Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Juha Säävälä

Aspektipohjaisen lokitusjärjestelmän toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Ohjelmistotuontanto

Insinöörityö

3.5.2020

Tekijä Otsikko Sivumäärä Aika	Juha Säävälä Aspektipohjaisen lokitusjärjestelmän toteutus 30 sivua 3.5.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Simo Silander Pääarkkitehti Sari Uotila

Insinöörityön tarkoituksena oli lisätä asiakkaan asiakkuudenhallintasovelluksiin lokitusta mahdollisimman pienellä työmäärällä, mutta kuitenkin niin kattavasti, että sitä voitaisiin hyödyntää testauksessa.

Asiakkaan sovellukset olivat vielä kehityksessä. Ne oli toteutettu Javalla käyttäen Spring-ohjelmistokehystä. Sovelluksissa oli hyvin vähän olemassa olevaa lokitusta eikä sekään ollut kovin hyödyllistä. Lokituksen haluttiin olevan mahdollisimman vähätöistä ja helposti muokattavissa, mutta sen piti kuitenkin kattaa sovelluksien sisäinen toiminta mahdollisimman kattavasti.

Työ koostui olemassa olevan lokituksen analyysista, ratkaisuvaihtoehtojen kartoittamisesta, ratkaisuvalinnasta, toteutuksesta ja dokumentaation sekä ohjeistuksen tuottamisesta. Toteutettavaksi ratkaisuksi valittiin lokituksen lisääminen AspectJ-ohjelmistokehystä käyttäen niin, että olemassa olevaan koodiin generoitaisiin lokituksen toteuttava koodi ohjelman käännösvaiheessa.

Työ saatettiin loppuun onnistuneesti ja otettiin heti käyttöön sovellusten kehitys- ja testityössä. Tämän jälkeen tehtiin jonkin verran jälkityötä käytössä ilmi käyneiden ongelmien korjaamiseksi ja toiminnallisuuden laajentamiseksi.

Avainsanat	Logging, AoP, AspectJ, Java

Author	Juha Säävälä
Title	Development of a Logging System Utilizing Aspects in Java
Number of Pages	30 pages
Date	3 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Simo Silander, Senior Lecturer Sari Uotila, Architecture Team Lead

The goal of this thesis was to add logging into the client's customer relationship management applications with as little work as possible, while still covering enough of the applications internal functionality to be useful.

The client's applications were still undergoing significant development, they were implemented with Java using the Spring framework. The applications had barely any logging and what existed was not very useful. The logging needed to be easily changeable with the fewest required changes while still covering as much of the functionality as possible.

The work consisted of analyzing the existing logging, determining usable solution options, choosing one of the options, implementing it and finally producing documentation and guidelines for the system. The chosen solution was to add logging by using the AspectJ framework to add advices that interacted with the existing code to generate logging without requiring changes to the code itself.

This work was finished successfully and immediately taken into use in development and testing of the applications. Some further follow-up has been done to add functionality and fix problems that were found in usage.

Keywords	Logging, AoP, AspectJ, Java

Sisällys

Lyhenteet

AOP	Aspekti-ohjelmointi. Ohjelmointiparadigma, jolla yritetään ratkaista ohjelman läpileikkaavia tarpeita keskitetyllä ratkaisulla.
OOP	Olio-ohjelmointi. Ohjelmointiparadigma, jossa ratkaisut toteutetaan olioiden yhteistoiminnalla.
CCC	Cross-cutting concern. Ohjelmiston läpileikkaava tarve.
API	Application programming interface. Ohjelmointirajapinta.
JVM	Java Virtual Machine. Ohjelma, jolla Java-ohjelmia voidaan ajaa.
MDC	Mapped Diagnostic Context. Säiekohtainen konteksti, jota voidaan käyttää prosessikohtaisten tietojen säilyttämiseen koko prosessin ajaksi.
CRM	Customer relationship management. Asiakkuudenhallintajärjestelmä, käytetään asiakkuuksien hallinnoimiseen.
SOAP	Simple Object Access Protocol. Viestiprotokolla, joka käyttää XML-formaattisia viestejä sovellusten väliseen tietoliikenteeseen.
IDE	Integrated development environment. Ohjelmistokehityksessä käytetty ohjelmisto, joka tarjoaa erinäisiä aputoiminnallisuuksia kehitystyön helpottamiseksi.

1 Johdanto

Lokitus on toiminto, jossa ohjelman prosessien suorittamisen aikana tallennetaan niistä tietoja halutulla tavalla. Lokitapahtumista koostuu loki, josta voidaan parhaimmillaan seurata järjestelmän operaatiota lokin tarkoituksen mukaisesti. Lokeja voidaan käyttää moniin eri tarkoitukseen, ja usein järjestelmät tuottavat monia erilaisia lokeja. Esimerkiksi auditointilokeja järjestelmän rahan käsittelystä ja lokeja järjestelmän autentikoinneista.

Insinöörityön tavoitteena oli kehittää Accenture Oy:n asiakkaan uusiin asiakkuudenhallintaohjelmiin (CRM) kattava lokitusratkaisu, jota voitaisiin hyödyntää sekä kehitys- ja tuotantoympäristöissä. Varsinaiset vaatimukset työlle tulivat itse kehitystiimin tarpeista ja lokituksesta ei tarvinnut keskustella suoraan asiakkaan kanssa, vaan se tehtiin osana kehityksen tukitoimia.

Asiakkaan CRM:t olivat olleet kehityksessä jo useamman vuoden ajan, mutta niissä ei ollut mitään keskitettyä ja laajaa lokitustapaa. Tämä vaikeutti kehitys- ja testaustyötä, koska kaikissa kehitys- ja testiympäristöissä ei ollut mahdollista ajaa CRM:iä debuggereissa. Debuggerit olivat lisäksi liian tekninen ratkaisu järjestelmän tapahtumien seuraamiseen ollakseen sovellettavissa yleiseen käyttöön. Lisäksi asiakastestauksen havaintojen taustatapahtumien selvittäminen asiakkaan nostettua niistä havainto ei aina ollut mahdollista, koska tapaukset eivät välttämättä olleet toistettavissa samalla testitapauksella. Tähän ratkaisuksi haluttiin lisätä sovelluksiin kattava lokitus, joka jättäisi tarkkoja pysyviä jälkiä prosessien etenemisestä sovelluksissa.

Työ tehtiin vuoden 2019 alussa muutaman kuukauden aikana käyttäen yleisesti viikoittain noin yksi päivä työhön, ja lisäksi yksi kokonainen viikko toteutusvaiheessa. Tämän jälkeen tehtiin vielä erinäisiä korjauksia ja jatkokehitystä.

Työn toisessa luvussa käydään läpi asiakkaan CRM:ien rakenne ja lokitukselle asetetut vaatimukset. Kolmannessa luvussa käsitellään lokitusta yleisesti ja myös Javan lokitusteknologioita. Neljännessä luvussa kuvataan aspektipohjaista ohjelmointia (AoP), joka oli valittu lokitusratkaisu. Luvussa viisi käydään läpi työhön vaikuttaneita teknologioita, joita sovelluksessa oli käytetty. Viides luku käsittelee itse toteutusprosessia ja mahdollisia jatkokehitysvaihtoehtoja. Viimeisessä luvussa tehdään yhteenveto tehdystä työstä.

2 Työn alkutilanne

Tässä luvussa kuvataan sovelluksen ja lokituksen alkutilanne ja vaatimukset lokituksen muutoksille.

2.1 Sovellusympäristö

Työn kohteena on asiakkaan kaksi erillistä yhteistoiminnallista asiakkuudenhallintasovellusta, jotka on toteutettu Javalla. Ne on toteutettu Model-View-Controller-arkkitehtuurilla käyttäen Spring-ohjelmointikehystä sekä käyttöliittymän että taustajärjestelmien toteutukseen. Sovelluksien ja muiden asiakkaan käyttämien ulkoisten palveluiden välinen liikenne oli toteutettu SOAP-tietoliikenneprotokollalla. Tietokantayhteyksiin sovelluksista käytettiin MyBatis-ohjelmistokehystä. Lisäksi sovellukset käyttivät Apache Mavenia käännösautomaatioon ja riippuvuuksien hallintaan. Ohjelmistot sisälsivät myös merkittävän määrän Spring Batch-kehyksellä toteutettuja eräajoja, jotka ylläpitivät asiakkaan jatkuvia prosesseja.

Sovellukset koostuivat muutamasta yhteiskäyttöisestä Java-kirjastosta ja kummatkin sovellukset on toteutettu omina Java-projekteinaan. Itse sovellusprojektien yhteiskoko oli yli miljoona riviä.

Työn alkaessa sovelluksissa ei ollut yleistä keskitettyä lokitusratkaisua. Toisessa sovelluksista osa virheviesteistä kaapattiin ja muokattiin yhdenmukaiseen muotoon, mutta tämä ei kattanut koko sovellusta. Yleisesti lokitus oli toteutettu luokittain, ja kaikki lokitusviestit oli lisättävä metodikohtaisesti ohjelmoijan toimesta. Ne käyttivät SLF4J-fasadia ja taustalla Log4J2-ohjelmointikehystä. Lokit luotiin useisiin tekstitiedostoihin eri lokaatioissa. Lisäksi lokitettu tieto oli epätarkkaa ja puutteellista eikä kattanut tarpeellista osaa sovelluksesta. Sovellukset toimivat myös monikäyttäjäympäristössä ja lokitusviesteistä ei voitu päätellä sen käynnistänyttä käyttäjää tai prosessia johtaen ongelmiin prosessien seuraamisessa.

2.2 Vaatimukset

Lokitusjärjestelmän muutoksille muodostettiin ensimmäisessä palaverissa seuraavat vaatimukset:

- Molempien sovellusten haluttiin tuottavan yhdenmukaista lokitusta. Tällä tarkoitettiin, että loki tuotteiden pitäisi olla samanlaisia molemmissa sovelluksissa, ja mielellään myös toteutettu samalla tavalla.
- Sovellusten haluttiin lokittavan kattavasti prosessien toiminnallisuutta. Tämä tarkoitti myös, että sovelluksessa lokitettaisiin mieluummin liikaa kuin liian vähän.
- Prosessien käynnistäjien piti olla tunnistettavissa lokiriveiltä. Tämä tarkoitti sitä, että vaikka useampi käsittelijä saattoi käyttää vastaavia toiminnallisuuksia sovelluksessa saman aikaisesti, pitäisi jokaisen käynnistetyn prosessin erottua lokeilta yksiselitteisesti. Tämän piti toteutua, vaikka sama käyttäjä käynnistäisi prosessin useamman kerran samanaikaisesti.
- Lokituksen toteuttamiseen haluttiin käyttää mahdollisimman vähän resursseja. Kenenkään ohjelmoijan ei saisi myöskään olla pakko tehdä muutoksia peruslokituksen lisäämiseksi uusiin tai jo olemassa oleviin toiminnallisuuksiin.
- Lokituksen määrää ja sijaintia piti pystyä muokkaamaan konfiguraatiotiedoston tai JVM-argumenttien kautta.
- Osana työtä piti tuottaa dokumentaatio siitä, miten lokitus toimii, sitä ohjataan, miten se pitää ottaa huomioon kehityksen aikana ja miten sitä voidaan käyttää.
- Lokituotteiden piti olla yhdenmukaisessa muodossa, joka piti myös pystyä lukemaan käsittelyjärjestelmiin ilman massiivisia jäsentimiä.

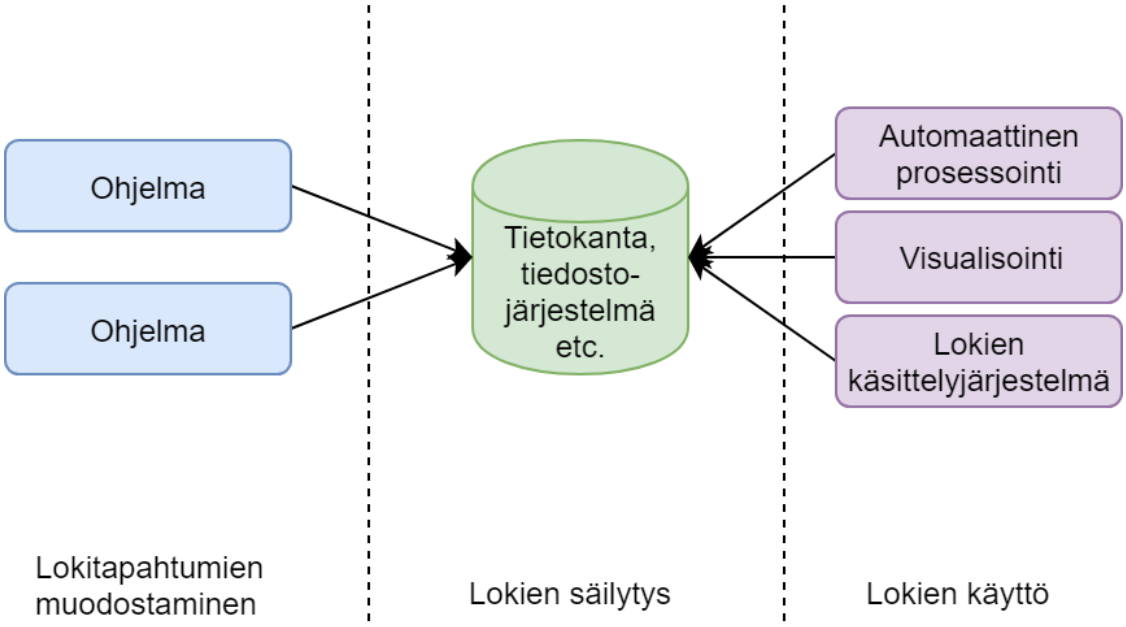
Varsinkin vähätöisyys oli painotettu vaatimuksena, koska sovellukset olivat yhä kehityksessä ja lokitusta tarvittiin kehityksen aikaiseen käyttöön nopeasti mutta siihen ei voitu keskittää merkittäviä resursseja.

3 Lokituksesta

3.1 Yleistä lokituksesta

Lokituksella tarkoitetaan ohjelman tekemää tapahtumarekisteriä, johon tallennetaan järjestelmän tapahtumia prosessista erilliseen muotoon. Lokitusjärjestelmät voidaan yleisesti jakaa kolmeen eri vaiheeseen: lokitapahtumien muodostamiseen, tallennukseen ja

lokien käsittelyyn. Näitä vaiheita havainnollistaa Kuva 1. Lokitiedostot koostuvat yleensä yhdestä rivistä tapahtumaa kohden ja yksittäisestä rivistä käy yleensä ilmi, mitä, milloin ja kenen tai minkä toimesta asia tapahtui. [1.]



Kuva 1. Lokituksen perusprosessi.

Ohjelmiston prosessien seuraamisessa esimerkiksi. tuotantoympäristössä lokitus on ensi arvoisen tärkeää virhetilanteiden tapahtuessa, koska niiden avulla voidaan selvittää paremmin, mitä taustalla tapahtui ja käytetyt arvot. Yleisesti lokituksen tuottamat lokit voidaan jakaa kahteen kategoriaan, diagnostiikka- ja auditointilokeihin.

Auditointilokeilla tarkoitetaan laillisista tai liiketoiminnallisista vaatimuksista nousevia tarpeita kuten transaktio- ja muutostilanteiden lokitusta. Yleisesti tämä tarkoittaa, että kaikista käsittelijän, asiakkaan tai automaattisen prosessin tekemistä muutoksista halutaan pysyvä jälki, jotta voidaan tarkistaa tai todistaa, mitä järjestelmässä on tehty, milloin ja kenen toimesta.

```
{
    "TIME" : "15:10:10.200 15.1.2019",
    "USER" : "JUHA",
    "TARGET_ID" : "5090",
    "TARGET_TYPE" : "CUSTOMER",
    "ACTION" : "UPDATE_BANK_ACCOUNT",
    "RESULT" : "SUCCESS"
}
{
    "TIME" : "15:12:08.112 15.1.2019",
    "USER" : "JUHA",
    "TARGET_ID" : "5090",
    "TARGET_TYPE" : "CUSTOMER",
    "ACTION" : "CREATE_REFUND",
    "RESULT" : "SUCCESS"
}
```

Kuva 2. JSON-muotoisia auditloki-rivejä jaoteltuna useammalle riville.

Kuva 2 kuvaa kahta JSON-muotoisia lokitapahtumaa, joista käy ilmi, mitä on tehty, milloin, kenen toimesta ja onnistuiko operaatio. Ensimmäisessä tapahtumassa on tieto siitä, että käyttäjä Juha on päivittänyt asiakkaan 5090 pankkitietoja onnistuneesti. Toisessa taas käyttäjä Juha on luonut asiakkaalle 5090 palautuksen.

Diagnostiikkalokeilla tarkoitetaan lokeja, joita käytetään virhetilanteiden jäljittämiseen ja sovelluksen kulun seuraamiseen. Näitä käytetään yleisesti ohjelmistokehityksen aikana mutta myös jo tuotannossa olevien sovellusten ongelmatilanteiden selvittämisessä. Kattava ja hyödyllinen diagnostiikkalokitus mahdollistaa ongelmatilanteiden nopeamman hahmottamisen ja testauksen. Varsinkin tuotantoympäristöissä on mahdollista, että sovelluksen ongelmatilanteiden syitä voidaan analysoida vain lokien ja tietokannan avulla.

```
{
    "LEVEL" : "DEBUG",
    "UUID" : "5321423",
    "TIME" : "15:10:10.150 15.1.2019",
    "USER" : "JUHA",
    "METHOD" : "com.juha.test.BusinessClass1.updateBankAccount",
    "CALLING_METHOD" : "com.juha.test.CustomerInfoController.updateBankAccount",
    "VALUES" : [
        {"TYPE" : "String","VALUE" : "CH8793829565825070848"},
        {"TYPE" : "Long","VALUE" : "5090"}],
    "TIME_TAKEN" : "58 ms" ,
    "RETURN" : "true"
}
{
    "LEVEL" : "DEBUG",
    "UUID" : "3432142,
    "TIME" : "15:12:08.048 15.1.2019",
    "USER" : "JUHA",
    "METHOD" : "com.juha.test.BusinessClass2.createRefund",
    "CALLING_METHOD" : "com.juha.test.RefundsController.returnPayment",
    "VALUES" : [
        {"TYPE" : "Long", "VALUE" : "523214"}],
    "TIME_TAKEN" : "158 ms" ,
    "RETURN" : "90953"
}
```

Kuva 3. JSON-muotoisia lokirivejä

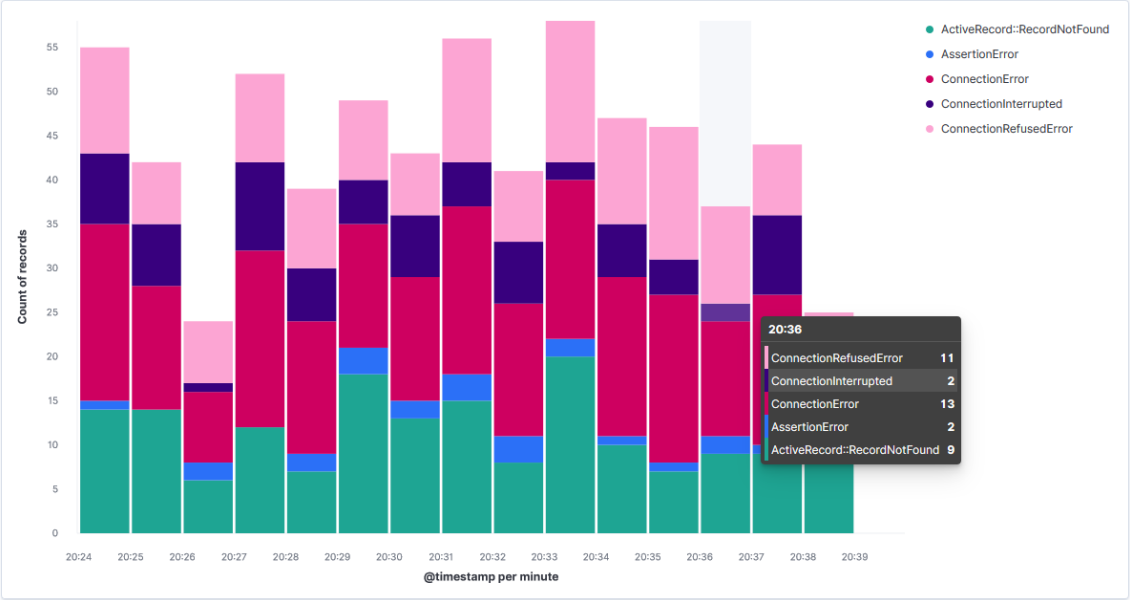
Kuva 3 kuvaa lokitapahtumia ja Kuva 2 toiminnat suorittavien metodien kutsuista nämä sisältävät metodin kutsussa käytetyt parametrit, kutsun paluutiedot sekä operaatioon kuluneen ajan.

Lokirivien struktuuri ja koneluettavuus ovat useimmissa käyttötarkoituksissa myös erittäin tärkeitä, ja yleinen ongelma lokien hyödyntämisessä on sovelluksen eri komponenttien tuottamien lokien vaihteleva rakenne. Eri lokitiedot ja struktuurit sovitetaan yhteen niin, että yksittäistä prosessia voidaan seurata alusta loppuun lokituksen kautta. Tämä vaatii käytännössä jonkinlaisen prosessin yksiselitteisesti tunnistavan tiedon lisäämistä kaikkiin lokitapahtumiin.

3.2 Lokitusdatan visualisointi

Usein jo tuotannossa olevat sovellukset saattavat tuottaa massiivisia määriä dataa lokeihin. Tämän datan hyödyntäminen voikin olla hyvin vaikeaa, ja hyödylliset tiedot huk-

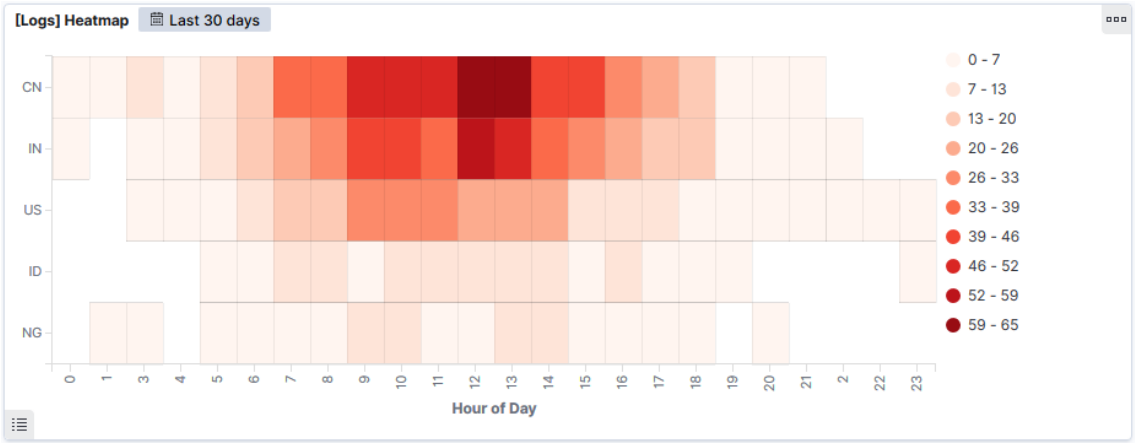
kuvat helposti massaan. Yksi tapa datan hahmottamisen helpottamiseksi on sen visualisointi. Visualisointi voi auttaa hahmottamaan poikkeustilanteita ja yleisesti järjestelmän performanssia.



Kuva 4. Kibanalla tuotettu visualisointi sovelluksen virhetyypeistä.

Esimerkiksi Kuva 4 visualisoi aikavälillä tapahtuneita poikkeuksia. Vastaavalla tavalla voitaisiin visualisoida esimerkiksi käsittelijän tekemien päivittäisten kutsujen kestoa ja jakaantumista. Tarkastelemalla tällä tavalla käsittelijän sovellusten toimintojen kutsuaikoja voitaisiin helpommin hahmottaa ja perustella parannuskohteita tieto-ohjautuvasti. Yleisesti visualisointi helpottaa merkittävästi datan hahmottumista ja auttaa myös myyntitilanteissa hyötyjen ja tarpeiden kuvaamisessa. [2.]

Lokidatan visualisointiin on paljon eri avoimen lähdekoodin ratkaisuja. Usein niiden käyttö saattaa kuitenkin vaatia lokien tallentamisen johonkin tiettyyn järjestelmään tai muotoon. Esimerkiksi Kibanan käyttö vaatii Elasticsearch-hakujärjestelmän käyttöä. Ne tarjoavat kuitenkin yleensä myös monia muita hyödyllisiä toiminnallisuuksia, joita voidaan käyttää järjestelmien analyysiin.



Kuva 5. Kibanassa muodostettu lämpökartta sivustolle tulleiden kutsujen lähdemaista.

Kuva 5 visualisoi sivustolle tulleiden kutsujen lähdemaita 30 päivän ajalta. Nämä on kuvattu pystyakselilla maatunnuksilla. Kibanassa voi muodostaa visualisoinneista oman kojelautansa kojelautoja voitaisiin esimerkiksi käyttää visualisoimaan sivuston liikennettä reaaliaikaisesti ja tekemään varoituksen halutussa tilanteessa. [3.]

3.3 Javan lokitusratkaisut

Laajasti käytettyjä Javan lokitusratkaisuja ovat Logback ja log4j2. Javan osana tulee myös Java Utilities Logging, mutta se ei ole läheskään yhtä käytetty kaupallisissa tuotteissa ja on merkittävästi hitaampi. Näitä kaikkia voidaan käyttää SLF4J-fasadin kautta, jolloin käytettyä lokituksen taustajärjestelmää on helppo vaihtaa ja käyttäminen voidaan toteuttaa samalla tavalla taustaratkaisuista riippumatta ainoastaan konfiguraatiota muuttamalla.

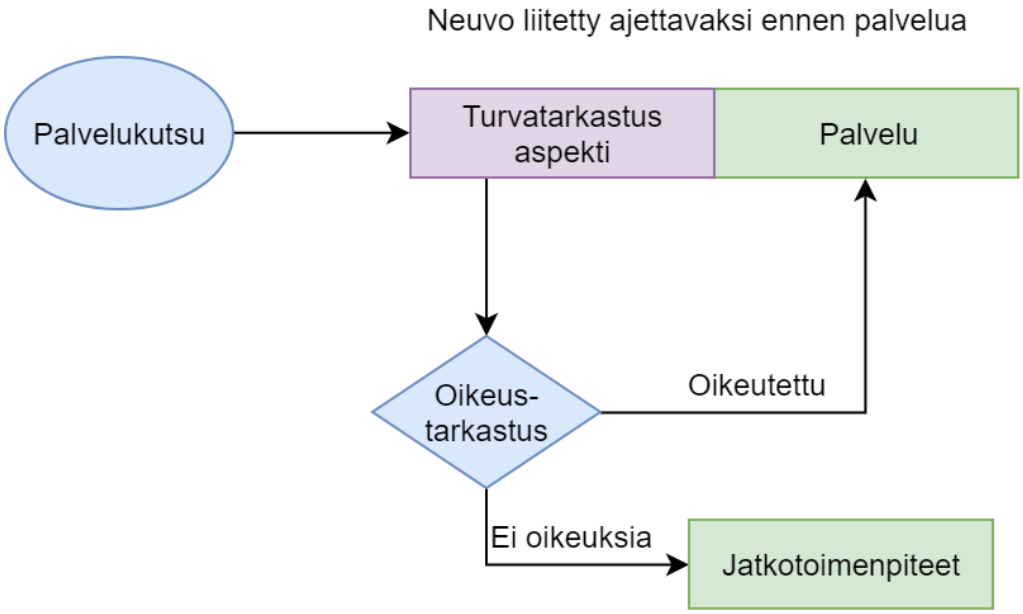
Javan käytetyt lokitusteknologiat ovat hyvin hajaantuneita ja ainoa yleisesti käytetty teknologia on SLF4J, joka mahdollistaa sovelluksessa käytettyjen lokitusratkaisujen yhtenäistämisen samaan lokitusratkaisuun. Sovelluksissa käytettiin jo ennestään Log4j2- ja SLF4J-teknologioita ja näistä ei nähty tarpeelliseksi siirtyä käyttämään jotain muuta ratkaisua. [4.]

4 Aspektipohjainen ohjelmointi

4.1 Yleistä

Ohjelmistokehityksessä tulee usein vastaan toiminnallisuuksia, jotka ovat sovelluksen toiminnallisuuden kannalta välttämättömiä mutta eivät suoraan liity sen liiketoiminnalliseen logiikkaan. AoP-ohjelmoinnissa tällaisia tarpeita kutsutaan läpileikkaaviksi tarpeiksi (Cross-cutting concerns). Näitä ovat esimerkiksi tietokantatransaktiot, lokitus ja auktorisointi.

Aspektipohjainen ohjelmointi on kehitetty vähentämään koodin hajautumista ja sotkeutumista irrottamalla ohjelman läpileikkaavat tarpeet keskitettyihin ratkaisuihin, mikä lisää ohjelmoinnin modulaarisuutta. Konseptina se kehitettiin 90-luvulla Xerox PARC-laboratoriossa ja samalla kehitettiin myös ensimmäinen AoP-kehys AspectJ. [5.]

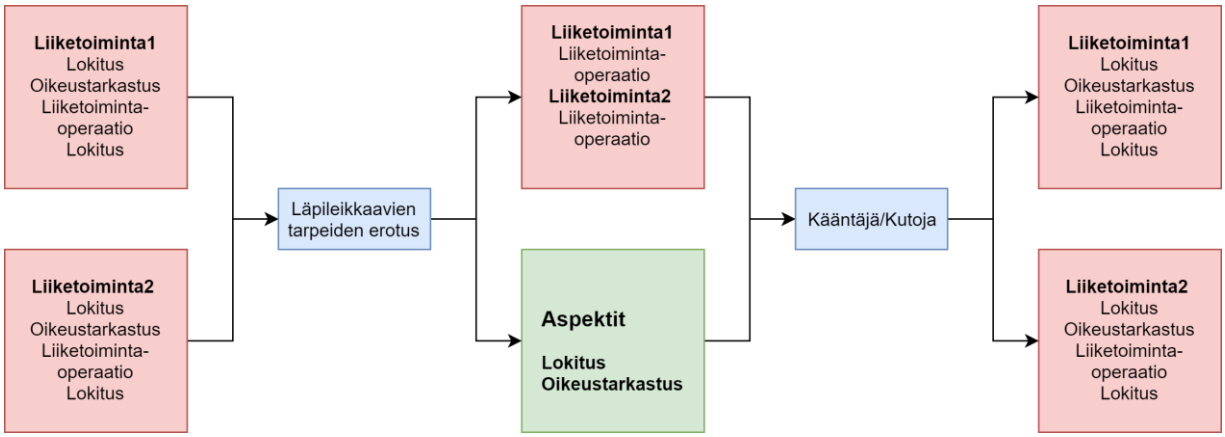


Kuva 6. Oikeustarkastus aspekteilla.

Esimerkiksi Kuva 6 sovellukseen tulevien palvelupyyntöjen turvatarkastukset on toteutettu ennen palvelua ajettavalla aspektilla, joka voi tarkistaa kutsujan oikeustason ja joko estää prosessin jatkumisen itse kontrolleriin tai päästää sen eteenpäin. Lisäksi aspekti voisi esimerkiksi lokittaa estetyt palvelupyynnöt erilliselle lokille. Tämä mahdollistaisi turvatarkastuksen erottamisen kokonaan itse kontrollerista.

Aspektipohjaisen ohjelmoinnin ideana on toteuttaa läpileikkaava tarve erillisenä toiminnallisuutenaan. Toiminnallisuuden eriyttäminen mahdollistaa koodin keskittämisen ja vähentää boilerplate-koodiin määrää muun toteutuksen seassa. Tällaista läpileikkaavan tarpeen toteuttavaa metodia kutsutaan AoP-ohjelmoinnissa neuvoksi. Neuvot kohdistetaan haluttuihin metodien liityntäkohtiin pointcuteilla, jotka kuvaavat halutun liityntätavan ja kohteet. Liityntäkohtia ovat esimerkiksi metodien kutsuminen, metodin aloitus ja metodin päättyminen. [5.]

Aspektipohjaisen ohjelmoinnin neuvoja voidaan rinnastaa esimerkiksi JUnit-testikehyksen @Before- ja @After-annotoituihin metodeihin, jotka ajettaisiin ennen ja jälkeen testin suorittamista. Vastaavasti neuvot voidaan kohdistaa suoritettaviksi ennen tai jälkeen metodin suorituksen. Erona kuitenkin on se, että neuvoja ei tarvitse määritellä samassa luokassa ja ne voidaan kohdistaa laajemmin moniin eri toiminnallisuuksiin myös muissa luokissa.



Kuva 7. Läpileikkaavien tarpeiden toteutus aspekteilla

Kuva 7 havainnollistetaan oikeustarkastuksen ja lokituksen erotus liiketoimintatarpeen toteuttavasta metodista omiin aspekteihinsa. Aspektit sisältävät neuvoja, jotka kudotaan liiketoiminnallisiin metodeihin erillisessä prosessissa. Tämän tarkoituksena on selkeyttää metodeja poistaen niistä muut läpileikkaavat tarpeet, jotka ovat pakollisia, mutta eivät kuitenkaan itsessään ole osa liiketoiminnallisesti kiinnostavaa prosessia.

Jos halutaan seurata AoP-ohjelmointiparadigmaa, on tärkeää tehdä muutoksia vain läpileikkaavien tarpeiden toteutukseen. Muutoin voidaan helposti rikkoa sekä OoP- ja AoP-

periaatteita toteuttamalla funktionaalisuutta, joka kuuluu itse metodin keskeisiin tarkoituksiin aspekteilla. Esimerkiksi sesonkialennus ostoksista voitaisiin toteuttaa muokkaamalla aspektilla tuotteiden hintaa, kun siirrytään maksuoperaatioon. Tämä voitaisiin toteuttaa muokkaamatta itse sovellusta kutomalla alennus jo käännettyyn sovellukseen. Kuitenkin tämän tyylinen toteutus olisi hyvin vaikea hahmottaa koodista ja vaikeuttaisi virheidenjäljitystä. Sen sijaan aspektiohjelmoinnilla pitäisi toteuttaa vain läpileikkaavia tarpeita, jotka eivät suoraan vaikuta prosessin liiketoiminnalliseen tarkoitukseen, vaan suorittavat tätä tukevia toimenpiteitä kuten transaktiointia ja oikeustarkastuksia.

AoP-ohjelmoinnissa neuvot, jotka toteuttavat läpileikkaavan tarpeen, kohdistetaan liityntäkohtiin pointcuteilla. Mahdolliset liityntäkohdat riippuvat käytetystä ohjelmointikehyksestä ja esim. Spring AoP tukee vain metodien suorittamista, kun taas AspectJ tarjoaa paljon muitakin vaihtoehtoja. Näistä selvästi yleisimmin käytetty on kuitenkin juurikin metodin suoritus. [6.]

4.2 Liittyminen

Neuvojen yhdistäminen liityntäkohtiin tehdään liittämällä neuvoon yksi tai useampi pointcut, joka ohjeistaa, mihin liityntäkohtiin neuvo yhdistetään. Näin yhdistetään läpileikkaavan tarpeen toteuttava neuvo liiketoiminnallisiin luokkiin. Tämä voidaan toteuttaa joko viittaamalla annotaatioon, joka lisätään luokkiin tai metodeihin, joihin neuvo kohdistetaan, tai viittaamalla luokkaan tai metodiin säännöllisellä lausekkeella. Näitä tapoja voidaan myös yhdistellä. Annotaatioita käytetään liittämällä ne kohdeluokkaan tai -metodiin ja niihin voidaan myös asettaa arvoja, joita neuvova aspekti voi käyttää toiminnallisuutensa ohjaamiseen. Tämä mahdollistaa hyvin suoraviivaisen yhdistämisen, jos neuvoa halutaan käyttää hyvin vaihtelevissa kohteissa tai jos neuvolle täytyy antaa ohjaavia argumentteja.

```
package com.demo.aspects;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention (RetentionPolicy.RUNTIME)
public @interface AdviceIgnore{}
```

Esimerkkikoodi 1. Java-annotaation julistava luokka.

Esimerkkikoodissa 1 on kuvattu annotaation “@AdviceIgnore” määrittely. Sen kohteiksi on asetettu metodit, joten sitä ei voida liittää esimerkiksi luokan määrittelyyn. Ne on myös määritelty sisällytettäviksi ohjelman ajovaiheessa. Myös “@Target” ja “@Retention” ovat Javan annotaatiota. Annotaatiot itsessään eivät tee mitään, vaan jotta niitä voidaan käyttää, pitää ne lisätä johonkin kohteeseen. Tämän lisäksi toteuttaa jonkinlainen niitä käyttävä funktionalisuus esimerkiksi aspekteilla.

Säännöllisillä lausekkeilla voidaan viitata metodiin, luokkaan tai kenttään, ja ne mahdollistavat hyvin laajan toiminnallisuuden neuvomisen muutamalla koodirivillä. Kuitenkin neuvot voivat myös kohdistua ei tarkoitettuihin luokkiin tai metodeihin, jos niitä ei ole suunniteltu tarkasti.

```
@Pointcut(“execution(public * com.demo.web.controller.*.* (..))”)
private void controllerCall() {}


@Pointcut(”@annotation(com.demo.aspects.AdviceIgnore)”)
private void adviceIgnore() {}
```

Esimerkkikoodi 2. Pointcutteja jotka kohdistuvat controller-paketin luokkien public-metodeihin ja kaikkiin JSON-paketin metodeihin. Lisäksi annotaatioon on kohdistettu pointcut.

Esimerkkikoodi 2 on kuvattu kolme erilaista pointcuttia. Ensimmäinen näistä kuvaa pointcuttia, joka on kohdistettu käyttämällä projektin luokkarakennetta. Rivi sisältää useita kohdistukseen käytettyjä määrityksiä:

- Ensimmäiseksi annetaan tieto poinctutin kohdistuksen tyypistä. “execution” kertoo, että kyseessä ovat kohdealueen metodit.
- Toinen kohta, jossa on annettu arvo “public”, määrittää kohteiden näkyvyysasteen. Arvona voidaan käyttää myös “*”, jolloin näkyvyysasteesta ei välitetä.
- Tämän jälkeisessä kohdassa voitaisiin määritellä kohdistettujen metodien paluuarvon tyyppi. Tässä tapauksessa on annettu arvoksi “*”, jolloin tästä ei välitetä.
- “com.demo...” määrittelee paketin, luokan ja metodin jonka sisällä olevaan toiminnallisuuteen pointcut osoittaa. Tässä tapauksessa pointcut osoittaisi kaikkiin projektin demo/web/controller/-kansion sisällä oleviin luokkiin, sekä niiden metodeihin.
- Viimeisessä kohdassa voidaan määritellä metodin saamien parametrien tyyppejä, voidaan esimerkiksi määritellä, että metodin parametrina saa olla

ainoastaan String. Annetulla arvolla “(..)” metodien parametreista ei välitetä.

Viimeinen “adviceIgnore()” kohdistuu kaikkiin paikkoihin, joihin on lisätty kyseinen annotaatio “@AdviceIgnore”. [7.]

Yhdistelemällä säännöllisiä lauseita ja annotaatioita voidaan neuvo kohdistaa hyvinkin tarkasti haluttuihin luokkiin. Varsinkin säännöllisillä lausekkeilla voidaan kohdennus toteuttaa helposti ja kattavasti minimaalisella työllä. Kuitenkin tämä tuo riskin, että uudelleen nimeämällä tai nimeämällä luokkia käytetyn konvention ulkopuolella luokat tai metodit jäävät neuvojen ulkopuolelle. Tätä voidaan vältellä konventioilla ja tekemällä kohdistus esim. pakettiin. Tämä ei myöskään ole läpinäkyvää itse kohteissa, joten on tärkeää ylläpitää näkyvyys projektin kehittäjille erilaisiin neuvoihin, joita sovellukseen on sisälletty.

4.3 Neuvo

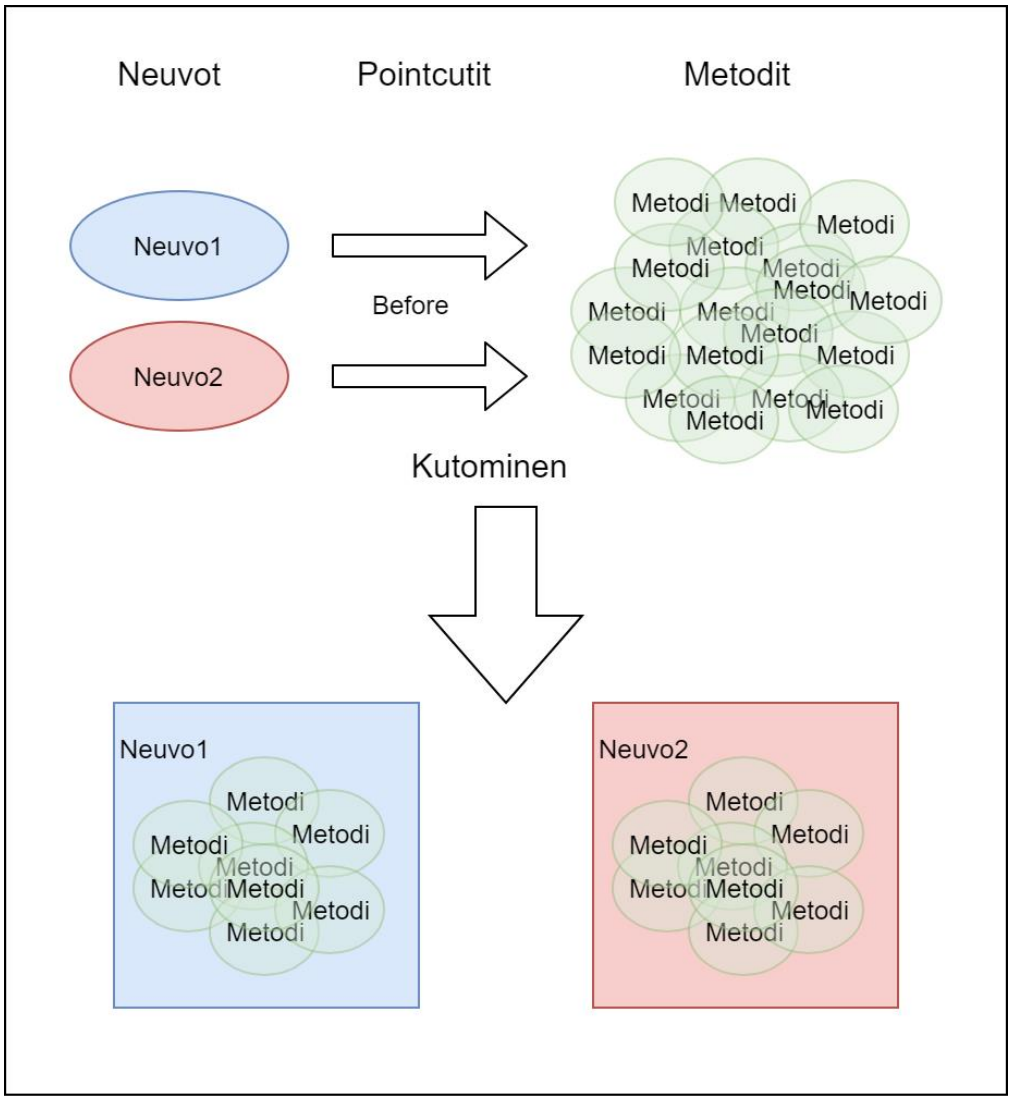
```
@Around(controllerCall() && !adviceIgnore() {}
public Object aroundService(final ProceedingJoinPoint point) throws Throwable
{advising code}
```

Esimerkkikoodi 3. Neuvo, joka kohdistuu kaikkiin com.demo.web.controller-paketin luokkien public-metodeihin, paitsi jos metodiin on liitetty @AdviceIgnore-annotaatio.

Esimerkkikoodi 3 kohdistetaan neuvo yhdistämällä pointcutit controllerCall() ja adviceIgnore(), jotka oli kuvattu aikaisemmin esimerkkikoodissa kaksi. Neuvon koodi suoritetaan metodien ympärillä. Tämä tarkoittaa neuvon suorittamista ennen ja jälkeen metodin suoritusta. Neuvon sisällä pitää tässä tapauksessa määritellä paluukohta, jonka kohdalla varsinainen neuvottu metodi suoritetaan.

Neuvot ovat ajatuksellisesti hyvin lähellä Decorator-suunnittelumallia. Ne lisäävät haluttua toiminnallisuutta erillisestä metodista jo olemassa olevan toiminnallisuuden päälle, toiminnallisesti ne kuitenkin kohdistuvat metodeihin eivätkä koko luokkaan. [8.] AoP-ohjelmoinnissa ei myöskään tarvitse käyttää perintää tai interfaceja, mutta tämä riippuu käytetystä AoP-toteutuksesta.

Neuvo on halutun funktionaalisuuden toteuttava metodi, joka liitetään haluttuihin paikkoihin pointcuteilla. Se koostuu itse neuvovasta metodista halutuista kohdistavista pointcuteista ja tiedosta, missä vaiheessa kohdeprosessia neuvo suoritetaan. Neuvo kohdistetaan yleisesti joko metodin alkuun, perään tai ympärille. Tämä mahdollistaa esimerkiksi helpon tavan kohdistaa oikeustarkastukset ennen suoritettavia metodeja, tai lokittaa metodien paluuarvoja.

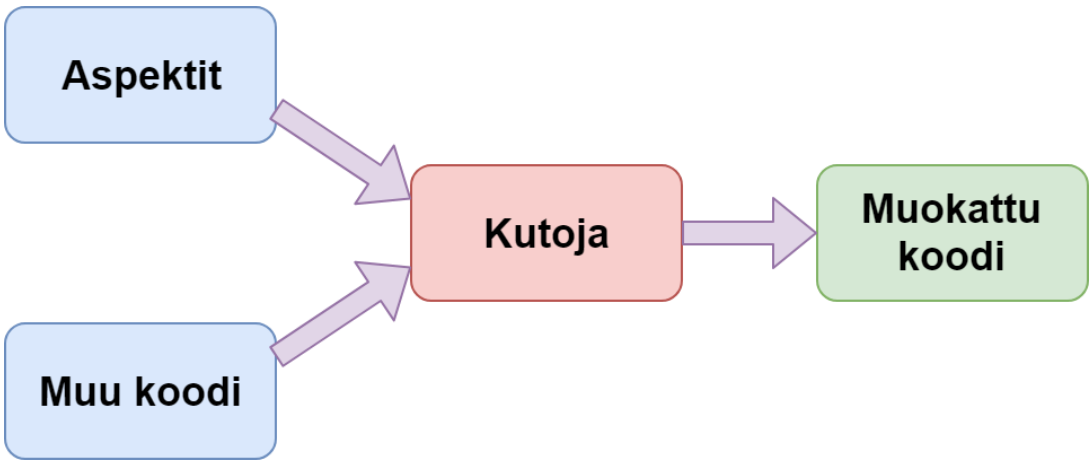


Kuva 8. Neuvojen lisääminen metodeihin.

Kuva 8 havainnollistaa neuvojen liittymistä kohdemetodeihin. Neuvojen koodi liitetään pointcuttien määrittelemiin metodeihin ajettavaksi ennen niiden suorittamista. Tämä prosessi tehdään kutojassa, jonka jälkeen neuvojen koodi on lisätty osaksi metodien suoritusta.

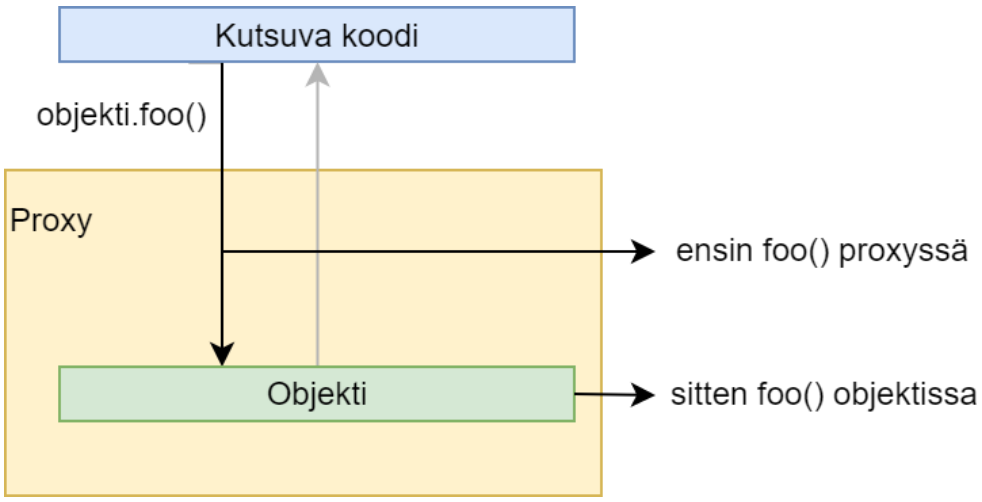
4.4 Kutominen

Kutomisella tarkoitetaan prosessia, jossa aspektit liitetään pääohjelmaan tuottaen uusi yhdistetty ohjelma kuten on havainnollistettu Kuva 7. Tämä voidaan tehdä joko ajon aikana luokkien latauksen yhteydessä tai tätä ennen joko ohjelman kääntämisen aikana tai jo käännettyyn ohjelmaan. Tämä mahdollistaa aspektien lisäyksen ohjelmaan riippumatta siitä, onko lähdekoodi saatavilla mahdollistaen diagnostiikan ja erinäisten kontrollien lisäämisen jälkikäteen. [9.]



Kuva 9. Kutominen lisää aspektit ohjelmakoodiin.

Kuva 9 prosessissa kutoja lisää aspektien koodin muun koodin joukkoon. Tämän prosessin toteutus riippuu käytetystä AoP-kehyksestä. Esimerkiksi Spring AoP -kutoja tekee sen generoimalla neuvotuista luokista proxy-luokat, joissa aspektin koodi sijaitsee. Kuva 10 havainnollistaa tätä. Kun objektin metodia kutsutaan, ajetaan ensin neuvon toteuttava koodi objektin proxynä toimivassa luokassa. Tämän jälkeen kontrolli palautetaan objektin metodille. [10; 11.]



Kuva 10. Neuvova aspekti ajetaan objektin kutsun yhteydessä.

5 Teknologiat

5.1 AspectJ-ohjelmointikehys

AspectJ oli ensimmäinen AoP-mallin toteuttava ohjelmistokehys ja on AoP:n tosiasiallinen standarditoteutus, jonka semantiikkaa ja toimintatapoja käytetään laajasti myös muissakin AoP-toteutuksissa. AspectJ toteuttaa kutomisen omalla kääntäjällään AJC:lla. Tämä tukee kolmea eri kutomistapaa: [12.]

- Compile-time weaving kutoo aspektit muun koodin joukkoon Javan peruskääntöprosessin aikana. Jos aspektit ovat vaadittuja järjestelmän kääntämistä varten, tämä on ainoa mahdollinen tapa. Kudonnan kohteena olevan koodin täytyy olla lähdekoodimuodossa.
- Post-compile weaving kutoo aspektit jo käännettyihin luokkatiedostoihin tai JAR:eihin. Tämä mahdollistaa kutomisen kohteisiin, joiden lähdekoodiin ei ole pääsyä.
- Load-time weaving kutoo aspektit, kun luokka ladataan ja määrittellään JVM:lle.

AspectJ:n käyttöönotto on hyvin suoraviivaista ja yleisesti projektiin pitää lisätä vain AspectJ-kirjastot sekä kutomisaskel projektin rakennusvaiheisiin.

```
...
<plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>aspectj-maven-plugin</artifactId>
      <version>1.11</version>
      <configuration>
            <forceAjcCompile>true</forceAjcCompile>
```

```
            <weaveDirectories>
                  <weaveDirectory>${build.directory}/classes</weaveDirectory>
            </weaveDirectories>
      </configuration>
      <executions>
            <execution>
                  <phase>process-classes</phase>
                  <goals>
                        <goal>compile</goal>
                  </goals>
            </execution>
      </executions>
      <dependencies>
            <dependency>
                  <groupId>org.aspectj</groupId>
                  <artifactId>aspectjrt</artifactId>
                  <version>${aspctj.version}</version>
            </dependency>
            <dependency>
                  <groupId>org.aspectj</groupId>
                  <artifactId>aspectjtools</artifactId>
                  <version>${aspctj.version}</version>
            </dependency>
      </dependencies>
</plugin>
...
```

Esimerkkikoodi 4. AspectJ-kutomisen lisääminen Mavenin rakennusprosessiin pom.xml:ssä

Esimerkkikoodi 4 kuvaa käytetyn Mavenin rakennusvaiheen, joka määrittelee AspectJ-aspektien kutomisen tehtäväksi sen jälkeen, kun class-tiedostot on muodostettu osana käännösprosessia. Lisäksi määritellään käytetyt kirjastoversiot. Riippuen siitä, miten kutominen halutaan tehdä, “<execution>”-elementin tietoja voidaan muokata. Työn aikana kävi ilmi, että kutominen piti tehdä nimenomaan class-tiedostoihin, koska Lombok lisäsi tarvittavat tiedot ennen tätä. Lombokin ja AspectJ:n konfliktista johtuen, jos kutominen yritettiin tehdä ennen class-tiedostoiksi kääntämistä, niihin ei muodostettu ollenkaan Lombokin tuottamaa koodia. Tästä johtuen olioiden kapseloituja arvoja ei voitu käyttää.

5.2 Spring-ohjelmointikehys

Usein sovelluksia kehitettäessä joudutaan ratkaisemaan samoja ongelmia kuten esimerkiksi käyttäjien sisäänkirjautumisen ja sessioiden hallintaa. Spring on Javan ohjelmointikehys, joka tarjoaa valmiita ratkaisuja näiden toteuttamiseen ja mahdollistaa kehitystyön paremman keskittymisen toteuttamaan ohjelman varsinaista tarkoitusta. [13.]

Spring koostuu useista modulaarisista ratkaisuista eri tarpeiden toteuttamiseen. Erilaisia moduuleja ovat esimerkiksi Spring Batch, joka tarjoaa valmiita ratkaisuja eräajojen ohjelmointiin ja Spring Securityn, joka tarjoaa autentikointiin ja oikeustarkastuksiin yleisen ratkaisun. Näitä käyttämällä verkkosovelluksen kehittämisessä voidaan keskittyä enemmän itse liiketoiminnallisuuden toteuttamiseen. Spring käyttää hyvin paljon AoP-ratkaisuja erilaisten läpileikkaavien tarpeiden toteuttamiseen, AoP mahdollistaa osaltaan Springin ratkaisujen yksinkertaisen lisäämisen haluttuihin toiminnallisuuksiin. [13.]

```
@Controller
public class HenkiloController {

@Autowired
private LaskuService laskuService;

@PreAuthorize("@securityService.hasPermission('katsele.laskut')")
@RequestMapping(value = "/henkilo/{henkiloId}/laskut"
public String laskut(@PathVariable("henkiloId) Long henkiloId, ModelMap model)
{
      List<Lasku> laskut = laskuService.haeHenkilonLaskut(henkiloId);
      model.put("laskut", laskut);
      return "henkilo/laskut"
}
```

Esimerkkikoodi 5. Kontrolleri, joka hakee henkilön laskut ja palauttaa sivun, jolla laskut näytetään. Lisäksi kontrolleriin on lisätty oikeustarkastus annotoimalla se.

Esimerkkikoodi 5 kuvattu luokka ei itse ota kantaa laskuServicen hakemiseen, vaan Spring injektoi sen @Autowired-annotaation takia. Se, mikä LaskuService-luokka on kontrollerin käytössä, määritetään Springin konfiguraatioissa. Tämä mahdollistaa sen helpon vaihtamisen. Lisäksi Spring käyttää useimmiten Singletoneja palveluluokille vähentäen käytettyjä resursseja.

5.3 Lombok-kirjasto

Usein Java-objektien luokat sisältävät merkittävältä osaltaan boilerplate-koodia kuten gettereitä ja settereitä. Tämä vaikeuttaa varsinaisten merkittävien metodien erottumista. Lombok on Java-kirjasto, joka yrittää vastata tähän puutteeseen.

Lombok-kirjasto muodostaa Javan käännösprosessin yhteydessä valitut metodit kuten setterit, getterit, konstruktorit ja toStringit vähentäen rivien määrää johtaen selkeämpiin

luokkiin ja mahdollistaen paremman keskittymisen itse luokan liiketoiminnallisiin tarkoituksiin. Tämä tarkoittaa myös, että luokkaa muokatessa ei tarvitse editoida erikseen kyseisiä funktionaalisuuksia. Lombok tarjoaa myös monia muita vastaavia helpotuksia. [14.]

```
@Data
public class Oppilas {
      private String nimi;
      private int ika;
      private List<Kurssi> kurssit;

      public void merkitseKurssiSuoritetuksi(String kurssiKoodi) {
            Toiminnallisuus…
      }
}
```

Esimerkkikoodi 6. Lombokin käyttö poistaa tarpeen lisätä ei-mielenkiintoista koodia luokkiin generoiden nämä automaattisesti annotaatioista. Tämä selkeyttää luokkaa ja tekee selkeämmäksi sen liiketoiminnallisesti kiinnostavat luokat.

Edeltävä koodi kuvaa luokkaa, johon on lisätty Lombokin tarjoama annotaatio toiminnallisuuden generoimiseksi. “@Data”-annotaation johdosta luokkaan lisätään käännösvaiheessa getterit, setterit, toString-metodi ja muutamia muita perustoiminnallisuuksia. Lombokin avulla voidaan toistuvan peruskoodin määrää vähentää merkittävästi. Peruskoodi voidaan generoida yleisesti myös useimmissa ohjelmointiympäristöissä, mutta tämä lisää luokkiin paljon ylimääräistä koodia, joka ei ole yleisesti kiinnostavaa.

5.4 MDC-teknologia

Usein jonkin prosessin kulkiessa sen eri metodien läpi halutaan näissä kaikissa käyttää jotain tiettyä tietoa, joka ei kuitenkaan ole merkittävä metodin liiketoiminnallisen tarkoituksen kannalta. Näin voitaisiin tehdä siirtämällä tietoja metodista toiseen parametreina, mutta tällainen toteutus johtaisi helposti parametrimäärien merkittävään kasvuun, mikä johtaa koodin luettavuuden heikentymiseen.

MDC on teknologia, jossa säikeen kontekstiin tallennetaan jotain tietoa, jonka halutaan olevan kaikkien prosessin suorittavien metodien käytettävissä. Tämä on hyvin tehokasta lokituksessa mahdollistaen tunnistamistietojen tallentamisen prosessin käynnistämisen yhteydessä ja saman tiedon käyttämistä läpi prosessin riippumatta metodista tai luokasta. Esimerkiksi asiakastietojen päivityskutsun yhteydessä voidaan kontekstiin lisätä

käyttäjänimi, IP-osoite, kutsuhetki, käytetty palvelin, uniikkitunniste tms. Nämä voidaan sen jälkeen lisätä mihin tahansa haluttuun lokiviestiin lisäämällä kontekstiin tallennetun tiedon tunniste lokituksen konfiguraation viestimalliin. Tiedon käyttämisestä lokituksessa kuvaavat Esimerkkikoodi 7 ja Esimerkkikoodi 8. Näitä tietoja voidaan myös käyttää haluttaessa itse metodin sisällä, esimerkiksi jos käyttäjänimi halutaan tallentaa tietokantaan muokkauksen yhteydessä. Tämä helpottaa prosessin ketjun tunnistamista ja esimerkiksi jos kontekstiin on lisätty uniikkinumero tällä numerolla, voidaan löytää kaikki prosessin lisäämät lokirivit yksinkertaisella haulla, myös useammista lokitiedostoista, mikä mahdollistaa laajemman viestien jaottelun useampiin tiedostoihin. [15.]

```
@Around(controllerCall() && !adviceIgnore() {}
public Object aroundController(final ProceedingJoinPoint point) throws
Throwable {
      MDC.put(“UUID”, UUID.randomUUID().toString();
      MDC.put(“USER”, SecurityContext.getUser().getUsername();
      // Jatketaan metodin suoritukseen.
      final Object retVal = point.proceed();
      //Tyhjennetään kontekstin MDC
      MDC.clear();
}
```

Esimerkkikoodi 7. Neuvo, joka suoritetaan kontrollerikutsun yhteydessä.

Esimerkkikoodi 7 kuvaa aspektia, joka suoritetaan kontrollerimetodien kutsujen ympärillä. Aspekti lisää säiekontekstiin operaation kutsujan käyttäjänimen sekä generoidun uniikin tunnisteen. Metodin suorituksen jälkeen kontekstiin tallennetut tiedot poistetaan. Esimerkkikoodi 8 on kuvattu Log4J2-konfiguraatiossa käytetty lokirivien pattern. Se käyttää tallennettuja tietoja jokaisessa patternia käyttävässä lokitapahtumassa. Tällä tavoin kaikissa lokitustapahtumissa prosessin aikana voidaan käyttää tunnistetietoja ilman niiden erillistä antamista parametreissa.

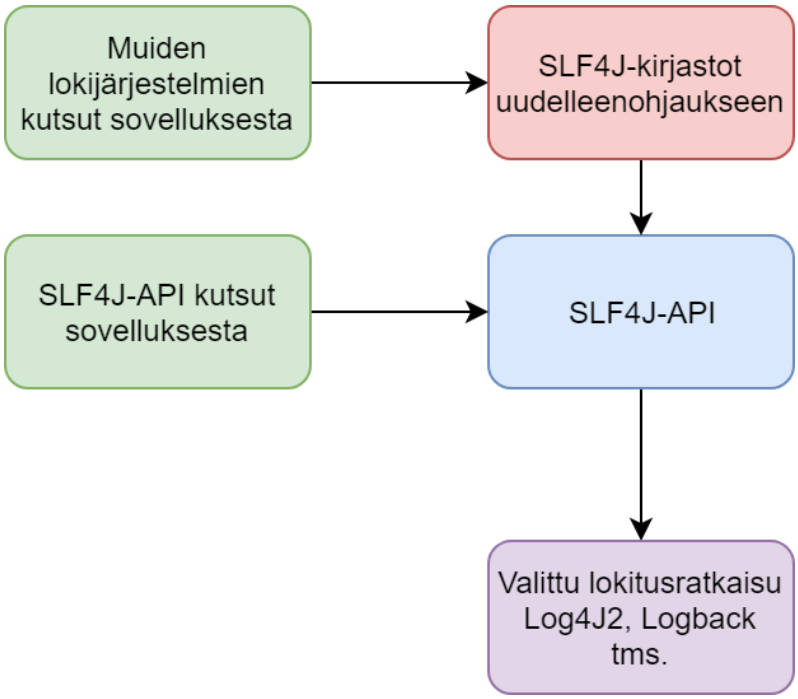
```
<Property name=”PATTERN”>UUID:[%X{UUID}] USER:[%X{USER}] - %m%n</Property>
```

Esimerkkikoodi 8. XML-muotoinen pattern-konfiguraatio lokitukseen, joka lisää sitä käyttäviin lokiriveihin UUID- ja USER-kentän arvot ennen varsinaista viestiä.

Tämä teknologia helpottaa merkittävällä tavalla lokien seuraamista monikäyttäjäympäristöissä, joissa lokirivejä voidaan tuottaa massiivisia määriä eri tiedostoihin ja kronologisen prosessin seuraaminen ei välttämättä ole yksinkertaista. Jos prosessit, joita kutsutaan, luovat säikeitä, ne eivät välttämättä peri kontekstia. Näissä tapauksissa kontekstin tiedot täytyy siirtää erikseen aliprosessiin sen käynnistyksen yhteydessä.

5.5 SLF4J-ohjelmointikehys

SLF4J-fasadi on Javan yleisesti käytetty fasadi, joka helpottaa lokituksen taustajärjestelmien vaihtamista. SLF4J-projekti tarjoaa myös työkalut taustajärjestelmän migraatioon riippumatta pääsystä lähdekoodiin. Tämä on toteutettu lähdekoodikonvertoijalla ja Java-paketeilla, jotka toteuttavat uudelleenohjauksen, jos käytetty järjestelmä oli Log4J, Java Utilities Logging tai Jakarta Commons Logging. Sen sijaan, että käytettäisiin suoraan esim. Log4J2- tai Logback-ohjelmistokehyksiä, voidaan niitä käyttää SLF4J:n kautta, jolloin taustajärjestelmää voidaan vaihtaa nopeasti muutaman rivin muutoksella. Lisäksi, jos ohjelma on muodostettu monista erillisistä paketeista, nämä voivat käyttää hyvin eriäviä lokitusratkaisuja. Tällöin pakettien käyttämä lokitus voidaan yleensä kaikki ohjata keskitettyyn järjestelmään ilman, että itse paketteja täytyy muokata tai laajentaa. [16.]



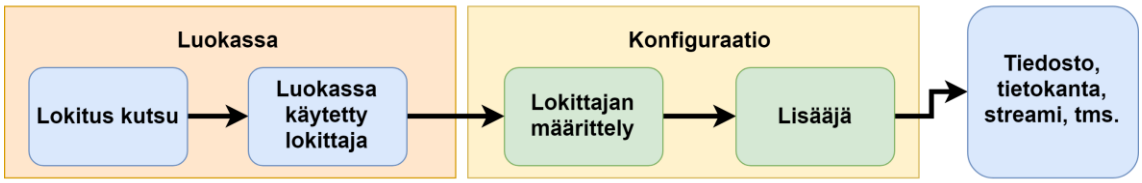
Kuva 11. Lokituksen ohjaaminen käyttämään SLF4J-kehystä

Kuva 11 havainnollistetaan lokituksen ohjaamista käyttämään SLF4J-kehystä, SLF4J tarjoamia kirjastoja sovellukseen lisäämällä voidaan esimerkiksi Java Utilities Logging-kehystä käyttävä lokitus uudelleenohjata suoraan SLF4J-APIin. Suoraa siirtoa ei kuitenkaan suositella, jos toteutuksen lähdekoodi on saatavilla. Tällaisiin tilanteisiin SLF4J-

projekti tarjoaa migraatiotyökalun, jonka avulla iso osa manuaalisesta työstä voidaan tehdä automaattisesti. Työkalun käytön lisäksi tarvitsee kuitenkin tehdä jonkin verran manuaalista työtä. Uudelleenohjauksen tekevät kirjastot on tarkoitettu tilanteisiin, joissa osana projektia käytetään muita kirjastoja, joita ei voida muokata, mutta kirjastojen tekemä lokitus haluttaisiin kuitenkin ohjata yhteiseen järjestelmään. [17.]

5.6 Log4J2-ohjelmointikehys

Log4J2 on Log4J:n seuraaja ja on yleisesti käytetty lokituksen ohjelmointikehys.



Kuva 12. Log4J2:n lokitusprosessi. Tämä vastaa pääosiltaan Logback-ohjelmistokehystä.

Sen toimintatapa, jota havainnollistetaan Kuva 12, on hyvin samanlainen kuin sen kilpailijoilla. Se koostuu konfiguraatiosta, jossa määritellään lokituksen toiminnallisuus, sekä itse lokitustapahtumista, jotka sijaitsevat sovelluksen metodeissa.

Konfiguraatiossa määritellään käytössä olevat tulostuskohteet ja niiden toiminnallisuus kuten arkistointitapa, maksimikoko, käytetyt viestipatternit, lokitustasot sekä lokittajien kohteet ja prioriteettijärjestys.

```
<RollingFile
      name="webserviceAppender"
      fileName="${LOG_DIRECTORY}/webserviceLog.log"
      filePattern="${LOG_DIRECTORY}/Archive/$${date:yyyy-MM}/webserviceLog
      -%d{MM-dd-yyyy}-%i.log.gz">
      <PatternLayout>
            <Pattern>${PATTERN}</Pattern>
      </PatternLayout
      <Policies>
            <TimeBasedTriggeringPolicy"/>
            <SizeBasedTriggeringPolicy size="100 MB" />
      </Policies>
      <DefaultRolloverStrategy>
            <Delete basePath="${LOG_DIRECTORY}" maxDepth="2">
                  <IfFileName glob="*/webserviceLog-*.log.gz" />
                  <IfLastModified age="30d" />
```

```
            </Delete>
        </DefaultRolloverStrategy>
</RollingFile
```

Esimerkkikoodi 9. Appender, joka määrittää lokitiedoston.

Esimerkkikoodi 9 kuvaa lokitiedoston määrittelyn Log4J2-konfiguraatiossa. Tiedoston sijainti voidaan määrittää konfiguraatiossa ja ottaa esim. JVM-argumenteista palvelinta käynnistettäessä. Lokitiedosto on määritetty arkistoitavaksi, kun sen koko kasvaa yli 100 megatavun, tai päivän vaihtuessa. Arkistointi tehdään pakkaamalla loki omaksi tiedostokseen filePatternin määrittelemällä tavalla. “<Pattern>”-konfiguraatio määrittelee lokiviestien muodon. Tässä tapauksessa käytetään aikaisemmin määriteltyä patternia kuten Esimerkkikoodi 8. Lisäksi arkistoidut lokitiedostot poistetaan, jos niitä ei ole muokattu 30 päivään.

6 Toteutus

Työn ensimmäisessä aloituspalaverissa sovittiin alustava aikataulu ja askeleet, joiden pohjalta työ tehtäisiin. Nämä olivat:

- Analysoidaan olemassa oleva toteutus, käytetyt teknologiat, lokituotteet ja tuotteiden sisältö.
- Muodostetaan lista halutuista tuotteista.
- Kartoitetaan ratkaisuvaihtoehdot ja niiden hyvät ja huonot puolet.
- Valitaan ratkaisu ja tarkennetaan listaa tuotteista.
- Toteutetaan alustava ratkaisu sovelluksiin.
- Muokataan toteutettua ratkaisua kehitysversion palautteen ja havaittujen puutteiden pohjalta.
- Dokumentoidaan lopputuotteet ja toiminnallisuus sekä käydään nämä läpi ohjelmoijien kanssa.
- Tehdään mahdollinen jatkokehitys.

6.1 Analysointi ja ratkaisuvalinta

Työ aloitettiin analysoimalla molempien sovelluksien olemassa oleva lokitus. Tätä tehtäessä kävi nopeasti ilmi, että minkäänlaista systemaattista ratkaisua tai ohjelmointitapaa

ei ollut käytetty. Sovelluksista löytyi suoria konsoliin tulostuksia, eri tasoisia lokiviestejä, viestien sanomat olivat joko yhdentekeviä tai muuten mitäänsanomattomia.

Tämän lisäksi, vaikka sovellusten arkkitehtuuri oli hyvin samanlainen, niiden lokituksen lopputuotteet olivat hyvin eriäviä sisällöllisesti, ja tarkoitukseltaan samanlaiset viestit saattoivat mennä eri tiedostoihin sovelluksissa. Lisäksi konfiguraatiossa oli muutamia virheitä, jotka osittain estivät lokitusta.

Tämän pohjalta lokituotteiden määrää leikattiin merkittävästi ja niitä yhdistettiin kokoaviin tiedostoihin. Lisäksi päätettiin, että haluttiin lokittaa mahdollisimman laajasti sovelluksen service- ja controller-kutsuja virheenjäljityksen helpottamiseksi. Päädyttiin rakenteeseen, jossa lokitettaisiin seuraavat tuotteet:

- Järjestelmäloki, johon laitettaisiin kaikki järjestelmän sisäiset viestit ja muut viestit, joilla ei ole tarkempaa paikka, lisäksi aspektien tekemä lokitus menisi tähän tiedostoon.
- Integraatioloki, johon laitettaisiin tietoja muista sovelluksista tehtyjen kutsujen käsittelystä.
- SOAP-loki, johon laitettaisiin SOAP-pyynnöt ja vastaukset sovelluksen kommunikoidessa muiden asiakkaan järjestelmien kanssa.
- Autentikointiloki, johon laitettaisiin viestit käyttäjien tunnistautumisesta ja muista oikeuksiin liittyvistä tapahtumista.
- 3 lokia, joihin eriteltäisiin liiketoiminnalliset, toimintaa tukevat ja eräajojen käyttämät SQL-kutsut.
- Lisäksi jokaisella eräajolla oli jo oma lokinsa mutta näiden sijainti muutettiin, jotta kaikki lokit olisivat yhdessä kansiossa.

Myöhemmin tuotteiden määrä kasvoi vielä kahdella lokilla, jotka sisälsivät tietoja siitä, kuka oli katsonut VIP- ja normaalihenkilöiden tietoja. Yleisesti tämä kaikki vähensi merkittävästi tiedostojen määrää ja selkeytti tietojen löytämistä lokeilta. Myös lokien automaattista varmuuskopiointia muutettiin ylimääräisten kansioiden vähentämiseksi.

Tämän jälkeen kartoitettiin mahdollisia ratkaisuvaihtoehtoja. Kartoituksessa selvästi yksinkertaisimmaksi ratkaisuksi nousi AoP-pohjainen lokitus. Toisessa sovelluksista oli jo yksinkertainen järjestelmävirheiden lokitukseen tarkoitettu ratkaisu, joka käytti aspekteja. Lisäksi AoP-pohjainen lokitus tarjosi nopean ja yksinkertaisen tavan lokittaa laaja-alaisesti haluttua toiminnallisuutta vähäisellä työmäärällä. Muutos päätettiin toteuttaa käyttämällä aspekteja. AoP-pohjaisen ratkaisun eduiksi katsottiin:

- Vähätöisyys, hyvin minimaalisella koodimäärällä voitiin kattaa merkittävä osa toteutuksesta.
- Keskitetty ratkaisu, mihinkään luokkiin ei tarvinnut koskea ja koko lokitus voitiin toteuttaa yhdellä hyvin pienellä luokalla.
- Kestävyys, uusiin metodeihin tai luokkiin ei tarvinnut sisällyttää minkäänlaista koodia lokituksen lisäämiseksi, mikä poisti mahdollisuuden, että kehittäjä unohtaisi tehdä sen. Tämä oli totta niin kauan, kun luokissa ja paketeissa käytettiin jo olemassa olleita nimeämiskonventioita.

Näistä syistä AoP otettiin pohjaksi valitulle ratkaisulle.

6.2 Implementaatio

Toteutus aloitettiin muokkaamalla lokituksen outputkonfiguraatiota, mikä poistaa ylimääräisiä lokitiedostoja, ja yhdistää lokittajien kohteita. Lisäksi konfiguraatiota muutettiin niin, että eri kohteiden lokitustasot määriteltiin JVM-argumenteilla ja muita kovakoodattuja arvoja korvattiin erinäisillä muuttujilla, joiden arvot asetettiin erillisessä konfiguraatiotiedostossa, joka oli ympäristökohtainen.

Tämän jälkeen tehtiin perusaspektit kontrollereille ja palveluille toisessa sovelluksessa ja testattiin näiden kohdistamista. Yleisesti päädyttiin ratkaisuun, jossa kaikkia UI:lta tulevia kontrolleri- ja JSON-kutsuja käsiteltäisiin omalla aspektillaan, ja lisäksi kaikkiin palveluluokkien julkisiin metodeihin kohdennettaisiin oma aspektinsa. Tämä jättäisi lokitukseen ulkopuolelle apumetodit ja objektien metodit, joiden katsottiin kuitenkin olevan tarpeettomia. Jos ne haluttaisiin ottaa mukaan, muutos olisi kuitenkin hyvin yksinkertainen. Palveluaspektin kohdennukseen lisättiin kuitenkin myös annotaatio, jolla mikä tahansa metodi, joka olisi peruspalvelukohdennuksen ulkopuolella, voitaisiin myös lokittaa lisäämällä siihen annotaatio.

Kontrollerimetodeihin kohdistuva aspekti oli toteutettu niin, että se asetti operaationtunnisteen ja kutsujatiedon säikeen kontekstiin. Kontekstissa olleita tietoja käytettiin tämän jälkeen kaikissa ohjelman sisäisissä lokitustapahtumissa, mikä vaatii ainoastaan kontekstitietojen käyttämisen konfiguroinnin Log4J2-patternissa. Lisäksi kontrolleriaspekti lokitti myös kontrollerin saamat parametrit sekä kutsun suorittamiseen kuluneen ajan. Palveluaspekti lokitti metodien suoritusaikoja sekä niiden kutsu- ja paluuparametrit.

Tämän prosessin suurin haaste oli sovellusten hajaantunut konfiguraatio ja sen puutteellinen dokumentaatio sekä vähänkään kompleksimpien konfiguraatioesimerkkien puute AspectJ- ja Spring-ohjelmistokehyksille.

Lisäksi AspectJ:n ja Lombokin yhteistoiminta Maven-projektissa osoittautui ongelmalliseksi, koska molemmat niistä yrittivät generoida koodia samassa kääntäjän vaiheessa. Tämä korjattiin määrittelemällä AspectJ lisäämään koodi jo käännettyihin luokkiin. AspectJ:n kutomisvaiheen siirrosta seurasi kuitenkin, että jotkin Eclipsen konfiguraatiot eivät tunnistaneet Lombokin generoimaa koodia. Itse ohjelman käytössä ja kääntämisessä ei Eclipsellä ollut mitään ongelmia, mutta se näytti virheviestejä kaikissa kohdissa, jotka sisälsivät Lombokin generoimien metodien kutsuja.

Samaan aikaan kun lokitusta toteutettiin, sovellukset siirrettiin Java7-versiosta Java8-versioon. Yhdeksi konfiguraatio-ongelmien syiksi osoittautui käytettyjen kirjastojen versioiden väliset konfliktit. Konfliktien takia tässä vaiheessa lokitusta itse kehitys pysäytettiin hetkeksi, kunnes Javan version 8 siirtyminen oli toteutettu. Tämän jälkeen kirjastojen versioiden korjaaminen oli helpompaa eikä myöskään tehty turhaa työtä, joka olisi jouduttu toistamaan.

Yleisesti AspectJ:n käyttö lokitukseen osoittautui hyvin nopeaksi ja isoimmaksi kompastuskiveksi osoittautui konfiguraation hajautuneisuus ja huono dokumentaatio. Tämä vaikutti kuitenkin yleiseltä ongelmalta Java-sovelluksissa.

Myöhemmin, kun toteutusta korjailtiin ja laajennettiin, sen kompakti rakenne osoittautui hyödylliseksi ja hyvin pienillä muutoksilla pystyttiin vaikuttamaan kattavasti ja yhdenmukaisesti lokitukseen koko sovelluksessa.

6.3 Dokumentaatio ja ohjeistus

Toteutuksen jälkeen tuotettiin dokumentaatio, joka kuvasi lokituksen toiminnan ja ohjauksen mahdollisilla annettavilla parametreilla. Lisäksi kuvattiin lokituksen oletustasot ja lokitiedostojen sijainnit. Lokien sisältöä kuvailtiin korkeammalla tasolla keskittyen niiden

tarkoitettuihin sisältöihin. Tässä vaiheessa tuotettiin myös löyhä ohjeistus siitä, minkälaisiin asioihin olisi hyvä lisätä debug- ja trace-tason lokitusta ja hyvistä käytännöistä lokiviestien muodostamisessa.

Kun ohjeistus oli valmistunut, käytiin se läpi kehitystiimin kanssa. Läpikäynnissä haasteeksi osoittautui, etteivät kaikki kehittäjät tehneet testausta. Testaus oli eritelty oman tiiminsä tehtäväksi, ja todennäköisesti tämän takia monet kehittäjät eivät lisänneet muun työnsä ohessa mitään testauksessa hyödyllistä lokitusta. Myös olemassa olevan toteutuksen hyvin vahva Suomen kielen käyttö vaikeutti osaltaan työskentelyä ja lokien käyttöä.

6.4 Jatkokehitys

Työn aikana ja jälkeen tunnistettiin muutama jatkokehityskohde. Kohteista merkittävin oli lokituksen integroiminen johonkin lokienkäsittelyohjelmaan kuten Apache Chainsaw. Tämä auttaisi lokien käsittelyä mahdollistaen lokitapahtumien näyttämisen säännöllisemmässä muodossa ja lokitiedostojen yhdistämisen. Jos jonkinlainen lokienkäsittelyohjelma otetaan käyttöön, tämä mahdollistaisi myös lokitettujen tietojen määrän kasvattamista ilman lokitapahtumien muuttumista vaikeasti luettaviksi. Koska lokitus pohjautui rivikohtaisiin tapahtumiin ja jos yksittäisellä rivillä olisi liikaa tavaraa, loki muuttuisi hyvin vaikeasti luettavaksi.

Mietinnän alla oli myös prosessikohtaisen lokituksen systemaattinen lisääminen liiketoiminnallisesti kriittisiin operaatioihin, mutta tämä vaatisi paljon enemmän analyysia sopivien kohtien tunnistamiseksi.

7 Yhteenveto

Insinöörityössä kehitettiin asiakkaan sovelluksiin lokitusjärjestelmä kehitys- ja virheenjäljitystyön avuksi. Tämä tehtiin lisäämällä Java-sovellukseen AspectJ-kehystä käyttäen neuvoja, jotka lisäsivät kaikkien lokitapahtumien käyttöön tunnistetietoja sekä lokittivat prosessien käynnistys- ja paluutiedot.

Työ alkuvaiheessa analysoitiin olemassa oleva lokitus, jonka jälkeen tuotettiin lista halutuista lokitustuotteista. Tämän vaiheen jälkeen mietittiin mahdollisia ratkaisuja ja valittiin näistä yksi toteutettavaksi. Kun ratkaisuvalinta oli tehty, siirryttiin suoraan toteutusvaiheeseen.

Työn suurimmiksi haasteiksi osoittautui sovellusten sekava rakenne varsinkin konfiguraatioiden osalta, sekä joidenkin käytettyjen ohjelmointikehysten väliset konfliktit. Nämä onnistuttiin kuitenkin ratkaisemaan oman työn pohjalta.

Työn tavoitteisiin päästiin, mutta jatkokehitettävää jäi vielä varsinkin lokien käsittelyn helpottamisen osalta sekä käytön aikana ilmenneiden puutteiden toteuttamisessa.

Lähteet

1 Eberhardt, Colin. 2014. The Art of Logging. Verkkoaineisto. <https://www.codeproject.com/Articles/42354/The-Art-of-Logging> Luettu 8.4.2020.

2 Nichols, Megan. 2019. Verkkoaineisto. <https://readwrite.com/2019/01/11/the-business-benefits-of-visualizing-your-data/> Luettu 23.4.2020.

3 Elastic NV. Verkkoaineisto <https://www.elastic.co/kibana> Luettu 23.4.2020.

4 Connolly, Stephen. 2017. Verkkoaineisto. <https://www.sitepoint.com/which-java-logging-framework-has-the-best-performance/> Luettu 8.4.2020.

5 Lopes, Cristina. 2002. Aspect-Oriented Programming: An Historical Perspective (What's in a Name?).

6 The AspectJ Programming Guide. Verkkoaineisto. <https://www.eclipse.org/aspectj/doc/next/progguide/semantics-joinPoints.html> Luettu 8.4.2020.

7 Espen. 2010. AspectJ Cheat Sheet. Verkkoaineisto. <https://blog.espenberntsen.net/2010/03/20/aspectj-cheat-sheet/> Luettu 8.4.2020.

8 Design Patterns. Verkkoaineisto. <https://sourcemaking.com/design_patterns/decorator> Luettu 8.4.2020.

9 Comparing Spring AOP and AspectJ. 2020. Verkkoaineisto. <https://www.baeldung.com/spring-aop-vs-aspectj> Luettu 8.4.2020.

10 Spring Framework. Verkkoaineisto. <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-proxying> Luettu 8.4.2020.

11 Design Patterns. Verkkoaineisto. <https://sourcemaking.com/design_patterns/proxy> Luettu 8.4.2020.

12 The AspectJ Development Environment Guide. < https://www.eclipse.org/aspectj/doc/released/devguide/ltw.html> Luettu 8.4.2020.

13 Spring. Verkkoaineisto. <https://spring.io/why-spring> Luettu 8.4.2020.

14 Project Lombok. Verkkoaineisto. <https://projectlombok.org/> Luettu 8.4.2020.

15 Logback project documentation. <http://logback.qos.ch/manual/mdc.html> Luettu 8.4.2020.

16 Paraschiv, Eugen. 2018. SLF4J: 10 Reasons Why You Should Be Using It <https://stackify.com/slf4j-java/> Luettu 8.4.2020.

17 SLF4J. Verkkoaineisto. <http://www.slf4j.org/legacy.html> Luettu 20.4.2020.