Lappeenrannan-Lahden teknillinen yliopisto LUT
LUT School of Engineering Science
Tietotekniikan koulutusohjelma

Kandidaatintyö

Valtteri Kokko

Ohjelmistotuotteen optimointi ja ylläpidettävyys

Työn tarkastaja(t): Jussi Kasurinen

Työn ohjaaja(t): Jussi Kasurinen
Tanja Toroi
Jukka-Pekka Kurki

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT
LUT School of Engineering Science
Tietotekniikan koulutusohjelma

Valtteri Kokko

Ohjelmistotuotteen optimointi ja ylläpidettävyys

Kandidaatintyö
2019

38 sivua, 8 kuvaa, 2 taulukkoa, 1 liite

Työn tarkastajat: Jussi Kasurinen

Hakusanat: kandidaatintyö, ohjelmisto, optimointi, ylläpidettävyys, tekninen velka
Keywords: bachelors' thesis, software, optimization, maintainability, technical dept

Työn tarkoituksena oli selvittää ohjelmistotuotteen business logic sovelluksessa olevia ongelmia. Ongelmat liittyivät huonoon suorituskykyyn järjestelmän käyttäjämäärän lisääntyessä. Työssä vastataan kysymykseen ”Kuinka ohjelmistotuotteen ylläpidettävyyttä voidaan parantaa?”. Ainestoa kerättiin PerfView ja dotTrace monitorointiohjelmilla. Ohjelmistosta löytyi optimoimattomia algoritmeja, suuria luokkia ja rajapintoja, sekä dokumentoinnin puutetta. Myöskään ohjelmiston suorituskyvyn seuraamisen ei käytetty tarpeeksi resursseja. Koodista löytyvät ongelmat ovat lähinnä teknistä velkaa ja ne voidaan ratkaista esimerkiksi SQALE – menetelmällä. Yrityksen tulisi myös lisätä testaukseen uusia tapoja ja työkaluja.

ABSTRACT

Lappeenranta-Lahti University of Technology LUT
LUT School of Engineering Science
Degree Program in Computer Science

Valtteri Kokko

Optimization and maintainability of a software product

Bachelor's Thesis

38 pages, 8 figures, 2 tables, 1 appendices

Examiners: Jussi Kasurinen

Keywords: bachelors' thesis, software, optimization, maintainability, technical dept

The purpose behind this study was to find problems in a software products business logic application. The problems were related to bad performance of the product under bigger user loads. This study answers the question "How to improve the maintainability of a software product?". The material for this work was collected with PerfView and dotTrace monitoring programs. The problems found from the product were unoptimized algorithms, large classes and interfaces and lack of documentation. There was also a lack of resources in monitoring the performance of the product. Nearly all the problems in the code were technical dept and it can be solved E.g. with the SQALE – method. Also, the company should add new practices and tools to testing.

ALKUSANAT

Kirjoitettu Lappeenrannassa loppuvuonna 2018 ja alkuvuodesta 2019. Kiitän Jussi Kasurista, Tanja Toroita ja Jukka-Pekka Kurkea työn ohjaamisesta. Erityiskiitos Tanjalle ja Jukka-Pekalle kärsivällisyydestä työn kanssa.

SISÄLLYS

1 JOHDANTO 3
1.1 Tausta 3
1.2 Tavoitteet ja rajaukset 4
1.3 Työn rakenne 4
2 Kirjallisuuskatsaus 6
2.1 Ylläpidettävyys 6
2.2 Massatestaus 8
2.3 Dokumentointi 8
2.4 Tekninen velka 9
3 Tutkimusmenetelmä ja aineiston keruu 11
3.1 Tutkimusmenetelmä 11
3.2 Massatesti ja kuorman aikaansaaminen 11
3.3 Seurantatyökalut 13
4 Tulokset 15
4.1 Löytyneet ongelmat 15
4.2 Ongelmat aineiston analysoimisessa 17
5 Ylläpidettävyyden parantaminen ongelmakohtien pohjalta 18
5.1 Ohjelmiston tekniset ongelmat 18
5.2 Dokumentointi ja suunnittelu 19
5.3 Uusien testausmenetelmien lisääminen osaksi ohjelmistotuotantoa 20
5.4 Teknisen velan hallitseminen SQALE – metodilla 21
6 Pohdinta ja tulevaisuus 25
7 Yhteenveto 27
8 Lähteet 29
Liitteet

SYMBOLI- JA LYHENNELUETTELO

BL	Business logic
EF	Entity Framework
EU	Euroopan unioni
GDPR	General Data Protection Regulation
HTTP	Hypertext Transfer Protocol
ORM	Object-relational mapper

1 JOHDANTO

Tässä kandidaatin työssä tutkitaan ohjelmiston optimointia ja ylläpidettävyyttä erään yrityksen tuotannossa olevan tuotteen kautta. Työssä käsitellään kyseisestä tuotteesta löytyviä ongelmakohtia, sekä miten niitä voisi tulevaisuudessa korjata ja mahdollisesti välttää, täten parantaen tuotteen yleistä toimivuutta ja ylläpidettävyyttä.

1.1 Tausta

Työn taustalla on yrityksen tarve selvittää ohjelmistotuotteen BL (Business Logic) sovelluksen koko ohjelmistoa hidastavaa toimintaa. Yrityksessä on aikaisemmin tehty testausta, jolla saatiin selville, että erityisesti BL sovellus on se mikä hidastaa järjestelmää eniten.

Tärkein tässä työssä käsiteltävä ongelma on selvittää, miksi BL sovellus ja sitä pyörittävä palvelin hidastuvat merkittävästi, kun sovelluksen samanaikainen käyttäjämäärä kasvaa. On täysin normaalia, että käyttäjämäärän kasvaessa sovelluksen kuorman määrä kasvaa, mutta tässä tapauksessa prosessorikuorma kasvaa paljon odotettua nopeammin ja isommaksi. Kun käyttäjiä on vain yksi ei ole mitään ongelmia, mutta kun käyttäjämäärä lähtee kasvamaan, prosessorikuorma kasvaa, kunnes se on lähes koko ajan 100%, eikä mene enää alaspäin. Prosessorikuorma on 100% jo muutamien kymmenien käyttäjien kohdalla. Tämä toiminta ei ole haluttua ja rajoittaa ohjelmiston käyttäjämäärää merkittävästi.

Kun koodi on sekavaa ja koodissa olevat algoritmit ovat huonosti toteutettuja, on tuotetta vaikea ylläpitää ja tätä myöten ylläpidon kustannukset kasvavat. Tämä tarkoittaa käytännössä sitä, että yksittäisen helpolta näyttävän toimeksiannon tekemisessä saattaakin vierähtää muutama tovi enemmän, kuin mitä alun perin suunniteltiin. Kun toimeksiantoon mennyt aika kasvaa, kasvaa myöskin kulut suoraan sen mukaisesti. Yleensä hyvin toimiva optimoitu koodi on helppoa ylläpitää ja kun koodi on optimoitua, toimii myös ohjelmisto mahdollisimman nopeasti, joka on asiakkaan näkökulmasta erittäin tärkeää.

1.2 Tavoitteet ja rajaukset

Tämän työn tutkimuskysymys on ”Kuinka ohjelmistotuotteen ylläpidettävyyttä voidaan parantaa?”. Kysymykseen vastataan erään yrityksen tuotannossa olevan tuotteen BL sovelluksen ongelmakohtien kautta. Tavoitteena oli löytää kyseisen ohjelmiston prosessorikuormaa aiheuttavia tekijöitä ja miettiä mitä ongelmat oikeasti ovat ja miten niitä voisi korjata sekä mahdollisesti välttää tulevaisuudessa. Voidaan todeta, että lähtöolettamuksena oli: huonosti ylläpidetty koodi aiheuttaa ongelmia ohjelmiston suorituskykyyn.

Lähtökohtana oli aiheuttaa kuormaa BL sovellukselle ja lähteä tarkkailemaan sitä eri työkaluilla keräten statistiikkaa. Sitä mukaa, kun dataa saatiin, voitiin sitä tulkita ja tätä myöten saada selville ongelmakohtia. Koska ohjelmisto, jota tutkitaan, on jo melko vanha ja kooltaan iso, kaikkia ongelmakohtia ei mitä luultavimmin löydetty ja tätä myöten niille ei myöskään voi löytää ratkaisuja. Löytämättömien ongelmien olemassaolo on kuitenkin erittäin todennäköistä, koska ohjelmistokehittäjät ovat tehneet omia ratkaisujaan, mikä tarkoittaa, että koodi ei ole yhdenmukaista ja samantapaisia ominaisuuksia on tehty monella eri tavalla.

Ongelmakohtia etsittiin ja tulkittiin tulkitsemalla erilaisten monitorointiohjelmien tuottamia tuloksia, sekä tarkkailemalla BL sovelluksen koodia käsin. Ohjelmisto on iso ja täten pelkkä koodin lukeminen olisi ollut erittäin työläs tehtävä ongelmakohtien löytämiseen ja täten sitä käytettiin vain erittäin rajatulla alueella koodissa.

Tässä työssä käsitellään vain tutkitun tuotteen BL sovellusta, eikä täten kosketa ohjelmiston muihin osiin kuin vain tarvittaessa. BL sovellus pyörii sille tarkoitetulla erillisellä palvelimella, mutta työssä on käytetty myös paikallisesti ajettavaa versiota kyseisestä sovelluksesta. Työn paino on erityisesti BL sovelluksen koodin parantamisessa. Tämän selvityksen lopputuloksia tullaan hyödyntämään yrityksen ohjelmistokehityksessä.

1.3 Työn rakenne

Johdannon jälkeen, kappaleessa kaksi, käsitellään aiheeseen liittyvää kirjallisuutta ja tutkimuksia. Kolmannessa luvussa selvennetään työn aineiston keräämistä ja neljännessä

luvussa kerrotaan aineistoa analysoimalla löydetyistä ongelmakohdista. Viidennessä luvussa löydetyille ongelmille pyritään löytämään ratkaisumenetelmiä ja selventämään miten ohjelmiston ylläpidettävyyttä voidaan parantaa kyseisten ratkaisujen avulla. Tämän jälkeen käsitellään tätä työtä kokonaisuutena ja miten työtä voisi vielä laajentaa tulevaisuudessa. Viimeinen luku on yhteenveto.

2 KIRJALLISUUSKATSAUS

Tässä kappaleessa käsitellään tämän työn kannalta olennaisia tutkimuksia ohjelmistojen optimointiin ja ylläpitoon liittyen. Kappaleen tarkoituksena on antaa pohjaa työssä käytetyille ratkaisumenetelmille.

2.1 Ylläpidettävyys

Suurin osa yrityksen ajasta ja resursseista, jopa 60 %, menee tuotteen ylläpitämiseen ja tämän takia ohjelmiston ylläpidettävyyden parantaminen on yritykselle erittäin kannattavaa (Perepletchikov 2010, 449). Koska ylläpitämiseen kuluu niin paljon aikaa ja resursseja, helpottaa asiaa huomattavasti, jos muokattava koodi olisi jo valmiiksi sellaista, johon on helppo tehdä muutoksia.

Ylläpidettävyyttä voidaan mitata erilaisilla menetelmillä ja mittareilla, joista osa on standardoitu, kuten ISO/IEC 9126-1:2001 Software engineering - Product quality - Part 1: Quality model. Tämä malli kertoo mitä on vaadittu työmäärä, jolla voidaan muokata ohjelmistotuotetta, sekä neljä mitattavaa luokkaa, joilla voi arvioida ohjelman laatua: analysoitavuus, muokattavuus, vakaus ja testattavuus (Perepletchikov 2010, 451). Edellä mainittu ISO standardi mittaa varsinaisesti ohjelmiston laatua, mutta laatu kertoo myös paljon ylläpidettävyydestä. Standardia on päivitetty alkuperäisen julkaisun jälkeen ja on nykyään ISO/IEC 25010:2011. Ylläpitotehtävien työmäärän laskemiseksi on myöskin kehitetty malleja, kuten Dr. Arvinder Kaurin, Kamaldeep Kaurin ja Dr. Ruchika Malhotran tutkimuksessa, jossa käytettiin erilaisia soft computing - malleja arvioimaan ohjelmiston ylläpidettävyyden työmäärää (2010). Myös olio-ohjelmoinnin suunnittelu mittareita voidaan käyttää laadun mittaamisessa, kuten Victor Basilin & muiden tutkimuksessa todettiin. Basilin tutkimuksessa käytettiin Chidamber & Kemererin kehittämiä olio-ohjelmoinnin suunnittelun mittareita ja hyödynnettiin niitä ohjelmiston virheherkkien luokkien löytämiseen. (Basili et. al. 1996, Chidamber & Kemerer 1994).

Kysymykseen “Miksi ohjelmistotuotteen ylläpitoa kannattaa tehdä?” on vastattu. A. A Takangin ja P. Grubbin Software Maintenance - Concepts and Practice kirjassa listataan muutamia hyviä syitä ylläpidon tekemiseen. Ensimmäinen niistä on, että ohjelmiston pitää

olla aina käytettävissä, sitä pitää korjata ja sen pitää selvitä mahdollisista ohjelmistoon jääneistä virheistä. Jos ohjelmisto kaatuu tai ei muuten vain toimi, voi siitä seurata tappiota yritykselle. Toiseksi ohjelmiston pitää tukea pakollisia päivityksiä, koska maailma muuttuu ja koko ajan tulee uusia määräyksiä valtiolta, EU:lta (Euroopan unioni) tai muulta taholta. Takang ja Grubb mainitsevat vielä, että päivittämällä ohjelmistoa voidaan pitää puolensa kilpailua vastaan. kolmanneksi kerrotaan, että ohjelmistoon pitää pystyä toteuttamaan asiakkailta tulevia pyyntöjä. Käyttäjän ja ohjelmiston tekijän näkemykset eivät aina kohtaa, joten tuotteen oikeilta käyttäjiltä tulee pyyntöjä muuttaa joitain asioita toimivimmiksi tai helpommin käytettäviksi. Neljäntenä syynä ylläpidon tekemiseen mainitaan, että ohjelmistoon pitää pystyä tekemään ylläpitoa myös tulevaisuudessa. (2003) Erittäin hyvä esimerkki uusien määräysten takia tehtävistä muutoksista on EU:n tietosuoja-asetus, eli GDPR (General Data Protection Regulation), joka tuli voimaan kesällä 2018.

Ohjelmiston ylläpidettävyyden parantamista on tutkittu paljon. Esimerkiksi Jeffrey Voas (1998) on tutkinut komponenttipohjaisen järjestelmän ylläpitämistä. Hän esittää, että seuraavia hyviä tapoja kannattaa noudattaa: komponenteista ei saa rakentaa minisysteemeitä, eli komponentit eivät saa olla riippuvaisia toisistaan. Jokainen komponentti tulisi dokumentoida ja pitää yksinkertaisena. Tämä siis tarkoittaa, että komponenttiin ei saa lisätä uusia toimintoja. Viimeisenä kerrotaan kahden sovelluksen jakaman komponentin käsittelystä, eli jos kaksi applikaatiota jakaa saman komponentin, mutta vain toinen sovellus vaatii jaettuun komponenttiin muutoksen, joka rikkoisi taas muut tätä komponenttia käyttävät sovellukset, on parempi tehdä kaksi samantapaista komponenttia, kuin yksi, joka rikkoo toisen sovelluksen. (Voas 1998, 27)

Myös Morshed et. al. kirjoittavat koodin ylläpidettävyydestä liittyen koodin kopioimiseen ja miten sen avulla ylläpidettävyyttä saadaan kasvatettua (2012). Kyseisen työn mukaan tunnistamalla duplikaattikoodia voidaan saada selville erilaisia koodikirjastoehdokkaita, apua ohjelman ymmärtämiseen ja pienentää kooditiedostojen kokoa. Etsimällä duplikaatteja voidaan jopa löytää haitallista koodia vertaamalla omaa koodia jonkin tunnetun haittaohjelman koodiin. Kopioidun koodin löytäminen on myös hyödyllistä mahdollisten tekijänoikeusrikkomusten selvittämisessä. Koodin kopioimisessa on todella paljon huonoja puolia, esim. tilanne, jossa kopioidussa koodissa on virhe, toistuu tämä sama virhe ohjelmistossa muuallakin, kuin vain alkuperäisessä paikassa, tai jos kopioitu

koodi ei sovikaan haluttuun paikkaan, voi se tuottaa uusia virheitä järjestelmään. Myös, jos alun perin käytetty toistuvuus on jo muutenkin huonoa koodia, toistuu se vain useasti ohjelmassa uudestaan. Yksi todella iso ongelma koodin kopioimisessa on sen tuottamat kustannusten nousut ohjelman ylläpidossa, jotka osittain johtuvat lisääntyneistä resurssitarpeista. (Morshed et. al. 2012, 2-3)

2.2 Massatestaus

Tässä työssä tehdään testausta suurella käyttäjämäärällä testattavan ohjelmiston ongelmakohtien löytämiseksi, mikä tarkoittaa todellisuudessa sitä, että generoidaan virtuaalisia käyttäjiä käyttämään järjestelmää. Tällaista testaamista, eli massatestaamista, on kolmea tyyppiä, kuormatestaamista, suorituskykytestaamista ja stressitestaamista. Kuormatestaamisen tarkoituksena on arvioida järjestelmän toimintaa suurella käyttäjämäärällä. Eli etsitään bugeja, joita esiintyy vain suurella käyttäjämäärällä tai muuta yleiseen laatuun suurella käyttäjämäärällä liittyviä ongelmia. Suorituskykytestaamisessa on tarkoituksena arvioida, kuinka hyvin ohjelmisto toimii suurella käyttäjämäärällä, eli mitataan vasteaikoja ja resurssien käyttöä. Stressitestaaminen tarkoittaa, että testattava ohjelmisto laitetaan äärirajoilleen, eli kuormaa on enemmän, kuin mitä ohjelmisto on suunniteltu kestämään. (Jiang & Hassan 2015, 1092 - 1093) Tähän työhön liittyen olennaisimmat menetelmät ovat kuormatestaaminen ja suorituskykytestaaminen, koska halutaan tietää syyt liian suureen prosessorikuormaan.

2.3 Dokumentointi

Ylläpidettävyyttä ajatellen ohjelmiston dokumentaatio on suuressa roolissa. Garuosin ja muiden tutkimuksessa tutkittiin, miten teknistä dokumentaatiota käytetään ja miten hyödylliseksi se koetaan ohjelmistokehityksessä. Asiaa tutkittiin vain yhdessä yrityksessä ja siellä todettiin, että ohjelmistokehittäjät selvittävät tarvittavia tietoja ensin työkavareiltaan, ja vasta sen jälkeen suoraan dokumentaatiosta. Kuitenkin ensimmäisenä dokumentaationa katsotaan lähdekoodia sekä sinne kirjoitettuja kommentteja ja vasta tämän jälkeen katsotaan ylemmän tason dokumentaatiota. Kuitenkin teknistä

dokumentaatiota käytettiin paljon tietolähteenä kehitystyötä tehtäessä. (Garousi et. al. 2013)

2.4 Tekninen velka

Tekninen velka on yksi erittäin tärkeä asia ohjelmiston ylläpitoa ajateltaessa. Tekninen velka tarkoittaa sellaisia päätöksiä koodissa, joissa on valittu nopeasti implementoitava ratkaisu parhaan ratkaisun sijasta. (Technopedia) Kun näitä valintoja alkaa kertymään enemmän, voidaan ohjelmistolla sanoa olevan teknistä velkaa. Velkaa voi olla myös muualla kuin koodissa, esimerkiksi joskus teknistä velkaa voi olla aina arkkitehtuuritasolla asti mutta se on paljon vaikeampaa huomata, kuin yleisempi koodissa oleva velka. Tekninen velka on kaikkea hankaloittava ongelma, koska se vaikuttaa kaikkeen aina ominaisuuksien määrittelystä koodin kirjoittamiseen ja tuotteen toimittamiseen asiakkaalle. Tämän lisäksi teknistä velkaa alkaa yleensä kertyä jo ohjelmistoprojektin alkuvaiheissa. (Avgeriou et. al. 2016, 66, 70)

Avgeriou ja muut esittelevät tavan, jolla teknistä velkaa voidaan hallita ja mihin suuntaan teknisen velan tunnistamiseen käytetyt työkalut ovat menossa. Heidän mukaansa teknisen velan hallinta voidaan jakaa viiteen eri osaan. Ensin on tunnistettava missä on velkaa, ja minkälaista se on. Tämä voidaan tehdä esimerkiksi koodin katselmoinnilla. Toisessa vaiheessa pitää arvottaa löydettyjä velkoja kustannusten ja korjauksesta saatavan hyödyn perusteella. Hyödyt ovat vaikeampia arvioida, mutta kustannuksia voidaan arvioida hyvinkin tarkasti erilaisilla työkaluilla. Kolmannessa vaiheessa arvotetut velat pitää priorisoida. Ensimmäisenä tulisi korjata kohdat, joista saadaan isoin arvo tai ovat muuten tärkeitä korjattavia. Kun tämä on saatu tehtyä, voidaan siirtyä kohtaan neljä, joka on itse ns. ”velan takaisinmaksu”, eli refaktoroidaan huono koodi. Viides kohta on jäljelle jääneiden velkojen seuraaminen, koska niiden arvo ei välttämättä pysy ajan kanssa samana. Tämä on tärkeää, koska jotkin näistä voivat olla jossain vaiheessa erittäin vaikeita hallita tai korjata. (Avgeriou et. al. 2016, 68 - 69)

Avgerioun ja muiden mukaan teknisen velan työkalut tulevat kehittymään pelkän lähdekoodin katsomisen sijasta myös sen ulkopuolelle. Tämä tarkoittaisi sitä, että työkalut voisivat tarkastella esimerkiksi ohjelmiston arkkitehtuuria ja etsiä siltä tasolta teknistä

velkaa. Työkalut voisivat vaivattomasti kertoa, missä komponentissa on korjattavaa ja mitä arkkitehtuurissa tulisi muokata sekä arvottaa ja priorisoida nämä kohdat valmiiksi. Toinen kehityskohta tulee heidän mukaansa olemaan koodirepositorioissa, josta tullaan etsimään refaktoroinnin tarpeessa olevaa arkkitehtuuria ja koodia. Löydetty tekninen velka tullaan automaattisesti dokumentoimaan velan takaisin maksua varten. (Avgeriou et. al. 2016, 69)

3 TUTKIMUSMENETELMÄ JA AINEISTON KERUU

Aikaisemmin tämän työn kohteena olevassa yrityksessä huomattiin, että tuotteen suorituskyky ei ole ihan sitä, mitä toivottaisiin ja tästä heräsi halu lähteä tutkimaan, mistä tämä johtuu. Alun perin luultiin, että käyttöliittymä on hidas, mutta tarkemman tarkkailun myötä paljastui, että oikea hidastelun lähde on tuotteen BL sovellus. Palvelimen prosessorilla oli vaikeuksia isompien kuormien kanssa. Prosessorikuorma menee todella nopeasti 100 %:iin, kun sovellukselle laitetaan enemmän kutsuja. Yritys luonnollisesti haluaa tietää, miksi suorittimelle tulee liiallista kuormaa.

3.1 Tutkimusmenetelmä

Tutkimusmenetelmänä tässä työssä on käytetty empirical case-study – menetelmää, mikä tarkoittaa sitä, että tämän työn aiheita on tutkittu vain yhden yrityksen erään tuotteen ympäristössä. Tuotteessa esiintyy ongelma, jota on käytetty lähtökohtana tälle tutkimukselle. Työssä vastataan kysymykseen ”Kuinka ohjelmistotuotteen ylläpidettävyyttä voidaan parantaa?” yrityksen tuotteen ongelmakohtien kautta. Kaikki työssä käsiteltävä aineisto ongelmakohtiin liittyen on yrityksen tuotteesta kerättyä. Ongelmakohtiin etsitään mahdollisia ratkaisuja, sekä yrityksen sisältä, että katselmoimalla aiheeseen liittyviä tutkimuksia ja kirjallisuutta.

Empirical case study on tutkimusmenetelmä, jossa käsitellään jotain ilmiötä sen oikeassa ympäristössä varsinkin, kun raja ilmiön ja ympäristön välillä ei ole suoraan nähtävissä. Kuten Yin kirjassaan “Case study research Design and methods” (2009) mainitsee tämän tutkimusmenetelmän olevan hyvä, jos epäillään ympäristön olevan oleellinen osa ilmiötä. (Yin, 2009, 13) Ympäristönä tässä työssä on tarkemmin tuotteen Bl sovellus, jonka sisäisten toteutuksien epäillään olevan iso syy sen hidastumiseen. Tämän takia tutkimusmenetelmäksi on valittu empirical case-study.

3.2 Massatesti ja kuorman aikaansaaminen

Työn tekeminen aloitettiin aiheuttamalla kuormaa BL sovellukselle. Kuorma saatiin aikaan käyttämällä yrityksessä käytettävän Microsoft Visual Studion massatestaustyökalua. Tällä

työkalulla voidaan simuloida oikeita käyttäjiä tallentamalla ensin haluttavat Hypertext Transfer Protocol (HTTP) kutsut yksittäiseen testiin ja sen jälkeen liittämällä tämä testi massatestiin. Kutsujen tallentaminen tapahtuu asettamalla nettiselaimeen kuuntelija, joka nappaa kaikki selaimen lähettämät kutsut. Eli käytetään ohjelmaa, kun sitä on tarkoitettu käytettävän ja otetaan tietystä tapahtumasarjasta ylös tapahtuvat kutsut. Tämän jälkeen näitä kutsuja voidaan kutsua tietystä testistä aina samalla tavalla. Kun testi on todettu toimivaksi, voidaan se liittää massatestiin. Massatestillä saadaan aikaan tarvittavaa kuormaa suorittamalla haluttuja testejä samanaikaisesti. Yhdessä massatestissä voi olla useita testejä, sen ajallinen pituus ja haluttu käyttäjämäärä voidaan määritellä halutunlaisiksi. Massatestin luonnetta voi myös muuttaa. Se voi olla joko inkrementaalinen tai bulkkitesti. Inkrementaalisessa testissä määritellään alkuperäinen käyttäjämäärä, sekä maksimikäyttäjämäärä ja käyttäjämäärän lisäys tietyin väliajoin. Eli määritellään, miten paljon käyttäjiä lisätään tietyin aikavälein, kunnes päästään haluttuun maksimiin. Bulkkitestissä massatestille annetaan vain käyttäjämäärä ja aika. Tällöin testi laittaa suoraan maksimikäyttäjämäärän ja suorittaa testiä halutun ajan tällä käyttäjämäärällä. Näissä molemmissa testejä ajetaan kuitenkin samalla tavalla. Testejä laitetaan käyntiin aina käyttäjämäärän verran ja kun yksi testi saadaan valmiiksi, aloitetaan heti uusi testi. Esim. jos tehdään bulkkityyppinen testi, jossa alkuperäinen käyttäjämäärä on 20, aloittaa massatesti välittömästi 20 testiä. Kun massatestin yksittäinen testi tulee päätökseen, aloitetaan uusi samanlainen testi, eli testausohjelma pyörittää koko ajan 20:ta testiä, kunnes aikaraja tulee vastaan.

Tässä työssä massatestiä varten on yksi testitapaus, joka on melko kattava tapaus siitä, miten oikea käyttäjä ohjelmaa käyttäisi. Massatesti on tehty aloittamaan 20 käyttäjällä ja lisäämällä aina 20 sekunnin välein 10 käyttäjää, kunnes käyttäjämäärä on tavoittanut 250 käyttäjää. Testi kestää kokonaisuudessaan 10 minuuttia. Tällä saadaan kattava kuva siitä, miten ohjelmisto käyttäytyy erilaisilla käyttäjämäärillä ja miten hyvin sovellus kestää kasvavaa kuormaa. Näillä asetuksilla testiä keretään vielä ajaa muutama minuutti tuolla täydellä 250 käyttäjän kuormalla.

3.3 Seurantatyökalut

Kun järjestelmälle on aiheutettu kuormaa, pitää pystyä erittelemään, mitä ohjelmistossa tällöin tapahtuu. Tämä tarkoittaa sitä, että on kerättävä dataa ohjelman käyttäytymisestä jollain tarkkailuohjelmalla. Tässä työssä on käytetty lähinnä kahta tähän tarkoitukseen olevaa ohjelmaa: Microsoftin avoimen lähdekoodin PerfView-ohjelmaa, sekä Jet Brainsin dotTrace- ohjelmaa.

PerfView on ilmainen suorituskyvyn analysointityökalu. Se on kehitetty tunnistamaan lähinnä prosessori- ja muistiongelmia. PerfView on Windowsille tarkoitettu työkalu, mutta siinä on myös osittainen tuki tutkia Linux- järjestelmiä. Työkalu soveltuu parhaiten .NET:in ajonaikaisten ongelmien etsimiseen, eli lähes täydellinen tämän työn puitteissa. Kun Perfviewiä ajetaan, tallentaa se kaikki tapahtumat, mitä tietokone ajon aikana käsittelee. Eli PerfView ei kerää vain halutun ohjelman tapahtumia, vaan kaikkien käyttöjärjestelmän prosessien tapahtumat. Tämä antaa meille erittäin kattavan, ellei täydellisen kuvan siitä, mitä oikeasti tapahtuu. (PerfView Github)

DotTrace on JetBrains s.r.o:n kehittämä työkalu erilaisten .NET sovellusten analysoimiseen. Se pystyy kertomaan tarkasti, mitkä ohjelmiston osat ovat hitaita tai muuten ongelmallisia. DotTracessa on tätä työtä varten yksi erittäin hyvä tapa kerätä dataa, ja dotTracessa sitä kutsutaan timeline-profiloinniksi. Se sallii metodien sijoittamisen aikajanalle, samaan tapaan kuin PerfViewissäkin. DotTracen antaa dataa, joka kertoo suoraan, ohjelmiston hitaat metodit sekä tietokantakutsut ja näyttää samalla, mistä ne oikeasti koodin puolella löytyvät. DotTrace kertoo myös kaikki polut, jota kautta metodeja on kutsuttu. (dotTrace)

Tässä työssä dotTracen profilointi kiinnitettiin seuraamaan BL prosessia. Itse profilointi ei kuitenkaan ala vielä kiinnityksessä, vaan profilointi pitää aloittaa itse. Profilointi aloitettiin manuaalisesti, koska timeline-profiloinnin tulokset vaativat todella paljon levytilaa profilointikoneelta ja halutiin tietää, mitä tapahtuu isommilla käyttäjämäärillä. Jos profilointi olisi aloitettu heti prosessin lähdettyä käyntiin, olisi saatu jonkin verran lähes tyhjää dataa, kun järjestelmässä on vain se alkuperäinen 20 käyttäjää. Profilointia tehtiin 30 – 60 sekuntia kerrallaan ja se aloitettiin jossain kohtaa keskellä massatestiä. Aloituskohdalla ei ollut niin paljoa väliä, koska haluttiin vain saada tietoon, mitä

ohjelmistossa tapahtuu, kun siellä on enemmän käyttäjiä. PerfViewiä ajettiin lähes samalla tavalla, tosin sitä ei erikseen tarvinnut kiinnittää mihinkään prosessiin. Kuvia ohjelmista ja kerätystä aineistosta löytyy liitteestä 1.

4 TULOKSET

Tässä kappaleessa esitellään aineiston perusteella ilmi tulleet ongelmat. Kappaleen lopussa käydään vielä nopeasti läpi ongelmia, joita kohdattiin aineistoa analysoitaessa.

4.1 Löytyneet ongelmat

Kerätyn datan tulkitsemiseen on käytetty samoja ohjelmia, millä dataa kerättiinkin, eli dotTracea, sekä PerfViewiä. Näissä on erittäin hyvät tulkitsemista auttavat työkalut. Molemmat näistä listaavat ohjelmiston tapahtumia erittäin yksityiskohtaisesti ja näitä tapahtumia voi ryhmitellä parhaaksi katsomallaan tavalla. Kummastakin saa periaatteessa samat asiat selvitettyä, mutta PerfViewistä saa vielä muut samaan aikaan tapahtuneet Windowsin tapahtumat kaikkien tuotteen tapahtumien päälle. Muihin tapahtumiin ei tässä työssä kuitenkaan kiinnitetä huomiota, koska ne eivät ole työn kannalta olennaisia.

Käytetyillä tavoilla löytyi ohjelmistosta melko isojakin ongelmia. Näistä ehkä tärkeimpänä prosessorikuormaa ajatellen oli järjestelmän tuottamat poikkeukset koodissa. Molemmista monitorointiohjelmista huomattiin, että järjestelmässä tapahtuu yllättävän paljon tapahtumia poikkeustenkäsittelijässä. Poikkeustenkäsittelijä aktivoituu, kun ohjelmistossa tapahtuu jokin määritelty poikkeus. Tätä ei pitäisi tapahtua oikeastaan ollenkaan normaalissa suorituksessa, koska virheiden käsittely on erittäin työläs operaatio. Vielä työläämmän siitä saa, jos virhe on odotettu tapaus, eli koodissa Try-Except lohkon Except osion sisään tehdään logiikkaa. Noin 30 sekunnin monitoroinnin aikana järjestelmä saattoi napata jopa 5000 poikkeuksenkäsittelyä, mikä on erittäin paljon verrattuna siihen, että niitä ei oikeastaan pitäisi olla ollenkaan. Tämän todettiin olevan erittäin korkealla prioriteetilla korjattavien asioiden listalla ja sen takia tähän pureuduttiin vielä vähän tarkemmin.

DotTracella tästä saatiin vielä selvitettyä, että yksittäisen testitapauksen aikana, vain yhdellä käyttäjällä, tapahtui 34 poikkeusta. Nämä poikkeukset ovat siitä mielenkiintoisia, että ajettaessa ohjelmistoa paikallisessa testiympäristössä debug-tilassa, ei Visual Studio osannut napata niitä. DotTracella saatiin kyllä selville, missä nämä poikkeukset tapahtuivat koodin puolella, mutta ongelmaksi tässä tuli selvittää, miksi työkalun osoittamasta paikasta tuli poikkeus. Koska tätä poikkeusta ei saatu kiinni omasta koodista, todettiin, että

poikkeuksen on tultava jonkin käytettävän kirjaston koodista. Lopulta selvisi, että poikkeus syntyi Entity Frameworkin (EF) koodissa. Entity Framework on object-relational mapper (ORM), joka on kehitetty Microsoftin .NET teknologialle. EF:n tehtävä on yhdistää olio pohjainen sovellus ja relaatiotietokanta käyttämällä vahvasti tyypitettyjä .NET objekteja, jotka ovat osa sovellusta. (Microsoft) Ohjelmiston poikkeukset johtuivat EF:n käyttämän objektin ja tietokannasta saadun datan eroavaisuuksista. Käytännössä tämä tarkoitti sitä, että tietokannasta ei saatu dataa kaikille objektin attribuuteille ja EF heitti virheen kaikista kohdista, joihin ei saatu dataa.

Poikkeusten lisäksi tuotteesta löytyi joukko muita ongelmia. Näitä ovat isot luokat ja rajapinnat, optimoimattomat algoritmit, EF:llä tehdyt raskaat kantakyselyt, kunnollisen dokumentoinnin puuttuminen, sekä suorituskykytestauksen puuttuminen.

Isoiksi paisuneita luokkia ja rajapintoja ohjelmasta löytyi melko paljon. Näitä ei suoraan pystytty tunnistamaan monitorointiohjelmilla, mutta niitä tuli vastaan, kun jotain monotoriohjelman osoittamaa ongelmapaikkaa lähti tutkimaan tarkemmin. Tällaiset luokat ja rajapinnat ovat ylläpidettävyyden kannalta erittäin ongelmallisia. Näissä voi olla suuren kokonsa takia metodeja, jotka eivät liity luokkaan, kuollutta koodia tai muita huomaamattomia ongelmia. Tällaisten luokkien ja rajapintojen uudelleenkäyttö voi olla erittäin työlästä, koska sen initialisoiminen koodissa voi tarvita turhaa työtä ja dataa. Jos luokka tai rajapinta tekee jotain mitä sen ei välttämättä tarvitse tehdä, pitää sitä muokattaessa varmistua, että muutos toimii kaikkialla, missä sitä saatetaan käyttää. Esimerkiksi jos ohjelmiston jossain osiossa huomataan bugi, joka korjataan, voi tämä muutos aiheuttaa toisen bugin jossain muussa osassa ohjelmistoa.

Tuotteesta löytyi optimoimattomia tai muuten raskaita algoritmeja. Nämä voivat olla yksi suuri osasyy prosessorikuorman kasvuun ja ovat yksi kohta, johon pitää puuttua ohjelmiston ylläpidettävyyden parantamiseksi. Huonosta toteutuksena esimerkkinä voidaan pitää tietokantakutsujen suorittamista silmukoiden sisällä. Eli metodit, jotka tekevät kutsuja parametrilistan läpi iteroivassa for-silmukassa. Tämä ei yleisesti ottaen ole hyvä tapa suorittaa useita tietokantalausekkeita. Tätä vielä entisestään hidastaa EF:in käyttö. Koska kysely joudutaan muodostamaan ja suorittamaan jokaisella for – silmukan kierroksella uudestaan EF:in toimesta, kasvaa resurssien käyttö huomattavasti. Tämän

lisäksi ohjelmistossa oli huonosti toteutettuja algoritmeja, joita on vaikea ymmärtää ja ylläpitää. Vaikean ymmärrettävyyden syynä oli harhaanjohtavasti nimetyt metodit, monimutkaiset ja epäselvät algoritmit sekä virheellinen tai arvoton kommentointi.

Yksi suurimmista ongelmista ylläpidettävyyden kannalta, mitä huomattiin, oli puutteet tuotteen dokumentaatiossa. Tällä tarkoitetaan dokumentaation puuttumista koodista, sekä joko vaikeasti löydettävää tai puuttuvaa dokumentaatiota suurimmasta osasta ohjelmiston ominaisuuksia. Tämä hankaloittaa ohjelmiston ylläpidettävyyttä ja voi tuottaa uusia ongelmia vanhoja korjatessa. Yrityksen laajentumisen kannalta tämä myös hankaloittaa asioita, koska esimerkiksi uusien työntekijöiden kouluttamiseen menee ylimääräistä aikaa ja tätä kautta rahaa.

Nykyisessä kehitysmallissa ei ollut systemaattista testausta tai muuta suoraa tapaa seurata ohjelmiston suorituskykyä. Tämä käytännössä tarkoittaa sitä, että ohjelmistoa kehitettäessä ei seurattu valmistuneiden ominaisuuksien suorituskykyä systemaattisesti. Tällainen johtaa lopulta nykytilanteeseen, jossa ohjelmisto hidastuu tuntemattomista syistä ja kun lopulta saadaan selville, miksi ohjelmisto hidastuu, todetaan, että ongelmien korjaaminen on erittäin kallis operaatio.

4.2 Ongelmat aineiston analysoimisessa

Kerätyn datan tulkitseminen ei aluksi ollut helppoa, koska käytettiin vain PerfView ohjelmaa. Tässä ongelmaksi tuli, että kyseinen ohjelma on 32 – bittinen ja sillä monitoiroitiin palvelimella 64 – bittistä prosessia. Tämän seurauksena ohjelma näytti vain jonkin verran hyödyllistä dataa. Tämän datan syvempi tutkiminen oli erittäin hankalaa, koska osa call stackeista olivat menneet rikki jossain datan keräämisen ja datan luettavaan muotoon muuttamisen välissä. Rikkinäiset stackit näkyivät ohjelmassa ”BROKEN”-merkillä, eikä näistä saanut mitään selvitettyä. Se mitä PerfViewillä kuitenkin saatiin selvitettyä auttoi näyttämään suuntaa, mitä pitäisi jollain toisella ohjelmalla tutkia tarkemmin. Tämä toinen ohjelma oli DotTrace, jolla päästiin tutkimaan tarkemmin PerfViewin osoittamia ongelmia.

5 YLLÄPIDETTÄVYYDEN PARANTAMINEN ONGELMAKOHTIEN POHJALTA

Tässä kappaleessa käsitellään edellisessä kappaleessa esille tuotuihin ongelmiin mahdollisia tai jo kokeiltuja ratkaisuja. Esitellyillä ratkaisuilla pyritään parantamaan BL sovelluksen ylläpidettävyyttä, sekä mahdollisesti suorituskykyä.

5.1 Ohjelmiston tekniset ongelmat

Järjestelmässä tapahtuville EF tuottamille poikkeuksille on muutama korjausvaihtoehto. Voidaan joko muuttaa luokkaa, johon EF laittaa kyselyn tulokset tai käyttää EF:n omaa data annotaatiota, joka voi korjata ongelman. Jos data annotaatiota ei haluta käyttää, on käytetty luokka muutettava sellaiseen muotoon, että se ei sisällä turhia attribuutteja. Tämä tarkoittaa sitä, että jokaiselle kyselylle pitäisi tehdä oma luokka. Koska tässä työssä ei testattu koko järjestelmää on vaikea arvioida, miten paljon tällaisia luokkia pitäisi korjata tai tehdä. Nykyisellä rakenteella, jos käyttöliittymäpuoleen ei haluta koskea ollenkaan, vaatisi tämä sitä, että kyselystä saadut tiedot tulisi sijoittaa vielä uuteen, BL sovellukselta eteenpäin annettavaan luokkaan.

Jos taas ei haluta tehdä jokaiselle kyselylle omaa luokkaa, voidaan käyttää Entity Frameworkin omaa NotMapped data annotaatiota. Jos tämän laittaa luokan määrittelyssä jollekin attribuutille, tarkoittaa se, että EF ei yritä etsiä kyseiselle attribuutille arvoa tietokantakyselyn vastauksesta. Tämän käyttäminen tarkoittasi, että luokassa voi olla kaikki halutut attribuutit, eivätkä ne aiheuttaisi ylimääräistä päänvaivaa. Käsiteltävän ohjelmiston nykyisessä tilassa tämän käyttäminen vaatii kuitenkin jonkin verran töitä, koska tätä kokeiltaessa huomattiin, että se ei nykyisellään toimi. Toimimattomuus johtuu siitä, että NotMapped annotaatio on siirretty eräässä .NET:in päivityksessä EF:n puolelle. Aikaisemmin kyseinen ominaisuus oli .NET:in omissa data annotaatiossa. Tästä johtuen tuotteessa käytetään sekä vanhaa .NET:in kirjastoa, että EF:n uutta kirjastoa, mikä johtaa siihen, että kyseistä ominaisuutta käytettäessä, ei kääntäjä osaa päättää kumpaa versiota tästä käytetään. Annotaatiota ei voida käyttää edes määrittämällä, mistä kirjastosta sitä käytetään, koska kirjastot ovat tismalleen saman nimisiä. Ratkaisuna tässä on poistaa

vanha kirjasto ja korjata kaikki sen aiheuttamat uudet poikkeukset, mutta tätä ei ole vielä keretty toteuttamaan.

Järjestelmässä on useita kohtia, joissa tehdään tietokantakutsuja silmukan sisältä ja kaikki tällaiset kohdat, joita tässä työssä käsiteltiin, oli jo valmiiksi merkattuna, kuten tulosten käsittelyssä aiemmin mainittiin. Tähän ratkaisuna on yksinkertaisesti vain etsiä kaikki tällaiset kohdat ja refaktoroida ne. Koska suurin osa tällaisista rakenteista on jo merkitty koodiin, jäljelle jää realistisesti vain niiden refaktoirointi. Useimmiten näissä rakenteissa silmukan jokaisella kierroksella muodostetaan uusi kysely parametreineen. Tämä pitää saada muotoon, jossa parametrit pidetään muistin varassa, kunnes kaikki tarvittava on tiedossa ja tämän jälkeen suorittaa mahdollisuuksien mukaan vain yksi kysely, jossa saadaan kaikki tarvittava data. Muille huonosti toteutetuille algoritmeille ei ole suoraan yhtä helppoa ja yleispätevää ratkaisua. Tätä vielä hankaloittaa, että kaikkia huonoja tai muuten hitaita algoritmeja ja metodeja ei ole merkitty tai löydetty. Tähän on kuitenkin olemassa ratkaisuja, joita käsitellään kohdassa 5.4.

Ylläpidettävyyden kannalta isot ja monimutkaiset luokat ovat tuotteessa erittäin hankalia, kuten aikaisemmin jo mainittiin. Näiden korjaamiseksi on oikeastaan vain yksi vaihtoehto. Isoja luokkia ja rajapintoja on pilkottava pienemmiksi kokonaisuuksiksi. Nykyisistä luokista on lähdettävä tunnistamaan pienempiä kokonaisuuksia ja tätä kautta ryhmitellä omiksi luokikseen, joita on huomattavasti helpompi käyttää ja ylläpitää. Rajapintoja saa pienennettyä sillä, että niiden ei tarvitse kuvata koko luokkaa. Tämä tarkoittaa sitä, että yhdellä luokalla voi olla monta rajapintaa, joita se toteuttaa.

5.2 Dokumentointi ja suunnittelu

Dokumentointia voisi lähteä toteuttamaan kahdella eri tavalla: lisäämällä kommentteja koodiin, jos koodi on vaikeasti ymmärrettävää ja tekemällä ylemmän tason dokumentaation siitä, miten ohjelmiston ominaisuuksien pitäisi toimia.

Koodin kommentointi voidaan lähteä tekemään joko refaktoroimalla koodi sellaiseksi, että se ei tarvitse kommentteja tai kirjoittamalla kaikkien sitä tarvitsevien metodien päälle, miksi kyseinen metodi on olemassa. On sanomattakin selvää, että koodin refaktorointi on

työläämpi vaihtoehto näistä, mutta se on pitkällä tähtäimellä ylläpidettävyyden kannalta erittäin kannattavaa. Tämä auttaa selventämään metodin olemassa olon syytä kaikille projektin parissa toimiville henkilöille. Koska tämä on erittäin työläs operaatio, olisi hyvä ottaa yleiseksi käytännöksi kirjoittaa vähänkään epäselvemmän metodin päälle tai mahdollisesti jopa sisälle kommentteja koodin toiminnasta. Tämä toimenpide nopeuttaa koodin ymmärtämistä, mikä tarkoittaa sitä, että se on helpompi muuttaa ymmärrettävämmäksi koodiksi, kun se lopulta refaktoroidaan. Kuten aiemmin mainittiin, on tämä yksi tärkeimmistä dokumentaatiosta ohjelmistokehittäjille (Garousi et. al. 2013).

Koska tässä työssä käsitelty tuote on paikoitellen erittäin ominaisuusrikas, voi välillä tulla vastaan tilanne, jossa bugia korjattaessa ei ole tietoa, miten ominaisuuden pitäisi oikeasti toimia. Erityisesti näihin tilanteisiin olisi erittäin hyvä olla dokumentti, johon on selvennetty ainakin jollain tasolla, miten kyseisten ominaisuuksien teknisesti tulisi toimia. Tällä dokumentilla voidaan paremmin varmistamaan bugien oikeanlainen korjaaminen, ilman uusien bugien tuottamista.

Dokumentoinnista on hyötyä uusia ominaisuuksia tehtäessä. Jos uusien ominaisuuksien toiminta suunnitellaan ja dokumentoidaan, on ne huomattavasti helpompi toteuttaa. Tämä periaatteessa tarkoittaa sitä, että kaikki uudet ominaisuudet tulisi suunnitella, ennen kuin niitä voidaan lähteä toteuttamaan. Kun ominaisuudet ovat hyvin suunniteltuja, on ne erittäin helppoa tämän jälkeen toteuttaa ja tulevaisuudessa ylläpitää. Käytännössä tämä voisi tarkoittaa esimerkiksi, että ohjelmistokehittäjä kirjoittaa ensin dokumentin työstämästään ominaisuudesta ja sen toimintaperiaatteesta, jonka jälkeen ominaisuus vasta toteutettaisiin.

5.3 Uusien testausmenetelmien lisääminen osaksi ohjelmistotuotantoa

Tällä hetkellä ohjelmistokehitysprosessissa ei ole käytössä kunnollista massatestausta ja ohjelmiston suorituskyvyn testaamiseen ei käytetä tarpeeksi resursseja. Nämä molemmat ovat melko helppoja toteutettavia esimerkiksi tässä työssä käytetyillä työkaluilla.

Microsoftin Visual Studiossa on hyvät työkalut massatestien toteuttamiseen. Testejä on helppo tehdä, niitä on helppo hallita ja ne saa automatisoitua suoraan Visual Studiosta.

Monitorointiin Jet Brainsin dotTrace on monipuolinen ja helppo työkalu. Ohjelmistokehittäjän testatessa omaa koodiaan, voi tällä työkalulla helposti seurata oman koodinsa tehokkuutta ja huomata optimointia tarvitsevat metodit. Monitorointi ohjelmalla saa myös selville ongelmia, joita ei perinteisillä menetelmillä huomata ollenkaan, kuten esimerkiksi järjestelmästä löytyneet poikkeukset.

Nykyiseen testausprosessiin tulisi implementoida tarkemmat ja säännöllisemmät koodikatselmukset. Katselmuksien lisääminen helpottaa virheiden löytämistä ajoissa, sekä on helppo implementoida osaksi nykyistä ohjelmistokehitysprosessia (Zhu 2016, 2). Käytännössä tämän voidaan implementoida kierrättämällä kaikki valmiit uudet ominaisuudet tai bugikorjaukset vähintään yhden saman projektin parissa työskentelevän kautta. Eli kaikki koodi menisi vähintään kaksien silmäparien läpi, ennen kuin koodi menee tuotantoon. Tällä tavalla voidaan saada karsittua jopa 20 – 90% uusista bugeista tai huonosta koodista (Zhu 2016, 2). Katselmoinneilla voidaan suoraan vähentää ylläpidon kustannuksia tulevaisuudessa ja niiden lisäämisen sivuvaikutuksena useammalla ohjelmistokehittäjällä olisi parempi ymmärrys ohjelmiston toiminnasta.

5.4 Teknisen velan hallitseminen SQALE – metodilla

Jos teknisen velan hallitsemiseksi ei tehdä mitään, voi se lopulta johtaa isoihinkin ongelmiin ja muutoksiin tuotteessa. Uudet velat vain kasautuvat vanhan päälle ja lopulta uusien ominaisuuksien toteuttaminen tai jopa asiakkaiden kokemukset voivat kärsiä ja tästä lopulta seuraa kustannuksien iso kasvu. Teknisen velan hallitsemiseen ei tällä hetkellä ole mitään standardoitua tapaa tai määritelmää (Ilkiewicz et. al. 2012, 44). Kuitenkin eräs kehitetty ja jo melko laajalti käytetty metodi on avoimen lähdekoodin Software quality assessment based on life-cycle expectations (SQALE).

SQALE on kehitetty yleispäteväksi ja kestäväksi ratkaisuksi arviomaan lähdekoodin laatua, koska nykyiset standardit ISO 9126 ja ISO/IEC 15939 eivät tähän kykene. SQALE ei ole riippuvainen mistään ohjelmasta tai koodikielestä, joten se on erittäin monikäyttöinen. SQALE pystyy: määrittelemään selkeästi, mitkä asiat luovat teknistä velkaa; arvioimaan teknisen velan määrää; analysoimaan velkaa teknisestä

liiketoiminnallisesta näkökulmasta sekä tarjoamaan erilaisia priorisointistrategioita optimaalisen suunnitelman velan takaisinmaksuun. (SQALE)

SQALE mallissa on neljä tärkeää konseptia: Laatumalli, Analyysimalli, Indeksit ja Indikaattorit. Näistä ensimmäisenä lähdetään käsittelemään Laatumallia. Sen tarkoituksena on määrittää vaatimukset koodille, mikä tapahtuu kehittämällä ohjelmistolle ei – toiminnallisia vaatimuksia. Tämä tarkoittaa sitä, että yrityksen tai projektitiimin on itse määriteltävä, millaista on hyvä koodi. Tämä on tehtävä, koska tällä määritetään, millainen koodi lasketaan tekniseksi velaksi. (Letouzey 2012, 32) Laatumalli on sopimus projektin parissa toimivien ohjelmistokehittäjien välillä siitä, miten ohjelmistoa kehitetään. Jotta malli toimisi, on vaatimusten oltava selkeitä, todennettavia ja itseään toistamattomia. Vaatimusten tulisi käsitellä mm. implementointia, nimeämistä, koodin ulkonäköä sekä arkkitehtuuria ja rakennetta. (Ilkiewicz et. al. 2012, 45) Ei-toiminnalliset vaatimukset voisivat näyttää Taulukon 1 mukaisilta.

Taulukko 1 – Ei – toiminnallisia vaatimuksia

Ominaisuus	**Ominaisuuden alalaji**	**Vaatimus**
Ylläpidettävyys	Luettavuus	Koodissa ei ole pois kommentoitua koodia
Luotettavuus	Poikkeustenkäsittelyn luotettavuus	Koodissa ei tule NullPointerException tyypin poikkeuksia
Luotettavuus	Kattavuus luotettavuus	Kaikilla tiedostoilla on yksikkötestit, jotka kattavat vähintään 70% koodista
Luotettavuus	Data luotettavuus	Koodissa ei vertailla floating pointteja keskenään
Tehokkuus	Koodin tehokkuus	Koodissa ei tehdä kantakutsuja silmukoiden sisältä

Kun laatumalli on saatu määritettyä kaikkine hyvän koodin ominaisuuksineen, voidaan lähteä arvioimaan teknistä velkaa. Tämä tapahtuu kehittämällä jokaiselle vaatimukselle korjaustoimenpide ja aika-arvio sen kestolle. Kaikilla määrittelyillä ei tietenkään ole samaa kestoa, koska esimerkiksi koodin sisentämisen korjaaminen vaatii vähemmän vaivaa, kuin rakenteellisen virheen korjaaminen, joka vielä yleensä vaatii uusien testien kirjoittamista.

(Letouzey 2012, 34) Vaatimusten korjaustoimenpiteet voisivat näyttää Taulukon 2 mukaisilta.

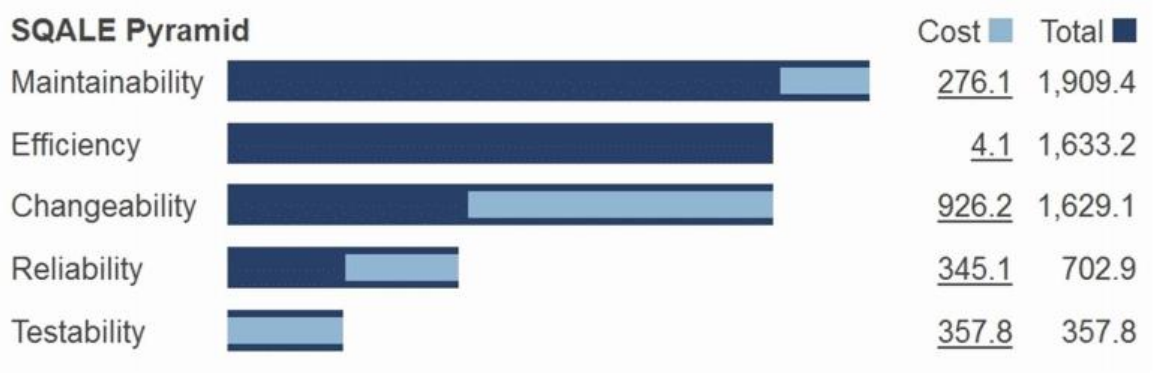
Taulukko 2 – Vaatimusten korjaustoimenpiteitä

Vaatimus	**Korjaustoiminto**	**Kesto**
Koodissa ei ole pois kommentoitua koodia	Poistetaan kommentoitu koodi	2 min / tiedosto
Koodissa ei tule NullPointerException tyypin poikkeuksia	Refaktoroidaan huono koodi	1h / tapaus
Kaikilla tiedostoilla on yksikkötestit, jotka kattavat vähintään 70% koodista	Kirjoitetaan puuttuvat testit, jotta 70% kattavuus täyttyy	20min / rivi
Koodissa ei vertailla floating pointteja keskenään	Refaktoroidaan huono koodi	20min / tapaus
Koodissa ei tehdä kantakutsuja silmukoiden sisältä	Refaktoroidaan koodi siten, että tarvittavat kyselyt pidetään muistissa, kunnes silmukka on saatu päätökseen	2h / tapaus

Korjaustoimintojen ja niiden vaatimien työmäärien määrittämisen jälkeen voidaan lähes suoraan määrittää ohjelmistossa oleva tekninen velka. Tähän tarkoitukseen on olemassa erikoistuneita työkaluja, joiden läpi oma ohjelmisto voidaan ajaa. Jokaiselle ominaisuudelle, joista on esimerkkejä Taulukossa 1, voidaan määrittää velan määrä ja näiden kaikkien summa on kokonaisvelka, jota voidaan kutsua nimellä SQALE quality index (SQI). SQALE määrittää vielä kahdeksan muuta indeksiä, jotka pohjautuvat SQALE:ssä määriteltyihin ominaisuuksiin. Nämä ominaisuudet ovat testattavuus, luotettavuus, muokattavuus, tehokkuus, turvallisuus, ylläpidettävyys, yhdistettävyys ja uudelleenkäytettävyys. (Ilkiewicz et. al. 2012, 45; Letouzey 2012, 34)

SQALE:ssa määritellään muutamia indikaattoreita teknisen velan analysoimiseen. Tällaisia ovat esimerkiksi luokittelu, sekä SQALE pyramidi. Luokittelu on melko yksinkertainen tapa jakaa velkakohteet velan ja kustannusten suhteessa. Luokittelun käyttäminen vaatii kuitenkin oman arviointiruudukon kehittämistä, johon luokittelu voidaan lopulta tehdä. Pyramidi on taas malli, jonka avulla voidaan hahmottaa, miten velka jakautuu eri ominaisuuksien välillä, kertomalla miten paljon missäkin ominaisuudessa on velkaa ja

miten suuri vaiva velan poistamiseksi tarvitaan. Pyramidi antaa priorisointijärjestyksen velan korjaamiseksi. Korjaaminen on hyvä aloittaa testattavuudesta ja sen jälkeen siirtyä ylemmille tasolle. (Letouzey 2012, 34 – 35) Tasojen järjestys on sama, kuin missä ne on lueteltu edellisessä kappaleessa. Esimerkki pyramidista on esitelty kuvassa 1.



Kuva 1 – SQALE pyramidi, jossa ominaisuuksina testattavuus, luotettavuus, muutettavuus, tehokkuus ja ylläpidettävyys (Letouzey 2012, 35)

Jotta yrityksessä saataisiin ohjelmiston ylläpidettävyyttä paremmaksi, voisi SQALE – metodia käyttää joko kokonaisuudessaan, tai osittain. Jos tätä metodia käytetään osittain, ainakin yhteisien sääntöjen tekeminen auttaa ylläpidettävyyteen huomattavasti. Tämä siis tarkoittaa ei – toiminnallisten määrittelyiden tekemistä. Kun sellaiset ovat käytössä, tulee kaikkien ohjelmoijien koodista samantapaista, sekä huono koodi tunnistettaisiin nopeammin. Jos taas käytettäisiin SQALE:a kokonaisuudessaan, on olemassa jo Visual studioon liitettäviä lisäosia, joilla voidaan helposti analysoida koodia ja saada tuotteen tekninen velka selville. Esimerkki tällaisesta lisäosasta on esimerkiksi SonarSourcen SonarC#. Kyseisessä työkalussa on jo valmiiksi määriteltynä joitain sääntöjä velan löytämiseksi. (SonarSource) SQALE on avointa lähdekoodia, joten jonkin SonarC# työkalun kaltaisen ohjelman käyttöönotto voi olla yritykselle kokeiltavissa oleva vaihtoehto.

6 POHDINTA JA TULEVAISUUS

Työn tavoitteena oli löytää ongelmakohtia erään yrityksen ohjelmistotuotteen BL sovelluksen koodista ja lähteä niiden kautta miettimään, miten ohjelmiston ylläpidettävyyttä voidaan parantaa. Samalla selvitettiin, miksi kyseisellä sovelluksella on suorituskykyongelmia. Alusta asti oli epäilys, että ohjelmistossa ei ole vain yhtä ainutta isoa ongelmaa, joka aiheuttaa prosessorikuormaa, vaan ongelman aiheuttajana on monet pienet, vuosien varrella kertyneet kehnot ratkaisut, jotka kaikki yhdessä luovat ongelman.

Nämä pienemmät ongelmat yhdessä ovat oikeasti teknistä velkaa, joten yrityksen tulisi näiden tulosten valossa keskittyä tämän velan hallitsemiseen ja asteittaiseen poistamiseen. Kappaleessa 5 on käsitelty joitain ratkaisuja, joiden avulla voidaan lähteä toimimaan teknisen velan hallitsemiseksi, sekä joitain keinoja, joilla pystytään välttämään teknisen velan syntymistä. Näitä ovat suunnittelun ja dokumentoinnin lisääminen, sekä uusien testaustapojen omaksuminen nykyiseen ohjelmistokehitykseen. Näillä ehdotetuilla ratkaisuilla saadaan parannettua ylläpidettävyyttä ja vähennettyä teknistä velkaa. Kun velkaa aletaan korjata, mitä luultavimmin paranee myös BL sovelluksen suorituskyky paremmin tehdyn koodin myötä.

Työssä käsiteltiin lähinnä yhtä toiminnallisuuskokonaisuutta BL sovelluksen koodissa. Tämä siis tarkoittaa, että suurinta osaa tuotteesta ei ole huomioitu ollenkaan. Tulevaisuudessa tämän työn laajuutta voidaan laajentaa kattamaan koko tuote, eikä vain BL sovellusta. Tätä tukee se, että BL puolelta löytyneet ongelmat viittaavat siihen, että muulla tuotteessa on samantapaisia tai vielä löytämättömiä ongelmia. Kuitenkin teknisen velan hallintaa on kohdassa 5.4 esitellyllä SQALE – menetelmällä helppo tehdä koko järjestelmän laajuisesti. Suurin osa työssä esitetyistä ratkaisumahdollisuuksista on vielä käytännössä toteuttamatta. Tulevaisuudessa olisi siis hyvä, jos saataisiin kokemuksia siitä, miten hyvin kyseiset ratkaisut ovat toimineet ylläpidettävyysmielessä.

Vaikka työssä etsitäänkin osittain vastausta, kuinka BL sovelluksen suorituskykyä voidaan parantaa, ei kaikki työssä esitellyt ratkaisumenetelmät tähän välttämättä pysty suoraan vaikuttamaan. Kuitenkin hyvä koodi on yleensä tehokasta, eli ylläpidettävyyttä parantamalla pystytään parantamaan myös suorituskykyä. Perfview – monitorilla kerätty

data ei välttämättä ole kaikkien luotettavinta, kuten aikaisemmin työssä jo mainittiin. Ajan puitteissa ei ollut mahdollista saada kyseistä ohjelmaa toimimaan oikein annetussa ympäristössä. Tämä siis tarkoittaa, että suurin osa monitorointiohjelmilla löydetyistä ongelmista on huomattu dotTracella. BL sovelluksen ja palvelimen prosessorikuormaa ajatellen tulevaisuudessa olisi hyvä saada Perfview toimimaan oikein palvelimella, koska Perfview on yleisesti kustannustehokkaampi. Perfview on avoimen lähdekoodin ilmainen ohjelma, kun taas dotTrace on JetBrainsin ylläpitämä ja myymä tuote.

Kysymykseen ”Kuinka ohjelmistotuotteen ylläpidettävyyttä voidaan parantaa?” vastauksena on siis tarttua koodikannassa oleviin nykyisiin merkattuihin ongelmiin määrätietoisesti ja välttää samantapaisten, sekä muiden ongelmien syntyminen parantamalla uusien ominaisuuksien suunnittelua ja dokumentointia, sekä lisäämällä testaukseen uusia käytäntöjä ja työkaluja. Näiden lisäksi tulisi ohjelmiston teknistä velkaa lähteä selvittämään ja sen jälkeen hallitsemaan.

Yrityksessä on tämän työn jälkeen jo lähdetty refaktoroimaan ongelmallisia kohteita, sekä yhdenmukaistamaan ohjelmistokehittäjien tapoja. Tuotteen suurimpiin ongelmiin on siis tartuttu ja suorituskykyä ja ylläpidettävyyttä on saatu parannettua. Parannusta on myös saatu aikaan parantamalla BL sovellusta pyörittäviä palvelimia.

7 YHTEENVETO

Tässä työssä tutkittiin erään yrityksen tuotteen BL sovelluksen ongelmia ja vastattiin kysymykseen ”Kuinka ohjelmistotuotteen ylläpidettävyyttä voidaan parantaa?”. Tutkimuksen painopiste oli etenkin kyseisellä palvelimella pyörivän koodin ylläpidettävyyden parantamisessa ja optimoimisessa. Tutkimuksen lähtökohtana käytettiin ongelmaa, jossa palvelimen prosessorikuorma kasvoi epätavallisen paljon tuotteen samanaikaisen käyttäjämäärän lisääntyessä.

Syynä tähän tutkimukseen oli yrityksen tarve selvittää, miksi BL sovellusta pyörittävän palvelimen kuorma kasvoi liikaa tuotteen käyttäjämäärän lisääntyessä, eli haluttiin lähteä parantamaan BL sovelluksen koodin laatua. Ongelmia lähdettiin tutkimaan massatestauksen ja monitorointiohjelmien avulla, jotta saataisiin selville, miksi ohjelma hidastuu. Tutkimusmenetelmänä käytettiin empirical case - study menetelmää, koska tutkittavana kohteena oli vain yhden yrityksen eräs tuote.

Löytyneet ongelmat saatiin analysoimalla monitorointi ohjelmien, Perfview ja dotTrace, dataa. Koodista löytyi optimoimattomia algoritmeja, erittäin pitkiä luokkia ja rajapintoja, sekä dokumentoinnin puutetta. Osa huonoista algoritmeista oli jo valmiiksi koodiin merkattu huonoiksi, mutta niihin ei oltu puututtu. Järjestelmässä tapahtui myös ennen huomaamattomia poikkeuksia ja huomattiin, että ohjelmistokehitysprosessissa ei järjestelmän suorituskyvyn seuraamiseen käytetty tarpeeksi resursseja.

Lähes kaikki koodista löytyneet ongelmat ovat teknistä velkaa, jota pitää lähteä esimerkiksi SQALE:n tapaisilla metodeilla selvittämään tarkemmin ja sen jälkeen hallitsemaan. Testaukseen tulee lisätä uusia toimintatapoja, joilla huono koodi huomattaisiin aikaisemmin, sekä ominaisuuksien dokumentointiin ja tarkempaan suunnitteluun tulee panostaa enemmän. Näillä keinoilla voidaan parantaa ohjelmiston ylläpidettävyyttä, sekä mahdollisesti parantaa järjestelmän tehokkuutta.

Tulevaisuudessa työtä voidaan laajentaa kattamaan kaikki tuotteen osat. BL sovelluksesta löydetyt ongelmat viittaisivat siihen, että muulla ohjelmistossa on samantapaisia ongelmia.

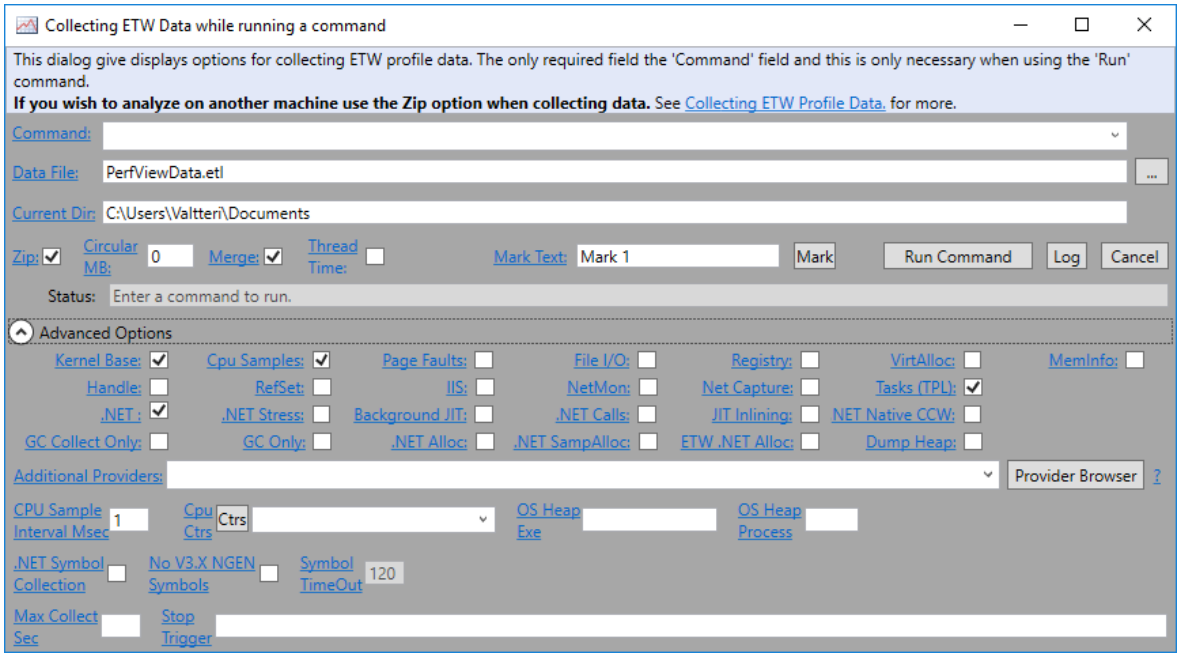
Muusta ohjelmistosta voitaisiin siis suoraan lähteä etsimään samantapaisia ongelmia, kuin mitä tässä työssä on esitelty.

8 LÄHTEET

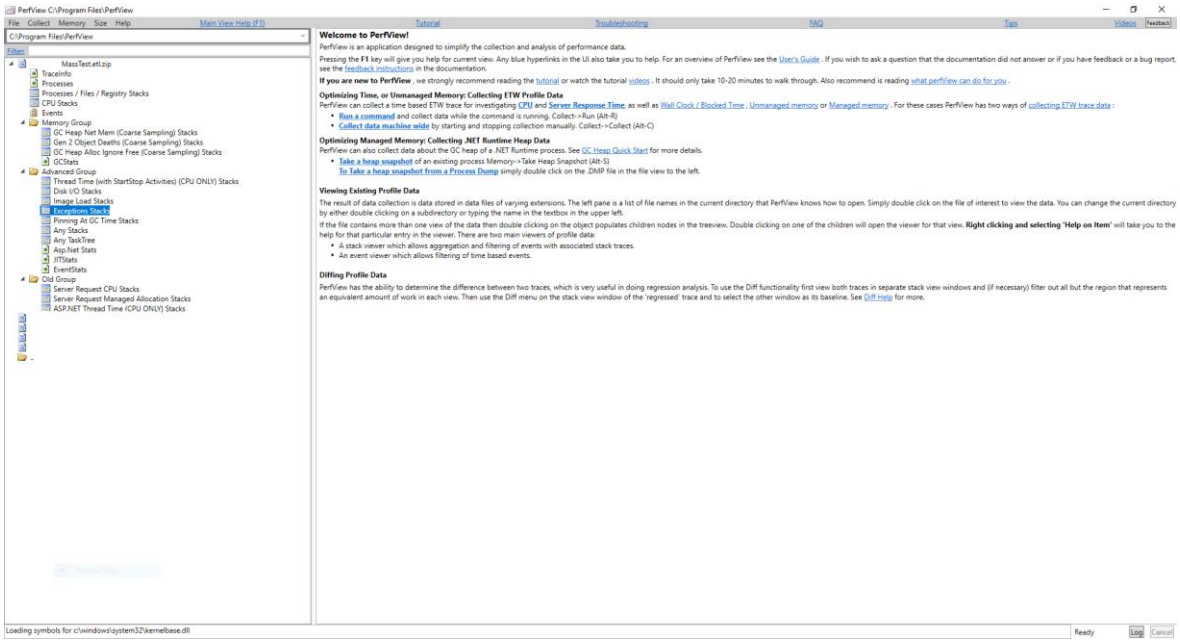
1. Paris Avgeriou, Philippe Kruchten, Robert L. Nord, Ipek Ozkaya, Carolyn Seaman, 2016, Reducing Friction in Software Development
2. Victor R. Basili, Fellow, IEEE, Lionel C. Briand, and Walcelio L. Melo, Member, IEEE Computer Society, 1996, A Validation of Object-Oriented Design Metrics as Quality Indicators
3. Shyam R. Chidamber and Chris F. Kemerer, 1994, A Metrics Suite for Object Oriented Design
4. Golara Garousi, Vahid Garousi-Yusifoglu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, Brian Smith, 2013, Usage and usefulness of technical software documentation: An industrial case study
5. Penny Grubb, Armstrong A Takang, 2003, Software Maintenance - Concepts and Practice, Second Edition
6. Michel Ilkiewicz, Jean-Louis Letouzey, 2012, Managing Technical Debt with the SQALE Method
7. JetBrains, dotTrace, saatavissa https://www.jetbrains.com/profiler/features/, viitattu 26.3.2019
8. Zhen Ming Jiang, Member, IEEE and Ahmed E. Hassan, Member, IEEE, 2015, A Survey on Load Testing of Large-Scale Software Systems
9. Dr. Arvinder Kaur, Kamaldeep Kaur, Dr. Ruchika Malhotra, 2010, Soft Computing Approaches for Prediction of Software Maintenance Effort
10. Jean-Louis Letouzey, 2012, The SQALE Method for Evaluating Technical Debt
11. Microsoft, Entity Framework, (2016) saatavissa <https://docs.microsoft.com/en-us/ef/ef6/>, viitattu 15.3.2019
12. Microsoft Github, Perfview, saatavissa <https://github.com/Microsoft/perfview>, viitattu 22.1.2019
13. Md. Monzur Morshed, Md. Arifur Rahman, Salah Uddin Ahmed, 2012, A Literature Review of Code Clone Analysis to Improve Software Maintenance Process

14. Mikhail Perepletchikov, Member, IEEE, and Caspar Ryan, 2011, A Controlled Experiment for Evaluating the Impact of Coupling on the Maintainability of Service-Oriented Software
15. SonarSource, SonarC, https://www.sonarsource.com/products/codeanalyzers/sonarcsharp.html, viitattu 18.3.2019
16. SQALE, saatavissa <http://www.sqale.org/details>, viitattu 17.3.2019
17. Jeffrey Voas, 1998, Maintaining Component-Based System
18. Yang-Ming Zhu, 2016, Software Reading Techniques: Twenty Techniques for More Effective Software Review and Inspection
19. Technopedia, Technical debt, saatavissa <https://www.techopedia.com/definition/27913/technical-debt>, viitattu 2.4.2019

Liite 1. Monitorointiohjelmat ja aineistoa



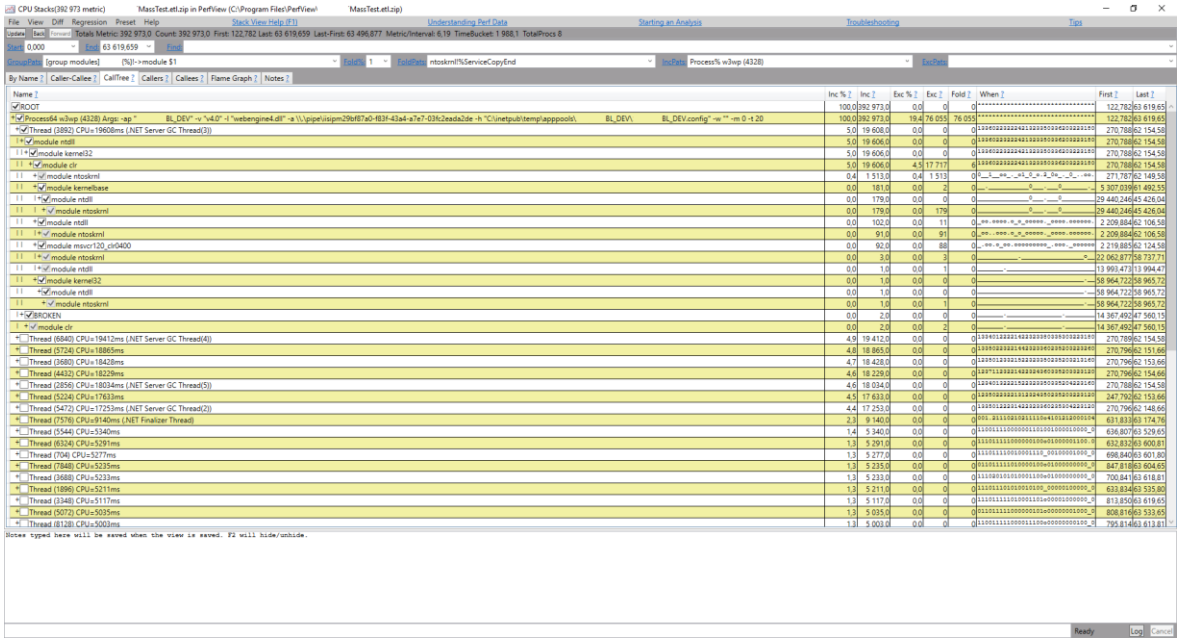
Kuva 2 – PerfViewin käynnistysnäkymä



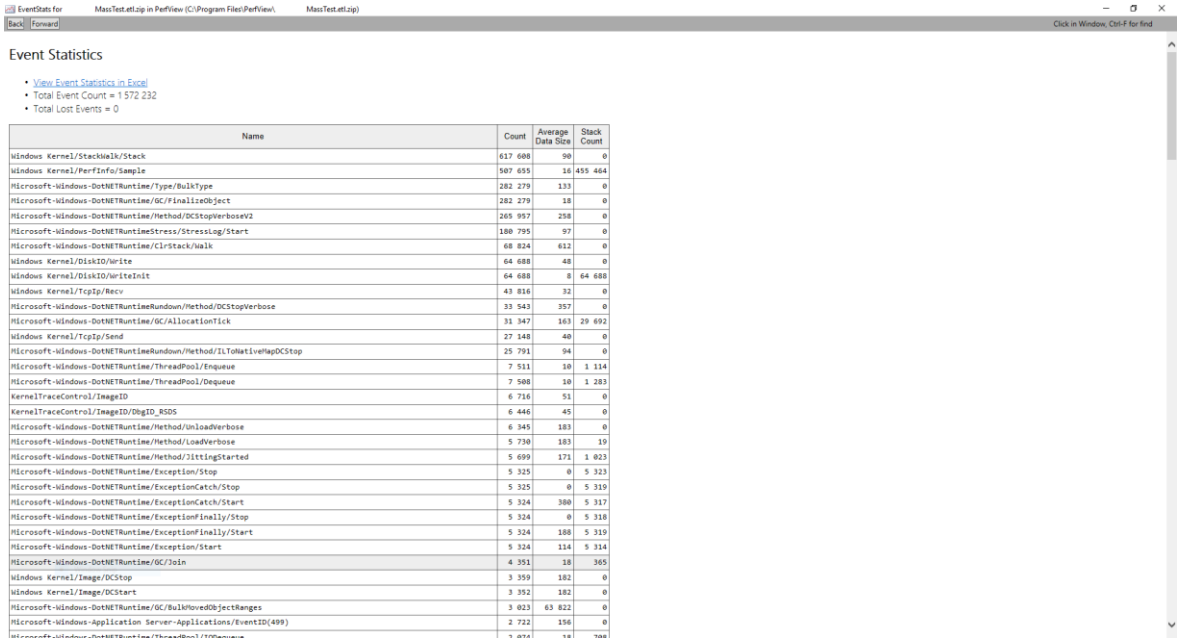
Kuva 3 - PerfViewin tulokset ajon jälkeen

(jatkuu)

Liite 1. (jatkoa)



Kuva 4 - PerfViewin antamat CallStackeja BL prosessista



Event Statistics

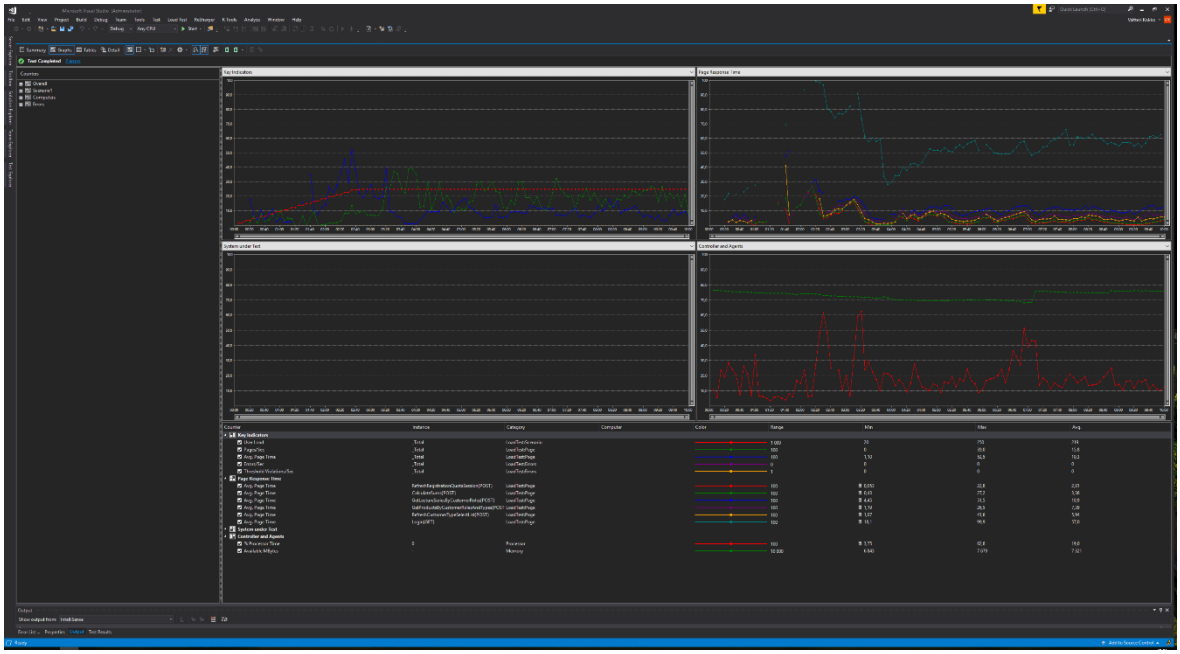
- View Event Statistics in Excel
- Total Event Count = 1 572 232
- Total Lost Events = 0

Name	Count	Average Data Size	Stack Count
Windows Kernel/StackWalk/Stack	617 608	90	0
Windows Kernel/PerfInfo/Sample	507 655	16	455 464
Microsoft-Windows-DotNETRuntime/Type/BulkType	282 279	133	0
Microsoft-Windows-DotNETRuntime/GC/FinalizeObject	282 279	18	0
Microsoft-Windows-DotNETRuntime/Method/DCStopVerboseV2	265 957	258	0
Microsoft-Windows-DotNETRuntimeStress/StressLog/Start	180 795	97	0
Microsoft-Windows-DotNETRuntime/ClrStack/Walk	68 824	612	0
Windows Kernel/DiskIO/Write	64 688	48	0
Windows Kernel/DiskIO/WriteInit	64 688	8	64 688
Windows Kernel/TcpIp/Recv	43 816	32	0
Microsoft-Windows-DotNETRuntimeRundown/Method/DCStopVerbose	33 543	357	0
Microsoft-Windows-DotNETRuntime/GC/AllocationTick	31 347	163	29 692
Windows Kernel/TcpIp/Send	27 148	40	0
Microsoft-Windows-DotNETRuntimeRundown/Method/ILToNativeMapDCStop	25 791	94	0
Microsoft-Windows-DotNETRuntime/ThreadPool/Enqueue	7 511	10	1 114
Microsoft-Windows-DotNETRuntime/ThreadPool/Dequeue	7 508	10	1 283
KernelTraceControl/ImageID	6 716	51	0
KernelTraceControl/ImageID/DbgID_RSDS	6 446	45	0
Microsoft-Windows-DotNETRuntime/Method/UnloadVerbose	6 345	183	0
Microsoft-Windows-DotNETRuntime/Method/LoadVerbose	5 730	183	19
Microsoft-Windows-DotNETRuntime/Method/JittingStarted	5 699	171	1 023
Microsoft-Windows-DotNETRuntime/Exception/Stop	5 325	0	5 323
Microsoft-Windows-DotNETRuntime/ExceptionCatch/Stop	5 325	0	5 319
Microsoft-Windows-DotNETRuntime/ExceptionCatch/Start	5 324	380	5 317
Microsoft-Windows-DotNETRuntime/ExceptionFinally/Stop	5 324	0	5 318
Microsoft-Windows-DotNETRuntime/ExceptionFinally/Start	5 324	188	5 319
Microsoft-Windows-DotNETRuntime/Exception/Start	5 324	114	5 314
Microsoft-Windows-DotNETRuntime/GC/Join	4 351	18	365
Windows Kernel/Image/DCStop	3 359	182	0
Windows Kernel/Image/DCStart	3 352	182	0
Microsoft-Windows-DotNETRuntime/GC/BulkMovedObjectRanges	3 023	63 822	0
Microsoft-Windows-Application Server-Applications/EventID(499)	2 722	156	0

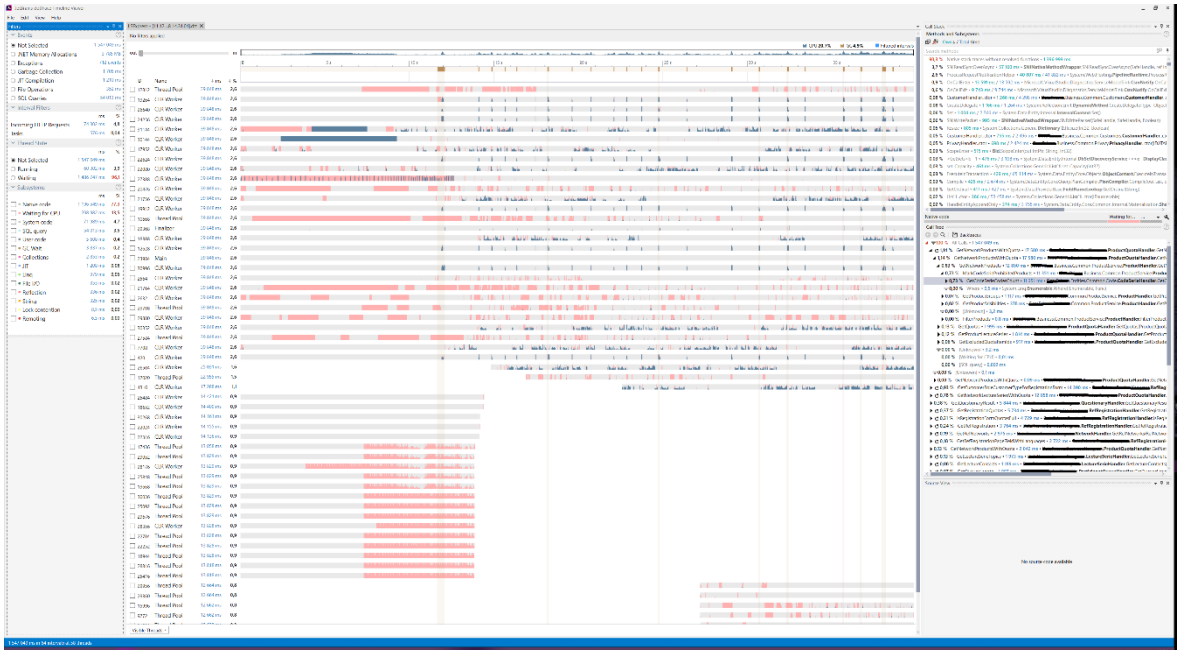
Kuva 5 - PerfViewin antamia koostettua dataa

(jatkuu)

Liite 1. (jatkoa)



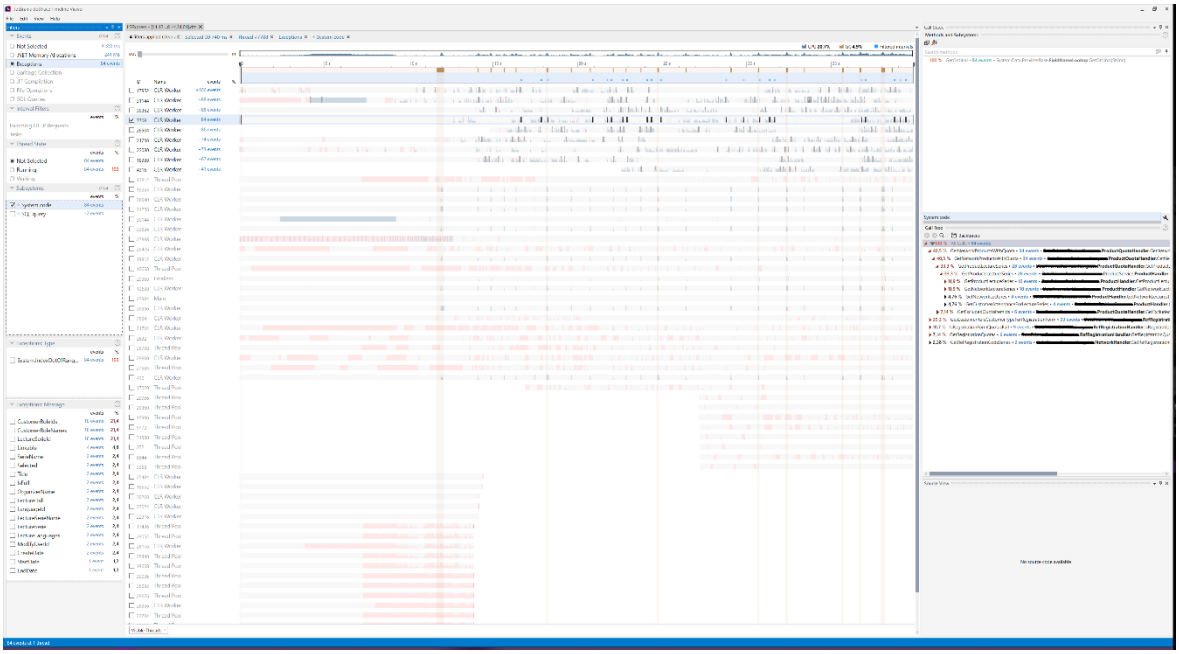
Kuva 6 - Monitorointiohjelmalla tarkkaillun massatestin tuloksia



Kuva 7 - dotTracella kerättyä dataa

(jatkuu)

Liite 1. (jatkoa)



Kuva 8 - dotTracen osoittamia poikkeuksia järjestelmässä