) {
2    TransferStateTransition transition = new TransferStateTransition ();
3    transition . setOriginalStatus ( originalStatus );
4    transition .setNewStatus(newStatus);
5    transition . setTransferStateId ( state . getDatabaseId () );
6    transferStateTransitionStore . create ( transition );
7  }
```

ja refaktoroinnin jälkeen:

AbstractUploadableFileService.java

```
1   public void updateComponentState(TransferState state) {
2     switch (state.getUploadStatus()) {
3       case STARTED:
4         state.setUploadStatus(UploadStatus.UPLOADED);
5         this.createNewTransition(UploadStatus.STARTED, UploadStatus.UPLOADED, state);
6         break;
7       case UPLOADED:
8         state.setUploadStatus(UploadStatus.FINISHED);
9         this.createNewTransition(UploadStatus.UPLOADED, UploadStatus.FINISHED, state);
10        break;
11      default:
12        break;
13    }
14  }
15
16  private void createNewTransition(UploadStatus originalStatus, UploadStatus newStatus,
        TransferState state) {
17    TransferStateTransition transition = new TransferStateTransition();
18    transition.setOriginalStatus(originalStatus);
19    transition.setNewStatus(newStatus);
20    transition.setTransferStateId(state.getDatabaseId());
21    transferStateTransitionStore.create(transition);
22  }
23 }
```

Koodin ulkoasu ei ole muuttunut, mutta itse koodi on siirtynyt apuluokkaan, joka kapseloi updateComponentState()-metodin, sekä sen käyttämän yksityisen apumetodin createNewTransition. Tämä on omiaan selkeyttämään koodia, sillä refaktoroimattomassa toteutuksessa ei ollut selvää minkä operaatioiden haluttiin kutsuvan createNewTransition()-metodia. Nyt tilasiirtymien luominen on kapseloitu osaksi TransferState-olion muutosta, joka antaa jonkinlaisen kontekstin, ollen osana AbstractUploadableFileServiceä.

Tämä on hyvä esimerkki siitä, mitä GoF (Gamma ym. 1995) ja Martin (Martin 2003) tarkoittavat yhden vastuun periaatteesta ja kytkentöjen vähentämisestä kirjoittaessaan. Pelkästään

yhden paikan tarvitsee tietää milloin createNewTransition()-metodia kutsutaan, eikä potentiaalisten muutosten tarvitse koskea muihin ohjelman osioihin.

On fakta että ratkaisu, jossa komponentin tilaa muutetaan yhdessä metodissa ja muuttunut tila tallennetaan toisessa metodissa, ei ole optimaalisin kummankaan periaatteen näkökulmasta. Mutta koska liiketoiminnalliset syyt preferoivat koodin uudelleenjärjestelyä uudelleenkirjoituksen sijaan, koodin absoluuttisen laadun analysointia ei tässä tilanteessa suoriteta.

Operaatioparia kutsutaan useasta kohdasta vaihtelevissa tiloissa olevilla parametreillä. Kuviossa 1 sitä kutsutaan seuraavissa kohdissa:

1. Vaiheiden 6 ja 7 välissä.
2. Vaiheen 10 jälkeen.

5.2.7 saveComponent(TransferState)

OldMessageProcessor.java

```
1  private void saveComponent(TransferState state ) {
2      transferStateStore .update( state );
3  }
```

Injektoidun transferStateStore-palvelun kutsu oli ennen refaktorointia omassa metodissaan virheenkäsittelyn takia. Refaktoroinnin seurauksena virheenkäsittely siirrettiin osaksi transferStateStore-j jolloin kutsu voitiin tehdä ilman erillistä virheenkäsitttelymekaniikkaa. Tämä on selkein paikka, jossa koodin anonymisointi hankaloittaa luettavuutta. Mutta tässäkin metodissa oli vain kaksi lopputulosta: onnistunut poistuminen, tai poikkeuksen heitto.

Koska kuviot operaatiojärjestyksestä pohjautuvat refaktorointiin, ei kuvioissa ole näkyvillä tämän operaation kohtaa. Mekanismi on vain korvaantunut edellisessä alikappeleessa esiintyvällä toiminnallisuudella.

5.2.8 checkUploadStatus(TransferState)

Tämäkin metodi on siirretty luokkaan, joka vastaa kolmannen osapuolen palveluiden kanssa keskustelusta. Itse koodissa ei ole muuta kuin ulkopuolelta injektoitava tapa tuoda tieto mistä

kolmannen osapuolen palvelusta on kyse.

OldMessageProcessor.java

```
1  private void checkUploadStatus( TransferState state ) {
2    Payload payload = payloadService.load( state.getPayloadId() );
3    Customer customer = customerService.load(payload.getOwner());
4     ThirdPartyAuthentication authentication =
          externalAuthenticationService . loadUserAuthentication (
5        customer.getId(), THIRD_PARTY_SERVICE_NAME);
6
7    try {
8      uploadService.checkUploadStatus(payload.getId(), authentication );
9    } catch (Exception e) {
10      TransferStateTransition transition = new TransferStateTransition ();
11      transition .setNewStatus(UploadStatus.ERROR);
12      transition . setOriginalStatus ( state.getUploadStatus() );
13      transition . setTransferStateId ( state.getDatabaseId() );
14      transition . setTransitionReason ("Error querying third party service for state " +
            state.getDatabaseId() );
15
16      state . setUploadStatus (UploadStatus.ERROR);
17
18      transferStateStore .update( state );
19      transferStateTransitionStore .update( transition );
20   }
21 }
```

AbstractUploadableFileService.java

```
1  public void checkUploadStatus( TransferState state ) {
2    Payload payload = payloadService.load( state.getPayloadId() );
3    Customer customer = customerService.load(payload.getOwner());
4     ThirdPartyAuthentication authentication =
          externalAuthenticationService . loadUserAuthentication (
5      customer.getId(), config.getThirdPartyServiceName());
6
7    try {
8      uploadService.checkUploadStatus(payload.getId(), authentication );
9    } catch (Exception e) {
```

```
10          TransferStateTransition transition = new TransferStateTransition();
11          transition.setNewStatus(UploadStatus.ERROR);
12          transition.setOriginalStatus(state.getUploadStatus());
13          transition.setTransferStateId(state.getDatabaseId());
14          transition.setTransitionReason("Error querying third party service for state " +
                state.getDatabaseId());
15
16          state.setUploadStatus(UploadStatus.ERROR);
17
18          transferStateStore.update(state);
19          transferStateTransitionStore.update(transition);
20      }
21  }
```

Operaatio tapahtuu refaktoroidun version tapauksessa kohdassa 9 ohjelmiston yleisestä toiminnallisuuskaaviosta 1.

Operaatiot näyttävät huomattavan identtisiltä koska, lukuunottamatta luokkaa jossa ne sijaitsevat, ne ovat. Ennen refaktorointia OldMessageProcessor-instanssiin injektoitiin eri yhteistyökumppanien tarvitsemat toteutukset. AbstractUploadableFileService-luokkaa perivät luokkainstanssit taas olivat injektoinnin kohteena refaktoroinnin jälkeen.

6 Evaluointi

6.1 Refaktoroinnin arviointi

Kaikki kappaleessa 2.2 esitetyt menetelmät olivat käytössä, tai päämääränä, refaktoroinnissa. Mutta hyödyllisiä työkaluja olio-ohjelmoinnissa on monia muitakin. Esimerkiksi Robert Martinin jo pitkään tunnettu SOLID-periaate (Martin 2005), jonka esittelemistä menetelmistä osaa onkin hyödynnetty refaktoroinnin yhteydessä. Koska kyseessä kuitenkin oli refaktorointi, joka tähtäsi olemassaolevan ohjelman modularisointiin, eikä sen kokonaisvaltaiseen korjaamiseen, muutokset noudattavat hyvin pitkälti samaa kaavaa.

Yleisellä tasolla refaktorointi voidaan kuvata seuraavasti:

1. Tunnista yleistettävät osat OldMessageProcess-luokasta.
2. Luo rajapinnat tai abstraktiot em. osista.
3. Kokoa Dependency Injection-työkaluilla ohjelmasta toimiva kokonaisuus.

Viimeistä kohtaa ei käsitellä tässä tutkielmassa, koska se on toteutusriippuvainen asia, eikä vaikuta ohjelmiston rakenteeseen, kuin injektiokirjastojen toiminnallisten vaatimusten suhteen.

Muutoksia yleisesti tarkasteltaessa, voidaan nähdä että refaktorointi siirtää MessageProcessor-toteutuksen toiminnallisuuden aliluokkiinsa. Pelkästään tilakone, toimintasekvenssin järjestys, sekä virheenkäsittely ovat jääneet refaktoroidun toteutuksen vastuulle. Valitettavasti sovellukseen vaikuttavat liiketoiminnalliset vaatimukset eivät sallineet kokonaisvaltaisempaa refaktorointia ja vastuun delegoimista mielestäni sopivimmille komponenteille.

Suurin ongelma, joka tämän refaktoroinnin jälkeen ohjelmistoon jäi, oli TransferState-olioiden käsittelyn vastuun epäselkeys. Tämä johti ehkä selkeimpään Yhden Vastuun Periaatteen rikkomiseen refaktoroinnin jälkeisessä ohjelmassa.

Alkuperäinen ohjelmakoodi siis muunnettiin sinällään yksinkertaisin toimenpitein yleistettäväksi käyttäen tilanteen mukaan vastuualueiden erottelun sekä yhden vastuun periaatteiden sekoitusta. Kumpaakaan näistä periaateista ei voitu noudattaa tiukasti, vaan käytetiin ohje-

nuorina, joilla irrotettiin yhdestä valtavan monta asiaa käsittelevästä luokasta helposti käsiteltäviä kokonaisuuksia. Tämä lähestymistapa auttoi abstrahoimaan monimutkaisen luokan toiminnallisuutta olioinstansseihin, jolloin eri instanssien kutsut auttoivat luomaan käsitteen kontekstista. Ennen refaktorointia koodissa oli “this.uploadfile()”- ja “this.checkUploadStatus()” tyylisiä metodikutsuja, refaktoroinnin jälkeen ne olivat “uploadableFileService.uploadFile()” ja “uploadableFileService.checkUploadStatus()”-tyyppisiä. Tämä erotus liittää jonkinasteisen kontekstin kutsuihin, kertoen koodin lukijalle että kyseiset kutsut todennäköisesti liittyvät etäjärjestelmään. Tämän lisäkontekstin hyödyt eivät kuitenkaan ole jatkossa itsestäänselvyys, vaan jo huono nimeäminen voi pilata koko idean.

Koska ohjelmisto käytti alusta alkaen Dependency Injection-työkaluja riippuvuuksien kokoamiseen, pystyttiin refaktoroinnin yhteydessä laajentamaan ohjelmistoa ja konfiguroimaan samasta lähdekoodista erilailla toimivia ohjelmistoja.

Voidaankin siis nähdä, että alkuperäinen tavoite yleistää ohjelmistoa sen verran, että siitä saataisi useamman kolmannen osapuolen järjestelmän kanssa yhteensopiva, on toteutunut.

Myöskin henkilökohtaisen kokemukseni pohjalta, arvioisin refaktoroinnin toteuttaneen toivotut liiketoimintavaatimukset, mutta liiketoiminnan halu pitää toiminnallisuus mahdollisimman muuttumattomana oli ongelmallinen. Kyseinen liiketoimintapäätös hankaloitti mielestäni selkeiden kokonaisuuksien määrittelemistä.

Pahimpana esimerkkinä tämän päätöksen aiheuttamista ongelmista pitäisin TransferStateServicen olemassaoloa itsenäisenä entiteettinä, eikä osana TransferStateStorea. Nyt ensimmäinen vastaa yksittäisten siirtotapahtumien kirjaamisesta, kun toinen seuraa Payload-tietueen siirron tilaa useiden siirtotapahtumien yli. Tämä taas tekee suorastaan tuskaisan hankalaksi toteuttaa atomisia transaktioita jotka varmistaisivat molempien tietueiden päivityksen toimivan “kaikki tai ei mitään”-periaatteella.

Mielestäni tämä ongelma on malliesimerkki tilanteesta, jossa lähtökohtaisesti huonoa koodia ei haluta korjata, koska kaikki isot muutokset edes jotenkin toimiviin kohtiin nähtiin huomattavan riskialttiina operaatioina. Näissä tapauksissa ongelma mielestäni juontaa joko projektin aikana, tai sitä ennen opitusta epävarmuudesta toteutuksen laatua ja regressiohallinan varmuutta kohtaan. Johtuuko tällainen asenne yleisesti ohjelmoinnin laadusta organisaatios-

sa, organisaation ongelmista, vai johdon asenteesta, riippuu täysin ympäristöstä. Laajemman skaalan analyysit asiasta voisivat kuitenkin osoittaa suurimpia johdon luoton puutteeseen vaikuttavia tekijöitä, joita purkamalla tiimit mahdollisesti kykenisivät rakentamaan puuttuvaa luottoa. Tällaisten asioiden tunnistamiseen ja niiden ratkaisemiseen suuntaava lisätutkimus olisi mielestäni hyödyllistä sekä yksittäisissä organisaatioissa, että yleisellä tasolla.

6.2 Toteutusaika

Projektin raportissa (liite A) esiintyy kolme perättäistä vaihetta:

- Alkuperäinen totetus: 20 kalenteriviikkoa neljällä kehittäjällä, eli 80 henkilötyöviikkoa. Vaiheessa toteutettiin ja asennettiin sovellus, siten että se kommunikoi kumppanijärjestelmän kanssa.
- Refaktorointi: työtä 28:na kalenteripäivänä. Vaiheessa yleistettiin sovelluksen rakenne uudelleenkäytettäväksi.
- Uudet toteutukset: 24 kalenteriviikkoa kahdella kehittäjällä, poislukien lomat. Yhteensä 46 henkilötyöviikkoa. Tässä ajassa saatiin kolme uutta sovellusta kommunikoimaan kumppanijärjestelmien kanssa, aloitettiin neljäs sovellus, sekä laajennettiin sovelluksen yleistä osaa tukemaan useita tietomalleja.

Aikamääreet eivät ole täysin identtisesti muodostettuja anonymisoinnin ja projektien seurannan kehittymisen takia, mutta kolmen aikamääreen huomattavien erojen takia koen arvioinnin tarkkuuden olevan riittävällä tasolla.

Toisena ongelmana aikamääreissä on NDA-ympäristössä suoritettavan analyysin hankaluus, jonka takia toteutusvaiheisiin on valittu skaalaksi viikot. Tämän skaalan toivon toimivan jokseenkin tasoittavana tekijänä työmäärien hetkittäisiin heilahteluihin.

Jos kaikki luvut muutetaan henkilötyöpäiviksi, saadaan seuraavat luvut:

- Alkuperäinen totetus: 20 vko * 4 hlö * 5 pv = 400 htp.
- Refaktorointi: 28 htp.
- Uudet toteutukset: 46 vko * 5 pv = 230 htp.

Pikainen laskutoimitus kertoo, että lisäämällä 28/400 = 7% työtä alkuperäiseen toteutukseen, saatiin sovelluksesta yleistettävä. Toinen laskutoimitus kertoo, että 230/400=0,575. Eli 7% + 57.5% = 64.5% lisätyötä riitti monistamaan yhden sovelluksen neljäksi, sekä parantamaan sovellusalustaa huomattavasti.

Pelkästään tällä luvulla voidaan katsoa että yleistäminen oli kannattavaa, mutta kuten raportissa mainitaan, uudet toteutukset tehtiin projektia aluksi tuntemattomien henkilöiden toimesta, sisälsivät tietomallituen laajentamisen, sekä uusia kommunikaatio-ongelmia. Uusien toteutusten ongelmia on tarkemmin purettu auki projektiraportissa ja mielestäni ne toimivat selkeinä lisäindikaattoreina refaktoroinnin hyödyllisyydestä. Raportissa mainitut ongelmat nimittäin ovat mielestäni olleet kuitenkin huomattavia.

Suurin ongelma ajankäytön analysoinnissa on kuitenkin koodin anonymisaatio. Koska uusien toteutusten implementointiin käyttämää aikaa ei ole voitu jakaa osiin eri rajapintojen tai abstraktioiden vaatimien uusien implementaatioiden suhteen, näistä aikaavievimpien asioiden erottelu on mahdotonta. On vain yksi könttäsumma joka todistaa ohjelmiston totaalisen uudelleentoteutuksen olleen todennäköisesti kannattamaton idea, mutta joka ei tarkemmin kerro miten uudelleenimplementaation vaatima aika jakaantui.

7 Yhteenveto

Tutkielmassa läpikäytiin Data Acquisition-palvelua toteuttavaa ohjelmistoa, sekä sen refaktorointia jonka tarkoitus oli yleistää ohelmisto yhden yhteistyökumppanin infrastrukturiin yhteensopivasta muotoon joka tarjoaa samaa palvelua useammille kumppaneille. Tutkimuksen kohteena oli ohjelman lähdekoodi ennen ja jälkeen kyseisen refaktoroinnin. Näitä artefakteja vertailtiin konstruktiivisesti, tarkoituksena tunnistaa refaktoroinnissa hyödynnettyjä ohjelmistoteollisuuden tekniikoita, sekä kuvata miten niitä hyödynnettin. Tämän lisäksi tarkasteltiin suppeasti refaktoroinnin vaikutuksia ohjelmiston jatkokehityksen nopeuteen.

Analyysi löysi viisi selkeää suunnitteluteknistä ideaa ja/tai käsitettä, joita käyttämällä tai olemassaolevia kohtia jatkojalostamalla ohjelmakoodi oli refaktoroitu uudelleenkäytettäväksi ja toimivammaksi.

Koska kaikki refaktoroinnissa hyödynnetyt ideat ovat vähintään 15 vuotta vanhoja ja ovat huomattavan hyödyllisiä myös toiminnallisuuden yleistämisen ulkopuolella, herää jatkotutkimuksen ajatellen parikin kysymystä:

- Kuinka monessa ohjelmistossa vastaavanlaisella yleistämisellä voitaisiin saavuttaa huomattavasti selkeämpää ohjelmakoodia?
- Kuinka paljon uusia ohjelmistoprojekteja totetutetaan, joissa kyseisiä ideoita hyödyntämällä ohjelmisto selkeytyisi? Ja miksi näissä ei hyödynnetä kyseisiä ideoita?

Ajankäytön puolesta huomattiin että yleistäminen oli kyseisen ohjelmiston tapauksessa kannattava vaihtoehto, kun arvioidaan jatkokehitykseen käytettyä aikaa. Useita vaihtoehtoisia sovelluskonfiguraatioita voitiin toteuttaa samasta pohjasta huomattavalla ajansäästöllä. Ohjelmisto myös parani koodin luettavuuden kannalta, sekä selkeytyi rakenteeltaan.

Mielestäni tämän ohjelmiston suhteen saatu ajankäytöllinen säästö ei ole kuin heikosti suuntaa-antava indikaatio tämäntyyppisten refaktorointien hyödyistä ohjelmistoille yleensä. Jotta ohjelmistojen yleistäminen refaktoroinnilla olisi kannattavaa millään mittarilla, tulisi potentiaalisen uuden ohjelman ratkaiseman ongelman olla sen verran yhtenäinen alkuperäisen ohjelman ratkaiseman ongelman kanssa, että yleistäminen voitaisi selkeästi toteuttaa olemassao-

levalla logiikalla. Mielestäni tämä vaatisi tavan kuvata ohjelmistoja huomattavasti paremmin kuin mihin nykyiset korkean tason ohjelmointikielet kykenevät.

Lähteet

Andrews, Gregory R., ja Fred B. Schneider. 1983. “Concepts and Notations for Concurrent Programming”. *ACM Comput. Surv.* (New York, NY, USA) 15, numero 1 (maaliskuu): 3–43. ISSN: 0360-0300. doi:10.1145/356901.356903.

Bock, David. 2003. “The Paperboy, The Wallet, and The Law Of Demeter”. Viitattu 4. lokakuuta 2017. https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf.

Crnkovic, Gordana Dodig. 2010. “Constructive Research and Info-computational Knowledge Generation”. Teoksessa *Model-Based Reasoning in Science and Technology: Abduction, Logic, and Computational Discovery,* toimittanut Lorenzo Magnani ja Claudio Carnielli Walterand Pizzi, 359–380. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-15223-8. doi:10.1007/978-3-642-15223-8_20.

Dijkstra, Edsger W. 1972. “The Humble Programmer”. Viitattu 29. tammikuuta 2017. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD340.html.

———. 1982. “On the Role of Scientific Thought”. Teoksessa *Selected Writings on Computing: A personal Perspective,* 60–66. New York, NY: Springer New York. ISBN: 978-1-4612-5695-3. doi:10.1007/978-1-4612-5695-3_12.

Fowler, Martin. 2004. “Inversion of Control Containers and the Dependency Injection pattern”. Viitattu 18. syyskuuta 2016. http://martinfowler.com/articles/injection.html.

Gamma, Erich, Richard Helm, Ralph Johnson ja John Vlissides. 1995. “Design Patterns”. Addison-Wesley. ISBN: 0201633612.

“How does a Queue compare to a Topic”. 2011. Viitattu 11. heinäkuuta 2017. http://activemq.apache.org/how-does-a-queue-compare-to-a-topic.html.

"JMS Load Balancing (Using the JMS Binding Component)". 2010. Viitattu 25. huhtikuuta 2018. https://docs.oracle.com/cd/E19182-01/820-7853/ghhhx/index.html.

Lawton, Brian, ja Don Awalt. 1999. "Using DTS to Populate a Data Warehouse". Viitattu 4. huhtikuuta 2018. http://www.itprotoday.com/business-intelligence/using-dts-populate-data-warehouse.

Mackinnon, Tim, Steven Freeman ja Philip Craig. 2001. "Endo-Testing: Unit Testing with Mock Objects". Joulukuu. Viitattu 14. helmikuuta 2018. https://www.researchgate.net/publication/2395276_Endo-Testing_Unit_Testing_with_Mock_Objects.

MacLeod, Stewart P., ja Casey L. Kieman. 2002. "Method and apparatus for import, transform and export of data". Viitattu 4. huhtikuuta 2018. https://patents.google.com/patent/US6356901B1/en.

March, Salvatore, ja Gerald Smith. 1995. "Design and Natural Science Research on Information Technology", 15 (joulukuu): 251–266.

Martin, Robert C. 2003. *Agile Software Development: Principles, Patterns, and Practices.* Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0135974445.

———. 2005. "The Principles of OOD". Viitattu 22. marraskuuta 2017. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

———. 2014. "The Single Responsibility Principle". Viitattu 22. syyskuuta 2016. https://8thlight.com/blog/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html.

Nidhra, Srinivas. 2012. "Black Box and White Box Testing Techniques - A Literature Review", 2 (kesäkuu): 29–50.

Parnas, David L. 1972. "On the Criteria to Be Used in Decomposing Systems into Modules". *Commun. ACM* (New York, NY, USA) 15, numero 12 (joulukuu): 1053–1058. ISSN: 0001-0782. doi:10.1145/361598.361623.

"RabbitMQ - Time-To-Live Extensions". 2017. Viitattu 11. heinäkuuta. https://www.rabbitmq.com/ttl.html.

Shaw, Zed A. 2017. "Programming, Motherfucker". Viitattu 10. tammikuuta 2018. http://programming-motherfucker.com/.

Shore, James. 2006. "Dependency Injection Demystified". Viitattu 12. maaliskuuta 2017. http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html.

Torchiano, Marco, Maurizio Morisio ja Hakan Erdogmus. 2005. "On the Effectiveness of the Test-First Approach to Programming". *IEEE Transactions on Software Engineering* 31 (maaliskuu): 226–237. ISSN: 0098-5589. doi:10.1109/TSE.2005.37.

West, Matthew. 2010. *Developing High Quality Data Models.* Joulukuu. ISBN: 9780123751072.

Liitteet

A Projektiraportti

Tutkimus tarkastelee MessageProcessor-luokan yleistämisen mahdollistamaa ajansäästöä, jonka katsotaan olevan valmis, kun data siirtyy palvelusta kumppanille. Tämä määritelmä tehtiin, koska palvelujen tuotantoonvientiin ja aktivointiin liittyy kehitysryhmästä riippumattomia liiketoimintapäätöksiä niin organisaation, kuin kumppanienkin tahoilta. Myöskin Payload-tietomallien konversioon liittyviä vaatimuksia ja ongelmia voitiin tarkastella kunnolla, vasta kun dataa voitiin siirtää. Tämä on toinen syy miksi konversion hienosäätöön kuluva aika on rajattu pois lisätoteutusten kestosta, toisen ollessa kummassakin MessageProcessor-implementaatiossa identtisenä säilynyt Separation Of Concerns-periaatetta totteleva rajapinta, joka hoitaa Payload -> byte[]-konversion.

Alkuperäisen implementaation toteutus kesti arviolta 20 kalenteriviikkoa, eli 80 henkilötyöviikkoa. Toteutuksessa oli mukana neljä kehittäjää.

Alkuperäisen implementaation julkaisun jälkeen aloitettiin ratkaisun refaktorointi useammille kumppaneille yleistettäväksi. Yleistävän refaktoroinnin suunnitteluun ja toteutuksen ajankäytön estimaatin yläraja on 28 henkilötyöpäivää. Summa laskettiin listaamalla projektinhallintatyökalusta kaikki refaktorointiin kuuluvat tehtävät ja yhteenlaskemalla niiden koodimuutosten uniikit päivät per tekijä.

Refaktoroinnin jälkeiset implementaatiot toteutettiin kahden kehittäjän toimesta. Kyseiset kehittäjät eivät osallistuneet projektiin alkuperäisen toteutuksen aikana.

Lisätoteutukset suoritettiin limittäin, kehittäjien priorisoidessa tehtäviä Scrumm Masterin kautta kulkevien tavoitteiden mukaisesti. Tämä nopeutti kehitystyön kokonaisaikataulua, sillä kehittäjien työtä estävät, lisäselvityksiä tai kommunikaatiota vaativat, asiat voitiin hoitaa muiden kuin kehittäjien toimesta. Vastapainoisesti, testiympäristöjen ongelmat taas hidastivat kaikkia käynnissä olevia lisätoteutuksia.

Kolme lisätoteutusta (L1, L2, L3) viikkotasolla:

- Viikko 1. L1 alkaa.
- Viikko 10. L2 alkaa.
- Viikko 11. L1 data siirtyy.
- Viikko 14. L3 alkaa.
- Viikko 16. Yksi kehittäjä lomalla.
- Viikko 17. Yksi kehittäjä lomalla.
- Viikko 20. L2 data siirtyy.
- Viikko 24. L3 data siirtyy.

Näiden kolmen lisätoteutuksen ohessa neljäs lisätoteutus aloitettiin suoritusviikolla 20. mutta sitä ei raporttia kirjoitettaessa, viikolla 25, ole saatu valmiiksi.

Yhteensä kolme lisätoteutusta siis veivät 24 kalenteriviikkoa kahdelta projektia ennen näkemättömältä kehittäjältä. Kokonaisaikaa siis kului 46 henkilötyöviikkoa, kun huomioidaan lomat.

Lisätoteutus 2:n ylimääräisenä hidastavana tekijänä oli yksi toteutus väärää rajapintaa vasten. Virheen syynä oli kommunikaatio-ongelmat kumppanin ja organisaation kanssa.

Lisätoteutus 3:n ylimääräisenä aikaavievänä tekijänä oli refaktoroidun toteutuksen lisärefaktorointi, joka lisäsi prosessointiin tuen useille Payload-tyypeille samassa ohjelmassa. Arkkitehdin arvio kyseisen lisätoteutuksen ajankäytöstä oli:

- 30% kumppani-integraatioon.
- 30% useiden Payloadien tukemista vaativaan refaktorointiin.
- 40% muihin töihin.

Projektin Scrumm Masterin arvion mukaan kehittäjät käyttivät 40%-50% ajastaan toteutukseen. Loput ajasta kului määrittelyn tarkentamiseen, testiympäristöjen säätämiseen, sekä erinäisiin selvittelyihin organisaation ja kumppaneiden kanssa.

Organisaation osa, jolla oli vaikutusta lisätoteutuksiin, tai joidenka työhön lisätoteutusten implementaatio ja tuote-, sekä palvelulukokonaisuuteen integrointi vaikutti, on suurudeltaan noin 100-120 henkilöä.

B Alkuperäinen lähdekoodi

Listauksessa on vanhan toteutuksen ainoa luokka jota jouduttiin rakenteellisesti muuttamaan. Tämä luokka löytyy Maven-moduulista uploader-implementations.

OldMessageProcessor.java

```
1  package fi.jyu.jutarant.old.service;
2
3  import fi.jyu.jutarant.general.messaging.MessagingInterface;
4  import fi.jyu.jutarant.general.model.Customer;
5  import fi.jyu.jutarant.general.model.EntityType;
6  import fi.jyu.jutarant.general.model.Payload;
7  import fi.jyu.jutarant.general.model.ThirdPartyAuthentication;
8  import fi.jyu.jutarant.general.model.TransferState;
9  import fi.jyu.jutarant.general.model.TransferStateTransition;
10 import fi.jyu.jutarant.general.model.UploadStatus;
11 import fi.jyu.jutarant.general.service.CustomerService;
12 import fi.jyu.jutarant.general.service.EncodedPayloadService;
13 import fi.jyu.jutarant.general.service.ExternalAuthorizationService;
14 import fi.jyu.jutarant.general.service.PayloadService;
15 import fi.jyu.jutarant.general.serviceinterface.MessageProcessor;
16 import fi.jyu.jutarant.general.thirdpartyspecific.TransferStateStore;
17 import fi.jyu.jutarant.general.thirdpartyspecific.TransferStateTransitionStore;
18 import fi.jyu.jutarant.general.thirdpartyspecific.UploadService;
19
20 /**
21  * Process a {@link Payload} and upload it to JYU.
22  */
23 public class OldMessageProcessor implements MessageProcessor {
24
25   private static final String THIRD_PARTY_SERVICE_NAME = "JYU";
26   private PayloadService payloadService;
27   private EncodedPayloadService encodedPayloadService;
28   private CustomerService customerService;
29   private ExternalAuthorizationService externalAuthenticationService;
30   private TransferStateStore transferStateStore;
31   private TransferStateTransitionStore transferStateTransitionStore;
32   private MessagingInterface messagingInterface;
```

```
33    private UploadService uploadService;
34
35    public void ignite(int payloadId) {
36      Payload payload = payloadService.load(payloadId);
37      Customer customer = customerService.load(payload.getOwner());
38       ThirdPartyAuthentication authentication =
              externalAuthenticationService.loadUserAuthentication(
39            customer.getId(), THIRD_PARTY_SERVICE_NAME);
40
41       if (authentication != null) {
42          TransferState transferState = this.createNewTransferState(payloadId, payload.getOwner(),
                EntityType.ORDER);
43
44          messagingInterface.enqueueTransferState(transferState.getDatabaseId());
45       }
46    }
47
48    public void process(int transferStateId) {
49       TransferState state = transferStateStore.load(transferStateId);
50      UploadStatus status = state.getUploadStatus();
51
52      switch (status) {
53        case STARTED:
54          byte[] encodedPayload = this.encodePayload(state);
55           state.setEncodedPayload(encodedPayload);
56           this.uploadFile(state);
57           this.updateComponentState(state);
58           this.saveComponent(state);
59
60           if (UploadStatus.ERROR != state.getUploadStatus()) {
61             messagingInterface.enqueueTransferState(state.getDatabaseId());
62           }
63          break;
64        case UPLOADED:
65           this.checkUploadStatus(state);
66           this.updateComponentState(state);
67           this.saveComponent(state);
```

```
68
69              if (UploadStatus.ERROR != state.getUploadStatus()) {
70                  messagingInterface.enqueueTransferState(state.getDatabaseId());
71              }
72          case FINISHED:
73          case ERROR:
74      }
75  }
76
77  private byte[] encodePayload(TransferState state) {
78      byte[] encodedPayload = encodedPayloadService.load(state.getOwner(), state.getPayloadId());
79
80      if (encodedPayload == null) {
81          TransferStateTransition transition = new TransferStateTransition();
82          transition.setTransferStateId(state.getDatabaseId());
83          transition.setTransitionReason("Could not encode payload for state " +
                state.getDatabaseId());
84          transition.setOriginalStatus(state.getUploadStatus());
85          transition.setNewStatus(UploadStatus.ERROR);
86          transferStateTransitionStore.create(transition);
87
88          throw new RuntimeException(transition.getTransitionReason());
89      }
90      return encodedPayload;
91  }
92
93  private TransferState createNewTransferState(int payloadId, int owner, EntityType order) {
94      TransferState state = new TransferState();
95      state.setEntityId(payloadId);
96      state.setOwner(owner);
97      state.setUploadStatus(UploadStatus.STARTED);
98      state.setEntityType(order);
99      return transferStateStore.create(state);
100 }
101
102 private void uploadFile(TransferState state) {
103     Payload payload = payloadService.load(state.getPayloadId());
```

```
104     Customer customer = customerService.load(payload.getOwner());
105      ThirdPartyAuthentication authentication =
             externalAuthenticationService.loadUserAuthentication(
106          customer.getId(), THIRD_PARTY_SERVICE_NAME);
107
108     try {
109       uploadService.uploadToThirdParty(state.getEncodedPayload(), authentication);
110     } catch (Exception e) {
111       TransferStateTransition transition = new TransferStateTransition();
112       transition.setNewStatus(UploadStatus.ERROR);
113       transition.setOriginalStatus(state.getUploadStatus());
114       transition.setTransferStateId(state.getDatabaseId());
115       transition.setTransitionReason("Error uploading to third party service for state " +
              state.getDatabaseId
116            ());
117
118       state.setUploadStatus(UploadStatus.ERROR);
119
120       transferStateStore.update(state);
121       transferStateTransitionStore.update(transition);
122     }
123   }
124
125   private void updateComponentState(TransferState state) {
126     switch (state.getUploadStatus()) {
127       case STARTED:
128         state.setUploadStatus(UploadStatus.UPLOADED);
129         this.createNewTransition(UploadStatus.STARTED, UploadStatus.UPLOADED, state);
130         break;
131       case UPLOADED:
132         state.setUploadStatus(UploadStatus.FINISHED);
133         this.createNewTransition(UploadStatus.UPLOADED, UploadStatus.FINISHED, state);
134         break;
135       default:
136         break;
137     }
138
```

```
139     }
140
141     private void saveComponent(TransferState state) {
142         transferStateStore.update(state);
143     }
144
145     private void createNewTransition(UploadStatus originalStatus, UploadStatus newStatus,
            TransferState state) {
146         TransferStateTransition transition = new TransferStateTransition();
147         transition.setOriginalStatus(originalStatus);
148         transition.setNewStatus(newStatus);
149         transition.setTransferStateId(state.getDatabaseId());
150         transferStateTransitionStore.create(transition);
151     }
152
153     private void checkUploadStatus(TransferState state) {
154       Payload payload = payloadService.load(state.getPayloadId());
155       Customer customer = customerService.load(payload.getOwner());
156        ThirdPartyAuthentication authentication =
              externalAuthenticationService.loadUserAuthentication(
157           customer.getId(), THIRD_PARTY_SERVICE_NAME);
158
159       try {
160         uploadService.checkUploadStatus(payload.getId(), authentication);
161       } catch (Exception e) {
162           TransferStateTransition transition = new TransferStateTransition();
163           transition.setNewStatus(UploadStatus.ERROR);
164           transition.setOriginalStatus(state.getUploadStatus());
165           transition.setTransferStateId(state.getDatabaseId());
166           transition.setTransitionReason("Error querying third party service for state " +
                state.getDatabaseId());
167
168         state.setUploadStatus(UploadStatus.ERROR);
169
170           transferStateStore.update(state);
171           transferStateTransitionStore.update(transition);
172       }
```

```
173     }
174 }
```

C Refaktoroitu lähdekoodi

Listauksessa ovat ne luokat, joita replikoivat OldMessageProcessor-luokan toteutuksen. Nämäkin luokat löytyvät Maven-moduulista uploader-implementations.

RefactoredMessageProcessor.java

```
1  package fi.jyu.jutarant.refactored.service;
2
3  import fi.jyu.jutarant.general.messaging.MessagingInterface;
4  import fi.jyu.jutarant.general.model.Customer;
5  import fi.jyu.jutarant.general.model.EntityType;
6  import fi.jyu.jutarant.general.model.Payload;
7  import fi.jyu.jutarant.general.model.ThirdPartyAuthentication;
8  import fi.jyu.jutarant.general.model.TransferState;
9  import fi.jyu.jutarant.general.model.UploadStatus;
10 import fi.jyu.jutarant.general.service.CustomerService;
11 import fi.jyu.jutarant.general.service.ExternalAuthorizationService;
12 import fi.jyu.jutarant.general.service.PayloadService;
13 import fi.jyu.jutarant.general.serviceinterface.MessageProcessor;
14 import fi.jyu.jutarant.general.thirdpartyspecific.TransferStateStore;
15
16 /**
17  * Process a {@link Payload} and upload it to a third party with the
18  * given {@link AbstractUploadableFileService} implementation.
19  */
20 public class RefactoredMessageProcessor implements MessageProcessor {
21
22     private PayloadService payloadService;
23     private CustomerService customerService;
24     private ExternalAuthorizationService externalAuthenticationService;
25     private TransferStateStore transferStateStore;
26     private MessagingInterface messagingInterface;
27     private UploaderConfiguration config;
28     private AbstractUploadableFileService uploadableFileService;
```

```
29    private TransferStateService transferStateService ;
30
31    public void ignite(int payloadId) {
32      Payload payload = payloadService.load(payloadId);
33      Customer customer = customerService.load(payload.getOwner());
34       ThirdPartyAuthentication authentication =
              externalAuthenticationService . loadUserAuthentication (
35        customer. getId (), config .getThirdPartyServiceName());
36
37       if ( authentication != null) {
38         TransferState transferState = this . createNewTransferState (payloadId, payload.getOwner(),
               EntityType.ORDER);
39
40         messagingInterface . enqueueTransferState ( transferState . getDatabaseId () );
41       }
42    }
43
44    public void process(int transferStateId ) {
45       TransferState state = transferStateStore .load( transferStateId );
46      UploadStatus status = state . getUploadStatus ();
47
48      switch ( status ) {
49      case STARTED:
50        byte[] encodedPayload = uploadableFileService .encodePayload( state );
51         state .setEncodedPayload(encodedPayload);
52         uploadableFileService . uploadFile ( state );
53          transferStateService .updateComponentState(state);
54          transferStateStore .update( state );
55
56        if (UploadStatus.ERROR != state.getUploadStatus()) {
57          messagingInterface . enqueueTransferState ( state . getDatabaseId () );
58        }
59        break;
60      case UPLOADED:
61         uploadableFileService .checkUploadStatus( state );
62          transferStateService .updateComponentState(state);
63          transferStateStore .update( state );
```

```
64
65          if (UploadStatus.ERROR != state.getUploadStatus()) {
66              messagingInterface.enqueueTransferState(state.getDatabaseId());
67          }
68      case FINISHED:
69          break;
70      case ERROR:
71          break;
72      }
73    }
74
75    private TransferState createNewTransferState(int payloadId, int owner, EntityType order) {
76      TransferState state = new TransferState();
77      state.setEntityId(payloadId);
78      state.setOwner(owner);
79      state.setUploadStatus(UploadStatus.STARTED);
80      state.setEntityType(order);
81      return transferStateStore.create(state);
82    }
83  }
```

AbstractUploadableFileService.java

```
1  package fi.jyu.jutarant.refactored.service;
2
3  import fi.jyu.jutarant.general.model.Customer;
4  import fi.jyu.jutarant.general.model.Payload;
5  import fi.jyu.jutarant.general.model.ThirdPartyAuthentication;
6  import fi.jyu.jutarant.general.model.TransferState;
7  import fi.jyu.jutarant.general.model.TransferStateTransition;
8  import fi.jyu.jutarant.general.model.UploadStatus;
9  import fi.jyu.jutarant.general.service.CustomerService;
10 import fi.jyu.jutarant.general.service.ExternalAuthorizationService;
11 import fi.jyu.jutarant.general.service.PayloadService;
12 import fi.jyu.jutarant.general.thirdpartyspecific.TransferStateStore;
13 import fi.jyu.jutarant.general.thirdpartyspecific.TransferStateTransitionStore;
14 import fi.jyu.jutarant.general.thirdpartyspecific.UploadService;
15
16 public abstract class AbstractUploadableFileService {
```

```
17
18      private UploadService uploadService;
19      private PayloadService payloadService;
20      private CustomerService customerService;
21      private ExternalAuthorizationService externalAuthenticationService ;
22      private UploaderConfiguration config;
23      private TransferStateTransitionStore transferStateTransitionStore ;
24      private TransferStateStore transferStateStore ;
25
26      /**
27       * Load a translated {@link Payload} as a byte array, that is suitable for
28       * uploading to a specific third party service.
29       * @param state
30       * @return
31       */
32      public abstract byte[] encodePayload(TransferState state);
33
34      /**
35       * Upload an encoded {@link Payload} within a {@link TransferState} to
36       * a third party service.
37       * @param state
38       */
39      public void uploadFile(TransferState state) {
40        Payload payload = payloadService.load(state.getPayloadId());
41        Customer customer = customerService.load(payload.getOwner());
42         ThirdPartyAuthentication authentication =
                externalAuthenticationService.loadUserAuthentication(
43          customer.getId(), config.getThirdPartyServiceName());
44
45        try {
46          uploadService.uploadToThirdParty(state.getEncodedPayload(), authentication);
47        } catch (Exception e) {
48           TransferStateTransition transition = new TransferStateTransition();
49          transition.setNewStatus(UploadStatus.ERROR);
50          transition.setOriginalStatus(state.getUploadStatus());
51          transition.setTransferStateId(state.getDatabaseId());
```

```
52          transition . setTransitionReason ("Error uploading to third party service for state " +
                state . getDatabaseId () );
53
54          state . setUploadStatus (UploadStatus.ERROR);
55
56          transferStateStore  .update( state );
57          transferStateTransitionStore  .update( transition );
58      }
59  }
60
61  /**
62   * Check that a {@link Payload} has been successfully uploaded.
63   * @param state
64   */
65  public void checkUploadStatus( TransferState  state ) {
66      Payload payload = payloadService . load( state . getPayloadId () );
67      Customer customer = customerService . load (payload . getOwner());
68      ThirdPartyAuthentication  authentication  =
                externalAuthenticationService  . loadUserAuthentication (
69          customer. getId () ,  config . getThirdPartyServiceName());
70
71      try {
72          uploadService . checkUploadStatus(payload. getId () ,  authentication );
73      } catch (Exception e) {
74          TransferStateTransition  transition  = new TransferStateTransition ();
75          transition .setNewStatus(UploadStatus.ERROR);
76          transition .  setOriginalStatus ( state . getUploadStatus () );
77          transition .  setTransferStateId ( state . getDatabaseId () );
78          transition . setTransitionReason ("Error querying third party service for state " +
                state . getDatabaseId () );
79
80          state . setUploadStatus (UploadStatus.ERROR);
81
82          transferStateStore  .update( state );
83          transferStateTransitionStore  .update( transition );
84      }
85  }
```

```
86 }
```

JyuUploadableFileService.java

```
1  package fi.jyu.jutarant.refactored.service;
2
3  import fi.jyu.jutarant.general.model.TransferState;
4  import fi.jyu.jutarant.general.model.TransferStateTransition;
5  import fi.jyu.jutarant.general.model.UploadStatus;
6  import fi.jyu.jutarant.general.service.EncodedPayloadService;
7  import fi.jyu.jutarant.general.thirdpartyspecific.TransferStateTransitionStore;
8
9  /**
10  * The uploader implementation for Third Party partner
11  * JYU (University of Jyväskylä).
12  */
13 public class JyuUploadableFileService extends AbstractUploadableFileService {
14
15   private EncodedPayloadService encodedPayloadService;
16   private TransferStateTransitionStore transferStateTransitionStore;
17
18   public byte[] encodePayload(TransferState state) {
19     byte[] encodedPayload = encodedPayloadService.load(state.getOwner(), state.getPayloadId());
20
21     if (encodedPayload == null) {
22       TransferStateTransition transition = new TransferStateTransition();
23       transition.setTransferStateId(state.getDatabaseId());
24       transition.setTransitionReason("Could not encode payload for state " +
             state.getDatabaseId());
25       transition.setOriginalStatus(state.getUploadStatus());
26       transition.setNewStatus(UploadStatus.ERROR);
27       transferStateTransitionStore.create(transition);
28
29       throw new RuntimeException(transition.getTransitionReason());
30     }
31     return encodedPayload;
32   }
33 }
```

TransferStateService.java

```
1  package fi.jyu.jutarant.refactored.service;
2
3  import fi.jyu.jutarant.general.model.TransferState;
4  import fi.jyu.jutarant.general.model.TransferStateTransition;
5  import fi.jyu.jutarant.general.model.UploadStatus;
6  import fi.jyu.jutarant.general.thirdpartyspecific.TransferStateTransitionStore;
7
8  /**
9   * Update a transfer state and create a matching transition.
10  */
11 public class TransferStateService {
12   private TransferStateTransitionStore transferStateTransitionStore;
13
14   public void updateComponentState(TransferState state) {
15     switch (state.getUploadStatus()) {
16       case STARTED:
17         state.setUploadStatus(UploadStatus.UPLOADED);
18         this.createNewTransition(UploadStatus.STARTED, UploadStatus.UPLOADED, state);
19         break;
20       case UPLOADED:
21         state.setUploadStatus(UploadStatus.FINISHED);
22         this.createNewTransition(UploadStatus.UPLOADED, UploadStatus.FINISHED, state);
23         break;
24       default:
25         break;
26     }
27   }
28
29   private void createNewTransition(UploadStatus originalStatus, UploadStatus newStatus,
         TransferState state) {
30     TransferStateTransition transition = new TransferStateTransition();
31     transition.setOriginalStatus(originalStatus);
32     transition.setNewStatus(newStatus);
33     transition.setTransferStateId(state.getDatabaseId());
34     transferStateTransitionStore.create(transition);
35   }
```

```
36 }
```

UploaderConfiguration.java

```
1  package fi.jyu.jutarant.refactored.service;
2
3  /**
4   * Shared configuration options.
5   */
6  public class UploaderConfiguration {
7
8    private String thirdPartyServiceName;
9
10   public String getThirdPartyServiceName() {
11     return thirdPartyServiceName;
12   }
13
14   public void setThirdPartyServiceName(String thirdPartyServiceName) {
15     this.thirdPartyServiceName = thirdPartyServiceName;
16   }
17 }
```

D Satunnaiset pienemmät palat lähdekoodia

Nämä kaksi tiedostoa ovat koko projektin Maven-määrittely, sekä refaktoroidun lähdekoodin sisältävän alimoduulin uploader-base Maven-määrittely.

pom.xml

```
1  <project>
2    <modelVersion>4.0.0</modelVersion>
3    <groupId>fi.jyu.jutarant</groupId>
4    <version>0.1-SNAPSHOT</version>
5    <artifactId>uploader-parent</artifactId>
6    <packaging>pom</packaging>
7    <modules>
8      <module>uploader-implementations</module>
9      <module>uploader-base</module>
10   </modules>
```

```
11      <build>
12      </build>
13      <dependencyManagement>
14          <dependencies>
15          </dependencies>
16      </dependencyManagement>
17  </project>
```

uploader-implementations/pom.xml

```
1  <project>
2    <modelVersion>4.0.0</modelVersion>
3    <parent>
4      <groupId>fi.jyu.jutarant</groupId>
5      <version>0.1-SNAPSHOT</version>
6      <artifactId>uploader-parent</artifactId>
7    </parent>
8    <artifactId>uploader-implementations</artifactId>
9    <packaging>jar</packaging>
10   <build>
11   </build>
12   <dependencies>
13     <dependency>
14       <artifactId>uploader-base</artifactId>
15       <groupId>fi.jyu.jutarant</groupId>
16       <version>${project.version}</version>
17     </dependency>
18   </dependencies>
19 </project>
```

E Uploader-base Maven-moduulin lähdekoodi

Koska kaikki muu lähdekoodi jaettiin sekä refaktoroimattoman että refaktoroidun koodin kesken, on ne kasattu selkeyden vuoksi omaan Maven-moduliinsa uploader-base.

uploader-base/pom.xml

```
1  <project>
2    <modelVersion>4.0.0</modelVersion>
```

```
3     <parent>
4       <groupId>fi.jyu. jutarant </groupId>
5       <version>0.1-SNAPSHOT</version>
6       < artifactId >uploader-parent</ artifactId >
7     </parent>
8     < artifactId >uploader-base</ artifactId >
9     <packaging>jar</packaging>
10    <build>
11    </build>
12    <dependencyManagement>
13        <dependencies>
14        </dependencies>
15    </dependencyManagement>
16  </ project >
```

Tästä eteenpäin kaikki tiedostot sijaitsevat polussa “uploader-base/src/main/java/fi/jyu/jutarant/general”, joka jätetään selkeyden vuoksi toistamatta.

messaging/ExternalMessageReceiver.java

```
1  package fi.jyu. jutarant .general.messaging;
2
3  import fi.jyu. jutarant .general. serviceinterface .MessageProcessor;
4
5  public class ExternalMessageReceiver {
6
7    private MessageProcessor processor;
8
9    /**
10    * Receive a message from the global message bus.
11    * @param message
12    */
13   public void listenToNotification ( ExternalPayloadNotificationMessage message) {
14
15     switch (message.getStatus ()) {
16     case INITIAL_REGISTRATION:
17       // We're not interested in the payload yet
18       break;
19     case WAITING_ADDITIONAL_DATA:
```

```
20          // We're not interested in the payload yet
21          break;
22        case FINISHED:
23          processor.ignite(message.getPayloadId());
24          break;
25        }
26      }
27  }
```

messaging/ExternalPayloadNotificationMessage.java

```
1   package fi.jyu.jutarant.general.messaging;
2
3   import fi.jyu.jutarant.general.model.Payload;
4
5   /**
6    * A message from an external service, indicates that a {@link Payload} has been modified.
7    */
8   public class ExternalPayloadNotificationMessage {
9
10    private int payloadId;
11    private PayloadProcessingStatus status;
12
13    public int getPayloadId() {
14      return payloadId;
15    }
16
17    public void setPayloadId(int payloadId) {
18      this.payloadId = payloadId;
19    }
20
21    public PayloadProcessingStatus getStatus() {
22      return status;
23    }
24
25    public void setStatus(PayloadProcessingStatus status) {
26      this.status = status;
27    }
28
```

```
29 }
```

messaging/InternalMessageReceiver.java

```
1  package fi.jyu.jutarant.general.messaging;
2
3  import fi.jyu.jutarant.general.serviceinterface.MessageProcessor;
4
5  public class InternalMessageReceiver {
6
7    private MessageProcessor messageProcessor;
8
9    /**
10    * Receive an internal state transition message from a message bus.
11    *
12    * @param message
13    */
14   public void process(InternalNotificationMessage message) {
15     messageProcessor.process(message.getTransferStateId());
16   }
17 }
```

messaging/InternalNotificationMessage.java

```
1  package fi.jyu.jutarant.general.messaging;
2
3  import fi.jyu.jutarant.general.model.TransferState;
4
5  /**
6   * A message describing for asynchronously handling
7   * {@link TransferState}s.
8   */
9  public class InternalNotificationMessage {
10
11   private int transferStateId;
12
13   public int getTransferStateId() {
14     return transferStateId;
15   }
```

```
16
17    public void setTransferStateId(int transferStateId) {
18        this.transferStateId = transferStateId;
19    }
20
21 }
```

messaging/MessagingInterface.java

```
1  package fi.jyu.jutarant.general.messaging;
2
3  public interface MessagingInterface {
4
5    /**
6     * Send an internal state transition message to a message bus.
7     */
8    void enqueueTransferState(int id);
9
10 }
```

messaging/PayloadProcessingStatus.java

```
1  package fi.jyu.jutarant.general.messaging;
2
3  import fi.jyu.jutarant.general.model.Payload;
4
5  /**
6   * Global {@link Payload} processing status.
7   */
8  public enum PayloadProcessingStatus {
9    INITIAL_REGISTRATION,
10   WAITING_ADDITIONAL_DATA,
11   FINISHED
12
13 }
```

model/Customer.java

```
1  package fi.jyu.jutarant.general.model;
2
```

```
3  public class Customer {
4
5    private int id;
6
7    public int getId() {
8      return id;
9    }
10
11   public void setId(int id) {
12     this.id = id;
13   }
14
15 }
```

model/EntityType.java

```
1  package fi.jyu.jutarant.general.model;
2
3  public enum EntityType {
4    ORDER
5  }
```

model/Payload.java

```
1  package fi.jyu.jutarant.general.model;
2
3  public class Payload {
4    private int id;
5    private int owner;
6
7    public int getId() {
8      return id;
9    }
10
11   public void setId(int id) {
12     this.id = id;
13   }
14
15   public int getOwner() {
```

```
16        return owner;
17    }
18
19    public void setOwner(int owner) {
20        this.owner = owner;
21    }
22 }
```

model/ThirdPartyAuthentication.java

```
1  package fi.jyu.jutarant.general.model;
2
3  /**
4   * A representation of an OAuth-like token for a customer,
5   * for uploading data to a third party service.
6   */
7  public class ThirdPartyAuthentication {
8
9  }
```

model/TransferState.java

```
1  package fi.jyu.jutarant.general.model;
2
3  /**
4   * A model for describing the processing state of a {@link Payload}
5   * within the Uploader.
6   */
7  public class TransferState {
8
9     private int databaseId;
10    private UploadStatus uploadStatus;
11    private EntityType entityType;
12    private int owner;
13    private int payloadId;
14    private byte[] encodedPayload;
15
16    public UploadStatus getUploadStatus() {
17       return uploadStatus;
```

```
18      }
19
20      public void setUploadStatus(UploadStatus uploadStatus) {
21          this.uploadStatus = uploadStatus;
22      }
23
24      public int getOwner() {
25          return owner;
26      }
27
28      public void setOwner(int owner) {
29          this.owner = owner;
30      }
31
32      public EntityType getEntityType() {
33          return entityType;
34      }
35
36      public void setEntityType(EntityType entityType) {
37          this.entityType = entityType;
38      }
39
40      public int getDatabaseId() {
41          return databaseId;
42      }
43
44      public void setId(int id) {
45          this.databaseId = id;
46      }
47
48      public void setEntityId(int payloadId) {
49          this.setPayloadId(payloadId);
50      }
51
52      public int getPayloadId() {
53          return payloadId;
54      }
```

```
55
56    public void setPayloadId(int payloadId) {
57        this.payloadId = payloadId;
58    }
59
60    public void setEncodedPayload(byte[] encodedPayload) {
61        this.encodedPayload = encodedPayload;
62    }
63
64    public byte[] getEncodedPayload() {
65        return encodedPayload;
66    }
67
68 }
```

model/TransferStateTransition.java

```
 1 package fi.jyu.jutarant.general.model;
 2
 3 /**
 4  * A log file describing the transition of a {@link TransferState}
 5  * from one {@link UploadStatus} to another.
 6  */
 7 public class TransferStateTransition {
 8     private int transferStateId;
 9     private UploadStatus originalStatus;
10     private UploadStatus newStatus;
11     private String transitionReason;
12
13     public int getTransferStateId() {
14         return transferStateId;
15     }
16
17     public void setTransferStateId(int transferStateId) {
18         this.transferStateId = transferStateId;
19     }
20
21     public UploadStatus getOriginalStatus() {
22         return originalStatus;
```

```
23      }
24
25      public void setOriginalStatus (UploadStatus originalStatus ) {
26        this . originalStatus = originalStatus ;
27      }
28
29      public UploadStatus getNewStatus() {
30        return newStatus;
31      }
32
33      public void setNewStatus(UploadStatus newStatus) {
34        this .newStatus = newStatus;
35      }
36
37      public String getTransitionReason () {
38        return transitionReason ;
39      }
40
41      public void setTransitionReason ( String transitionReason ) {
42        this . transitionReason = transitionReason ;
43      }
44
45  }
```

model/UploadStatus.java

```
1   package fi .jyu . jutarant . general .model;
2
3   /**
4    * The possible processing states of a {@link Payload}
5    * within the Uploader.
6    */
7   public enum UploadStatus {
8     STARTED, UPLOADED, FINISHED, ERROR
9   }
```

service/CustomerService.java

```
1   package fi .jyu . jutarant . general . service ;
```

```
2
3  import fi.jyu.jutarant.general.model.Customer;
4
5  public interface CustomerService {
6
7    Customer load(int owner);
8
9  }
```

service/EncodedPayloadService.java

```
1  package fi.jyu.jutarant.general.service;
2
3  import fi.jyu.jutarant.general.model.Payload;
4
5  public interface EncodedPayloadService {
6
7    /**
8     * Translates a {@link Payload} to a third party specific format and
9     * returns a byte array, for uniform handling.
10    * @param owner
11    * @param payloadId
12    * @return
13    */
14   byte[] load(int owner, int payloadId);
15
16 }
```

service/ExternalAuthorizationService.java

```
1  package fi.jyu.jutarant.general.service;
2
3  import fi.jyu.jutarant.general.model.ThirdPartyAuthentication;
4
5  public interface ExternalAuthorizationService {
6
7    /**
8     * Loads an OAuth-style authentication token for a
9     * specific user-third party service combination.
```

```
10      * @param userId
11      * @param thirdPartyName
12      * @return
13      */
14     public ThirdPartyAuthentication loadUserAuthentication(int userId, String thirdPartyName);
15 }
```

service/PayloadService.java

```
1  package fi.jyu.jutarant.general.service;
2
3  import fi.jyu.jutarant.general.model.Payload;
4
5  public interface PayloadService {
6
7    Payload load(int payloadId);
8  }
```

serviceinterface/MessageProcessor.java

```
1  package fi.jyu.jutarant.general.serviceinterface;
2
3  import fi.jyu.jutarant.general.model.Payload;
4  import fi.jyu.jutarant.general.model.TransferState;
5
6  /**
7   * The business logic interface. Responsible for managing the state and
8   * processing of {@link Payload} in the service.
9   */
10 public interface MessageProcessor {
11
12   /**
13    * Receive a notification that a {@link Payload} is ready for processing.
14    * @param payloadId
15    */
16   public void ignite(int payloadId);
17
18   /**
19    * Perform the next processing step in the processing sequence of
```

```
20      * the given {@link TransferState }.
21      * @param transferStateId
22      */
23     public void process(int transferStateId);
24 }
```

thirdpartyspecific/TransferStateStore.java

```
1  package fi.jyu.jutarant.general.thirdpartyspecific;
2
3  import fi.jyu.jutarant.general.model.TransferState;
4
5  /**
6   * Persistent {@link TransferState} storage.
7   */
8  public interface TransferStateStore {
9
10   public TransferState create(TransferState state);
11
12   public TransferState load(int transferStateId);
13
14   public void update(TransferState state);
15
16 }
```

thirdpartyspecific/TransferStateTransitionStore.java

```
1  package fi.jyu.jutarant.general.thirdpartyspecific;
2
3  import fi.jyu.jutarant.general.model.TransferState;
4  import fi.jyu.jutarant.general.model.TransferStateTransition;
5
6  /**
7   * Persistent storage for state changes of a {@link TransferState }.
8   */
9  public interface TransferStateTransitionStore {
10
11   public TransferStateTransition create(TransferStateTransition state);
12
```

```
13    public void update( TransferStateTransition    transition );
14
15  }
```

thirdpartyspecific/UploadService.java

```
1  package fi.jyu.jutarant.general.thirdpartyspecific;
2
3  import fi.jyu.jutarant.general.model.Payload;
4  import fi.jyu.jutarant.general.model.ThirdPartyAuthentication;
5
6  /**
7   * Upload a processed {@link Payload} to a third party service.
8   */
9  public interface UploadService {
10
11    public void uploadToThirdParty(byte[] encodedPayload, ThirdPartyAuthentication    authentication );
12
13    public void checkUploadStatus(int payloadId, ThirdPartyAuthentication    authentication );
14 }
```

F Pakattu lähdekoodi YouSourcessa

Linkin toimivuus on todennäköisesti huomattavan rajattu.

https://yousource.it.jyu.fi/~jutarant/latex-thesis-classes/rantanen-gradu/blobs/master/gradu-rantanen-cleaned.zip