Tampereen yliopisto

Timo Juutilainen

VANHAN TIETOKANTAJÄRJESTELMÄN VERSIOHALLINTA

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Huhtikuu 2019

TIIVISTELMÄ

TIMO JUUTILAINEN: Vanhan tietokantajärjestelmän versiohallinta
Tampereen yliopisto
Diplomityö, 57 sivua
Huhtikuu 2019
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Yliopistonlehtorit Timo Aaltonen ja Timo Poranen

Avainsanat: versiohallinta, tietokanta, tekninen velka, subversion

Työssä esitellään tietokantaa normaalia enemmän hyödyntävä legacy-järjestelmä. Järjestelmää kehittäessä ei ole käytetty versiohallintaa ollenkaan, ja myöhemminkin vain varmuuskopionäkökulmasta. Järjestelmän ohjelmistologiikka on suurelta osin toteutettu tietokantojen proseduureissa, funktioissa ja liipaisimissa. Järjestelmästä on tehty julkaisuja asiakastoimituksia varten kopioimalla vanhoja toimituksia varten tehtyjä julkaisuja ja ajan kuluessa järjestelmän versiohistoria on hämärtynyt tai unohtunut kokonaan.

Työssä pyritään tunnistamaan tästä aiheutuvia ongelmia ja perustelemaan, miksi ne ovat haitallisia. Ongelmat ovat merkittäviä ja aiheuttavat haittaa eri tavalla näkökulmasta riippuen. Ohjelmistokehittäjien työ vaikeutuu ja muuttuu epämiellyttävämmäksi, kun eri versioiden tilan selvittämiseen täytyy käyttää aikaa. Jo tehdyn kehitystyön unohtuminen aiheuttaa päällekkäisen työn tekemistä ja ylipäätään järjestelmän ohjelmistotekninen laatu laskee. Projektinjohtonäkökulmasta työn määrittely ja suunnittelu vaikeutuvat samoista syistä, sekä liiketoimintanäkökulmasta kaikkeen päällekkäiseen ja ylimääräiseen työhön kuluva aika maksaa rahaa.

Ongelmien tunnistamisen jälkeen etsitään ratkaisukeinoja ohjelmistotieteen kirjallisuudesta. Erityisesti paneudutaan tapauksiin, joissa on ratkaistu vastaavia ongelmia. Voidaan todeta, että aiheesta on kirjoitettu vähän verrattuna versiohallintaan yleisesti. Suurin haaste tietokantojen versiohallinnoinnissa on ohjelmistologiikan muuttaminen versiohallintaa tukeviksi tiedostoiksi. Ratkaisujen oleellisin osuus onkin työkalujen käyttö tämän vaiheen helpottamiseksi. Keinoja, jotka tekisivät tietokantojen versionhallinnoinnista yhtä helppoa kuin tyypillisen ohjelmistokoodin versiohallinnoinnista ei löytynyt.

Seuraavaksi esitellään joukko toimenpiteitä esitellyn järjestelmän tilan parantamiseksi. Toimenpiteet keskittyvät päällekkäisen työn välttämiseen, versiohistorian selventämiseen ja ohjelmistokoodin tason parantamiseen. Toimenpiteiden työmäärä arvioidaan *planning poker* -menetelmällä ja niistä valitaan toteutettavaksi kustannustehokkaimmat.

Toimenpiteiden toteutuksen jälkeen niiden toteutusta ja tehoa arvioidaan. Osa niistä toteutettiin eri tavalla kuin oli suunniteltu, joitain jäi toteuttamatta ja lisäksi tehtiin joitain suunnittelemattomia toimenpiteitä. Voidaan todeta, että päällekkäisen kehitystyön tekeminen ja versiohistorian hämärtyminen saatiin estettyä, mutta itse prosessin ylläpitämisestä tuli työläämpää. Jatkoehdotuksissa ehdotetaan korjauksia tähän prosessiin ja organisaatioon yleensä. Lopuksi todetaan, että vanhaa järjestelmää ei kannata korjata liian työläästi ja ongelmiin päätyminen voidaan alkujaan välttää poistamalla juurisyynä toiminut resurssipula.

ABSTRACT

TIMO JUUTILAINEN: Version control of an old database system
Tampere University
Master of Science Thesis, 57 pages
April 2019
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiners: University lecturers Timo Aaltonen and Timo Poranen

Keywords: version control, database, technical debt, subversion

This thesis describes a database-centric legacy system. The system wasn't originally under version control at all and while version control was introduced later, it was backup-minded in nature. Business logic of the systesm was largely coded in database procedures, functions and triggers. Deployments of the system were done by copying previously deployed databases and over time the system's version history was blurred or entirely forgotten.

Problems caused by this are identified and their harmfulness is described. The problems are significant and cause harm different ways depending on the point of view. Programmers find that their work becomes more difficult and less appealing when they are wasting time figuring out the state of different environments, end up doing extra work because of forgotten features and the state of the software code in general deteriorates. Project management will find specification and planning more difficult for the same reasons and businesswise all wasted time ends up costing money.

After identifying the problems solutions are looked from the software engineering theory. Cases where similar problems have been solved are especially looked for. It can be said, that much less is written about version controlling databases than on version control in general. The biggest challenge when version controlling databases is extracting the business logic from the database into version controllable scripts. The key part in various solutions is using tools to make this part easier. Methods that would make database version control as easy as version control in general cannot be found, though.

Next a set of actions is introduced to improve the state of the described system. The actions aim at helping to avoid overlapping work, make the version history more clear and improve level of the software code in general. Workload of the actions is estimated using the *planning poker* method and the most cost effective ones are selected to be implemented.

Implementation and effectiveness of these actions are evaluated. Some of the actions were implemented differently than planned, some weren't implemented at all and some non-planned actions were implemented. It could be acknowledged that overlapping development work could be prevented and version history became clearer, but maintaining the process created more workload overhead. Next changes to this process and organization in general are suggested. Finally, it is pointed out that not too much time should be spent fixing a legacy system and the root cause of problems can be avoided altogether by allocating more resources to the development.

ALKUSANAT

Tätä diplomityötä lukiessa on syytä muistaa, että se keskittyy vain ja ainoastaan kehitystä vaativiin seikkoihin. Työssä käsiteltävä järjestelmä on osa suurempaa hajautettua järjestelmää, joka on auttanut ratkaisemaan monia ongelmia eri asiakkaille vaihtelevissa käyttötapauksissa. Tuote on markkinajohtaja sektorillaan Suomessa ja se on käytössä monissa muissa Euroopan maissa. Se on myös kaupallinen menestys.

Työssä ei ole tarkoitus myöskään moittia ketään järjestelmää nykyään tai ennen kehittänyttä henkilöä. Kuten niin usein IT-alalla, myös tässä työssä käsiteltävä järjestelmä on kehitetty ja ylläpidetty alan reaaliteeteistä aiheutuvien resurssi- ja kustannuspaineiden alla. On epärealistista olettaa, että tällöin päästään täydellisyyteen. Myöhemmin haasteellisiksi osoittautuneita ratkaisuja on alun perin tehty järkevillä perusteluilla. Myös työn kirjoittaja on osaltaan vastuussa järjestelmän kehityksessä tehdyistä virheistä sekä ohjelmistokehittäjänä että projektipäällikkönä. Jälkiviisaus on aina helppoa.

Kirjoittamisen aikana yliopisto, johon työ palautetaan, vaihtui. Työ on kirjoitettu aloitushetkellä käytössä olleelle TTY:n opinnäytetyöpohjalle ja siihen on vaihdettu uusi kansilehti. Lisäksi RefWorksissä käytetty TUT numerical -viittaustyyli lakkasi toimimasta ja päädyin korvaamaan sen geneerisellä Vancouver-tyylillä. Pyrin käyttämään suomenkielisiä termejä aina, kun se oli mahdollista. Kaikille termeille ei kuitenkaan ole järkevää ja riittävän tunnettua käännöstä. En ihmettele, että aika ajoin saa lukea huolestuneita uutisia englannin vaikutuksesta suomen kieleen.

Kiitokset työn tarkastaneille sekä ohjanneille Timo Aaltoselle ja Timo Poraselle neuvoista sekä korjausehdotuksista, työnantajalle diplomityön tekemisen mahdollistamisesta ja perheelle tuesta. Erityisesti kiitos äidille oikolukemisesta.

Tampereella, 4.4.2019

Timo Juutilainen

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. JÄRJESTELMÄ LÄHTÖTILANTEESSA 3
2.1 Järjestelmässä käytetyt teknologiat 3
2.2 Järjestelmän tekninen kuvaus 4
2.3 Tapa tehdä toimituksia 6
2.4 Versiohallinnan rooli 7
2.5 Tietokantapohjaisuus ja sen erityispiirteet 8
2.6 Miten tilanteeseen on päädytty? 10
3. ONGELMAT LÄHTÖTILANTEESSA 11
3.1 Ongelmat ohjelmistokehittäjälle 11
3.2 Ongelmat projektipäällikölle 12
3.3 Ongelmat liiketoiminnalle 13
4. KIRJALLISUUSKATSAUS JA MUIDEN RATKAISUT 15
4.1 Ohjelmistotuotannon teoreettinen näkökulma 15
4.1.1 Tyypilliset puutteet ja ongelmat 16
4.1.2 Kehitysympäristön parantaminen 16
4.1.3 Ratkaisuja ongelmiin 17
4.1.4 Muutosten riskejä ja rajoitteita 19
4.2 Redgaten ratkaisu 19
4.2.1 Tietokantojen versiohallinta 20
4.2.2 Tietokantojen käyttöönottoympäristöt ja toimenpiteet 21
4.2.3 Käyttöönoton automatisointi 23
4.3 Ideaaliratkaisu 23
4.3.1 Siirrytään pois tietokantakeskeisyydestä 24
4.3.2 Käytetään moderneja tuotteen toimitustapoja 24
4.3.3 Siirrytään käyttämään Git-versiohallintajärjestelmää 25
4.3.4 Tehdään muutoksia frontend-toteutukseen 26
4.3.5 Yhteenveto ideaaliratkaisusta 26
5. PÄÄTETYT TOIMENPITEET 28
5.1 Tavoitteet 28
5.2 Versiohallinnan rakenteen muutokset 28
5.3 Julkaisujen tekeminen 30
5.4 Moduulirakenteen palauttaminen alkutilaan 31
5.5 Moduulirakenteen käytön laajentaminen 31
5.6 Logiikan siistiminen 32
5.7 Luodaan koodistandardi 32
5.8 Kirjoitetaan kuvaus tyypillisestä toimitusprojektista 33
5.9 Parannetaan tietokannan tasoa yleisesti 33
5.10 Luodaan muistilista tehtävistä asioista 34

5.11 Toimenpiteiden työmääräarvio 34
6. TULOKSET 36
6.1 Toteutetut suunnitellut toimenpiteet 36
6.1.1 Versiohallinnan rakenteen muutokset 36
6.1.2 Julkaisujen tekeminen 37
6.2 Suunnitellut toteuttamattomat toimenpiteet 37
6.2.1 Logiikan siistiminen 37
6.2.2 Luodaan koodistandardi 38
6.2.3 Kirjoitetaan kuvaus tyypillisestä toimitusprojektista 38
6.2.4 Parannetaan tietokannan tasoa yleisesti 38
6.2.5 Luodaan muistilista tehtävistä asioista 38
6.3 Suunniteltujen muutosten lisäksi tehdyt toimenpiteet 39
6.3.1 Tuotteen teknisen laadun heikkenemisen lopettaminen 39
6.4 Tilanne muutosten jälkeen 39
7. TULOSTEN ARVIOINTI 42
7.1 Ero alkutilanteeseen 42
7.2 Ero ideaaliratkaisuun 43
7.3 Muutos- ja jatkoehdotukset 44
7.3.1 Korjaukset tapaan käyttää versiohallintaa 44
7.3.2 Tehdään suunnitelma uuden järjestelmän kehitysaikataulusta 45
7.3.3 Siistitään tärkein osa ohjelmistokoodista 46
7.3.4 Selvitetään parempia tapoja käyttää kehitystietokantoja 46
7.3.5 Muutetaan organisaatiota kehitystiimien ajan myymiseksi 47
7.3.6 Teknisen velan määrän arvioiminen ja tekeminen näkyväksi 48
7.4 Kannattiko vanhan korjaaminen? 48
7.5 Miten lähtötilanteeseen päätyminen voidaan välttää? 50
8. YHTEENVETO 52
LÄHTEET 55

LYHENTEET JA TERMIT

ACID	Atomicity, Consistency, Isolation and Durability
AJAX	Asynchronous JavaScript and XML
CD	Continuous Delivery, jatkuva toimitus
CI	Continuous Integration, jatkuva integraatio
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
MSSQL	Microsoft SQL Server
ORM	Object-relational Mapping
PC	Personal Computer
PHP	PHP: Hypertext Preprocessor
SaaS	Software as a Service
SQL	Structured Query Language
SVN	Apache Subversion, kts. Subversion
TDD	Test-Driven Development, testivetoinen kehitys
T-SQL	Transact SQL
UI	User Interface, käyttöliittymä
UML	Unified Modeling Language
WWW	World Wide Web, internetissä toimiva hypertekstijärjestelmä

Backend	Taustajärjestelmä
Devops	Toimintamalli tuotantoympäristöjen ketterään ylläpitämiseen
Frontend	Etujärjestelmä
Jira	Atlassianin tehtävienhallintaohjelmisto
LabVIEW	National Instrume
Planning poker	Menetelmä työmäärän arviointiin
Redgate	Redgate Software, tietokantojen hallintaohjelmistoja kehittävä yhtiö
Repositorio	Repository, tietovarasto
Tietovarasto	Versiohallinnan paikka tiedon tallentamiseen
Trello	Atlassianin tehtävienhallintaohjelmisto
Scrum	Ketterän ohjelmistokehityksen projektinhallinnan viitekehys
Subversion	Apache Subversion, versiohallintajärjestelmä
Wiki	Verkkosivu, jonka sisällön tuottavat sen käyttäjät

1. JOHDANTO

Tässä diplomityössä esitellään versiohallinnan ulkopuolella oleva tietokantapohjainen legacy-järjestelmä ja kuvataan sen toimenpiteitä edeltävä tilanne. Diplomityön tutkimusongelmana on järjestelmän versiohallinnoimattomasta tilasta aiheutuvien ongelmien toteaminen ja ratkaisujen etsiminen. Lisäksi pyritään tunnistamaan syitä, jotka aiheuttivat ongelmia ja erityisesti löytämään tapoja, joilla tilanteeseen päätyminen voidaan välttää. Vastaavien ongelmien voidaan arvioida olevan yleisiä ohjelmistotekniikan alalla. Erityisnäkökulman työhön tuo se, että tietokanta on tarkastelun kohteena olevassa järjestelmässä hyvin keskeisessä osassa.

Esiteltävässä järjestelmän kehityksessä ja toimituksissa on voimassa prosessi, joka heikentää tuotteen laatua, ylläpidettävyyttä ja ohjelmistokoodin uudelleenkäytettävyyttä. Kaikki tämä kasvattaa sekä kehitystyön että järjestelmästä tehtävien asiakastoimitusten työmäärää ja siis kustannuksia. Se tekee kehittäjien arkityöstä epämiellyttävämpää ja raskaampaa.

Työssä käsiteltäviä tutkimuskysymyksiä ovat:

- Miten vallitseva tapa toimia eroaa ohjelmistotuotannon alalla yleisesti hyväksi todetuista käytännöistä?
- Mitä ongelmia tästä poikkeamasta aiheutuu?
- Miten tilanteeseen on ajauduttu ja miten siihen ajautuminen voitaisiin välttää?
- Miten ongelmat voidaan ratkaista?
- Mitä päätettiin tehdä ja mitä vaikutusta toimenpiteillä oli?

Työssä kerrotaan tilasta aiheutuneet ongelmat, käydään läpi mitä ohjelmistotuotannon teoria kertoo aiheesta ja listataan päätetyt toimenpiteet. Toimenpiteitä pyritään löytämään sekä ohjelmistotieteen teoreettisesta materiaalista, että käytännön ratkaisuista, joita muut ovat toteuttaneet vastaavien ongelmien ratkaisemiseksi. Toimenpiteiden työmäärä arvioitiin ja niistä valittiin toteutettavaksi kaikista hyödyllisimmät ja työmäärältään pienimmät - eli kustannustehokkaimmat.

Työn kirjoitushetkellä ensimmäisessä vaiheessa toteutettavaksi valitut muutokset on toteutettu. Näiden muutosten tuloksia arvioidaan verrattuna alkutilanteeseen, suunnitelmaan ja ideaalitilanteeseen. Arvioidaan myös, kuinka hyvin toimenpiteiden toteutus vastaa etukäteen suunniteltua.

Lopuksi tuloksien vaikutusta arvioidaan ja esitetään jatkotoimenpiteitä sekä muutosehdotuksia. Muutosehdotukset kohdistuvat sekä tässä diplomityössä esiteltyjen toimenpiteiden toteutuksen kehittämiseen että sen ulkopuolisiin työn aikana tunnistettuihin seikkoihin. Samalla pohditaan ongelmia aiheuttaneita syitä ja tapoja välttää ne jo ennen ongelmien syntymistä. Yksi oleellinen näkökulma johtopäätöksissä onkin se, kuinka paljon työtä kannattaa käyttää vanhan järjestelmän korjaamiseen suhteessa kokonaan uuden kehittämiseen, ja jos uusi järjestelmä kehitetään, miten vältetään samoihin ongelmiin joutuminen uudestaan.

2. JÄRJESTELMÄ LÄHTÖTILANTEESSA

Tässä luvussa kuvataan järjestelmä ja sen versiohallinta kuten se oli ennen toimenpiteitä. Vaikka luvussa käydään lyhyesti läpi kaikki kokonaistuotteen komponentit, tässä diplomityössä käsitellään vain paikallisympäristöjen järjestelmän ongelmia. Kaikkia tuotteen komponentteja leimaa tietokantakeskeisyys, mutta siitä johtuvat ongelmat ovat suurimmillaan vain tässä komponentissa.

2.1 Järjestelmässä käytetyt teknologiat

Keskeisessä osassa käsiteltävää järjestelmää on relaatiotietokanta, jossa taulujen välistä dataa yhdistellään toisiinsa yhteyksillä eli relaatioilla [1]. Relaatiotietokantaa hallitaan yleensä *SQL* (Structured Query Language) -kyselykielen avulla. On olemassa monia eri relaatiotietokantoja. Tässä järjestelmässä käytetään vain *Microsoftin SQL Server* -tietokantahallintajärjestelmää, jonka kyselykielenä on *Transact-SQL (T-SQL)* [2].

Käytetty tietokantahallintajärjestelmä tukee lisäksi SQL-kyselykielisten proseduurien, funktioiden ja liipaisimien tekemistä. Proseduureja ja funktioita voidaan kutsua sekä tietokannan ulkopuolelta että sen sisältä. Niiden ero on se, että funktiot eivät voi tehdä tietokannan sisältöä muokkaavia toimenpiteitä. Liipaisimia puolestaan kutsutaan, kun niihin liitettyihin tauluihin tehdään liipaisimissa määritettyjä toimenpiteitä. Näihin objekteihin on tallennettu haluttu ohjelmistologiikka, joka on kirjoitettu SQL-kyselykielellä, ja juuri tätä ominaisuutta käytetään tässä työssä käsiteltävässä järjestelmässä runsaasti hyväksi. [1]

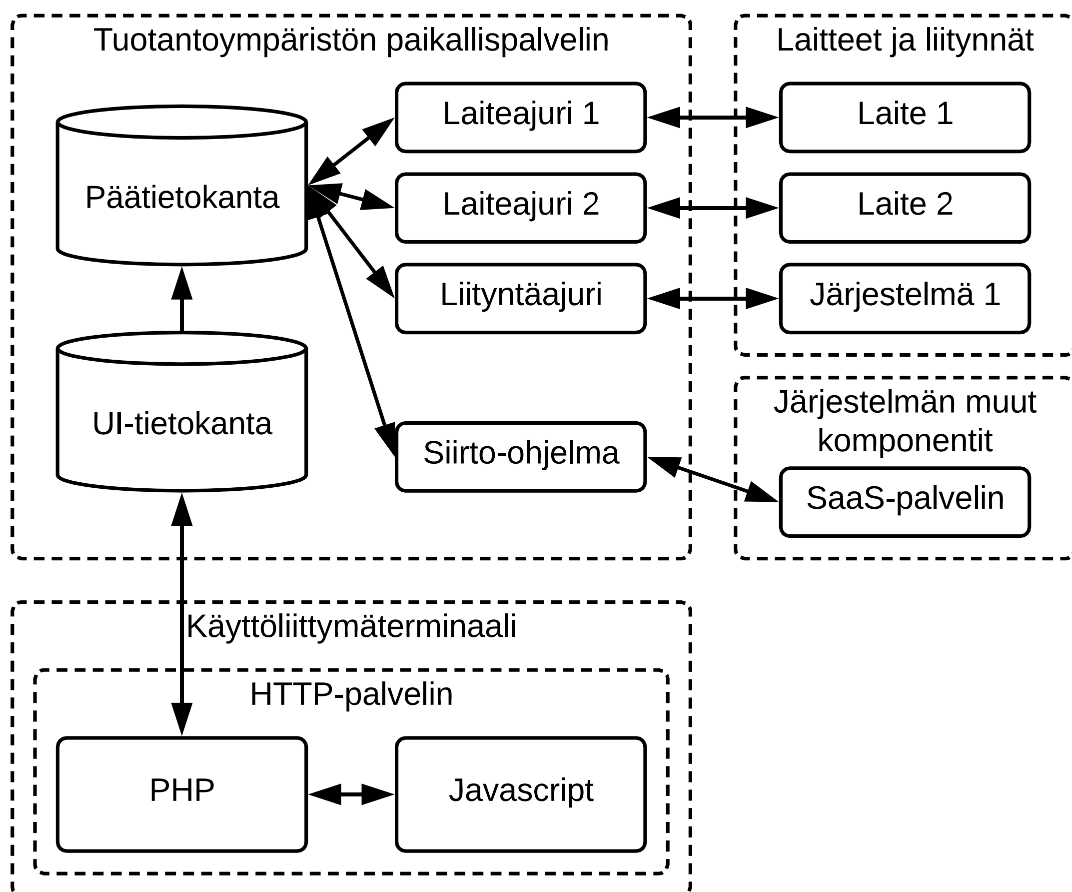
Järjestelmä on lähtötilanteessa kytkettynä *Apachen Subversion (SVN)* –versiohallintajärjestelmään [3]. SVN on keskustietovarastopohjainen tapa hallita tiedostoja niiden eri versioissa. Kehittäjät ottavat oman kopion keskustietovarastosta, tekevät siihen muutoksia ja tallentavat niitä takaisin [4]. Julkaisuja tehdään ottamalla kopioita kehityshaarasta. Tässä työssä käsiteltävä järjestelmä on lähtötilanteessa versiohallinnoitu lähinnä varmuuskopiotyylisesti eikä SVN:n hyödyllisiä ominaisuuksia käytetä hyväksi.

Vaikka suurin osa järjestelmän ohjelmistologiikasta on tietokannan objekteissa, tarvitaan silti frontend- ja backend-järjestelmät. Frontend-järjestelmää käytetään asiakaspäässä, kun taas backend-järjestelmää palvelinpäässä. Järjestelmässä backend-tekniikkana käytetään *PHP (PHP: Hypertext Preprocessor)* -ohjelmointikieltä [5]. Frontend puolestaan on toteutettu *Javascript, HTML (Hyper Text Markup Language) ja CSS (Cascading Style Sheets)* -ohjelmointikielillä [6-8]. Järjestelmän *HTTP (Hypertext Transfer Protocol)* -palvelimena käytetään *Apache HTTP Serveriä* [9].

2.2 Järjestelmän tekninen kuvaus

Käsiteltävä järjestelmä on yksi osa suuremman tuotteen kokonaisuudessa. Tuote koostuu asiakkaalle toimitettavasta WWW-liittymästä, asiakkaan toimittajien käyttämästä yhteisestä WWW-liittymästä, eri toimijoiden käyttämästä mobiiliohjelmistosta ja asiakkaiden ympäristöön asennettavasta, kosketusnäytöillä käytettävästä käyttöliittymästä sekä laitoksen käyttöön liittyvistä taustatoiminnoista. Tämä diplomityö käsittelee näistä viimeistä.

Järjestelmää käytetään tiedon keräämiseen tuotteen alimmalla tasolla. Sitä käyttävät kosketusnäyttöpaneelien avulla tuotteen ostaneet asiakkaat ja heidän toimittajansa. Koska järjestelmä asennetaan asiakkaan tuotantotilojen läheisyyteen sekä fyysisesti että verkkoteknisesti, käytetään sitä myös näihin tiloihin sijoitettujen laitteiden liityntöihin.



Kuva 1. *Järjestelmän tekninen rakenne.*

Laitokselle asennettava järjestelmä tekee karkeasti ryhmiteltynä kolmea asiaa: tarjoaa kosketuskäyttöliittymät käyttäjien käyttöön, hallitsee laiteajureita ja ylläpitää paikallisesti käytettäviä ohjelmistorajapintoja. Järjestelmän tekninen rakenne suhteessa tietokantaan on eritelty kuvassa 1. Kaikkien näiden toimintojen ohjelmistologiikka on toteutettu backendin tietokannassa.

Järjestelmässä on Javascript-ohjelmointikielellä toteutettu frontend-osuus, joka generoi komponentteja tietokannasta saamillaan tiedoilla. Komponenttien pohjatoteutus on tehty Javascript-osassa, mutta niiden käyttö ja takaisinkutsut on määritelty tietokannassa. Esimerkiksi nappia generoitaessa määritellään tietokantaan sen sijainti, teksti, mahdolliset muotoilut, ikonit ja proseduuri, jota kutsutaan nappia painettaessa. Haluttu toiminnallisuus toteutetaan takaisinkutsuproseduuriin.

Komennot ja pyynnöt välitetään tietokantaan PHP-ohjelmointikielellä toteutetun, hyvin kevyen backend-osion kautta. Frontendin Javascript-osio kutsuu PHP-toteutusta AJAX (Asynchronous JavaScript and XML) -toimintojen avulla. Kutsut on jo alun perin kirjoitettu tietokantakyselyinä ja PHP-toteutus vain ajaa ne siinä tietokannassa, jota käyttöliittymä on konfiguroitu käyttämään. Näin saadaan kuitenkin toteutettua dynaamisia muutoksia frontendissä generoituun HTML-toteutukseen.

Apache HTTP -palvelin on tyypillisesti pystytetty kosketusnäyttöterminaalin paneeli-PC:lle. Itse käyttöliittymän frontend on paneeli-PC:n kioski-tilassa käytettävässä selaimessa. Kioski-tilassa selaimen ikkunan palkit on piilotettu. Frontend on yhteydessä saman PC:n HTTP-palvelimella olevaan backend-toteutukseen. Käytettävä tietokanta on tyypillisesti asennettuna paikallispalvelimelle asennettuun Microsoft SQL Server -instanssiin.

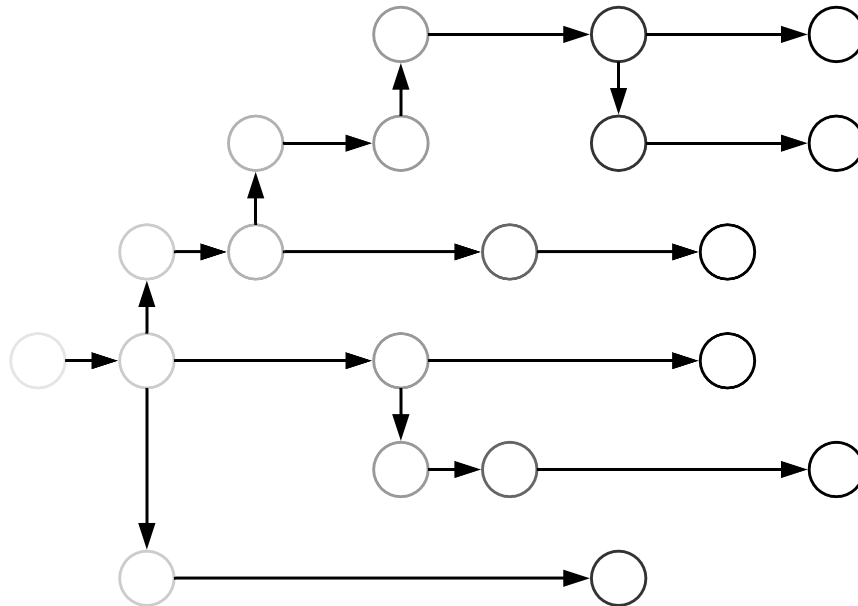
Laite- ja liityntäajurit sekä siirto-ohjelmat on pääasiallisesti toteutettu National Instrumentsin *LabVIEW*-ohjelmointiympäristöllä [10]. Myös niiden parametrit ja mahdolliset takaisunkutsut konfiguroidaan ja toteutetaan tietokannassa.

Tietokanta on siis järjestelmän merkittävin yksittäinen osa. Lähes kaikki toiminnallisuus on toteutettu sen proseduureihin, funktioihin ja liipaisimiin, ja näihin myös tehdään kaikki tyypillinen kehitystyö. Kanta johon käyttöliittymä tai ajurit ovat yhdistettynä voidaan vaihtaa toiseen ilman muuta kehitystyötä ja samalla niiden toiminta muuttuu uuden tietokannan konfigurointien mukaiseksi.

Todellisuudessa kantoja on useita. Järjestelmän varsinainen päätietokanta pitää sisällään kaiken datan, mutta rakenteen selkiyttämiseksi tämän pääkannan toiminnallisuutta on jaettu erillisiin tietokantoihin. Tiettyä spesifiä toiminnallisuutta tekevän käyttöliittymän kaikki toiminnallisuus on kapseloitu omaan tietokantaansa ja se on kytketty datan sisältävään pääkantaan synonyymien kautta. Synonyymit ovat eräänlaisia tietokantaosoittimia, joiden avulla voidaan osoittaa toisen tietokannan tauluihin, funktioihin tai proseduureihin [1]. Tällöin järjestelmän toimituksia voidaan räätälöidä paketoimalla niihin vain tarvittavat toiminnallisuudet sisältävät tietokannat.

2.3 Tapa tehdä toimituksia

Toimitusprojektin kehitystyö alkaa projektipäällikön ja kehitystiimin yhteisestä arviosta siitä mikä ympäristö olisi samankaltainen nyt toimitettavan kanssa. Nämä samankaltaiset tietokannat valitaan pohjaksi uuteen toimitukseen. Niistä otetaan kopiot uusille nimille uusimmalla kehityspalvelimen tietokantainstanssilla, jonka jälkeen kantojen synonyymit ja käyttäjät konfiguroidaan uutta käyttöä varten.



Kuva 2. *Olemassa olevia ympäristöjä kopioimalla muodostuva puurakenne.*

Käytännössä aikaisempia toimituksia varten kehitettyjä tietokantoja kopioimalla muodostuu kuvassa 2 kuvattu puurakenne, jossa jokainen solmu on uusi toimitus. Haarautumisia ei kuitenkaan dokumentoida minnekään. Kuvassa aika etenee oikealle. Haarautumisia on tehty sekä samaan että eri aikaan.

Tämän jälkeen projektipäällikkö käy läpi toimituksen sisällön erityisesti verrattuna pohjaksi otettuun ympäristöön ja määrittelee kehitystiimille tehtävät. Toimintoja ei määritellä niinkään kuvaten kokonaistoimintaa vaan muutoksina pohjakannassa jo toteutettuihin toimintoihin. Toimintoja myös tuodaan muista kannoista. Tehtävät kirjoitetaan Scrum-kehitysmallin mukaisina kehitystehtävinä (story) tuotteen kehitysjonoon (backlog), mistä ne toteutetaan kehitystiimin sprinteissä. Vaikka nämä määritykset sinänsä jäävät talteen, ei niitä saada millään tavalla yhdistettyä kopioimalla tehtyihin haarautumisiin.

Järjestelmän paneeli-PC:lle asennettava osio (frontend, backend ja HTTP-palvelin) on paketoitu yhtenäiseksi paketiksi, joka on suoraviivainen asentaa haluttuun terminaaliin. Laiteajurit LabVIEW-kirjastoineen asennetaan paikallispalvelimelle. Nämä paketit on versiohallinnoitu SVN:n tyypillisten käytäntöjen mukaisesti eikä niiden hallinnassa ole

yleensä ongelmia [4]. Paketteihin ei kuitenkaan tarvitse tavanomaisessa toimitusprojektissa tehdä kehitystyötä.

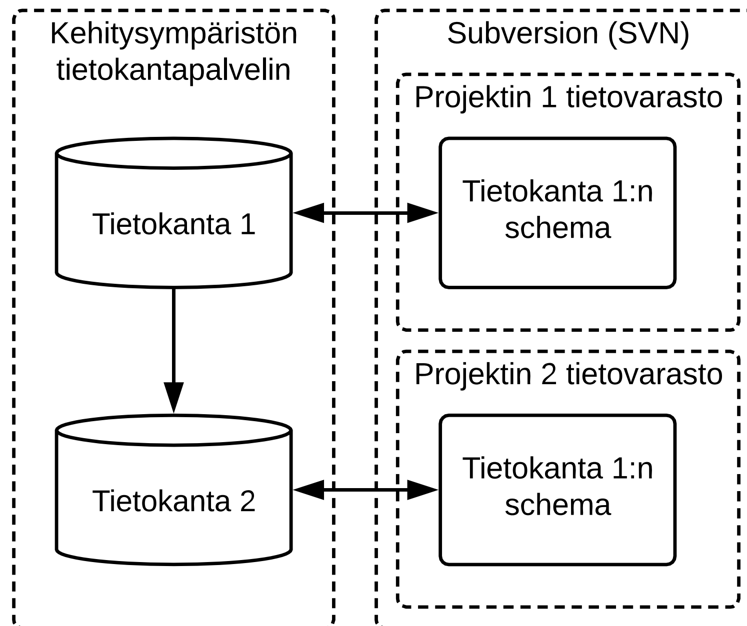
Kehittäjillä on omilla koneillaan asennettuna tuotantoympäristöä vastaava ympäristö. He kytkevät tämän ympäristön nyt luotuun uuteen kantaan ja alkavat toteuttaa tehtävillä määriteltyjä muutoksia. Kaikki kehittäjät tekevät kehitystyötään samaan kantaan. Toimitettavat laitteet asennetaan testejä varten tuotantotiloihin. Ajurit niitä varten asennetaan tuotantopalvelinta vastaavaan virtuaalipalvelimeen kehitysympäristössä.

Käyttöönottotilanteessa kehitysympäristöä vastaava ympäristö pystytetään tuotantoympäristöön ja kehitystyö viedään sinne kopioimalla kehityskanta tuotantoympäristöön.

2.4 Versiohallinnan rooli

Koko tuotteen versiohallintajärjestelmänä käytetään Apache Subversionia (SVN). Versiohallintajärjestelmän valinta ja käyttö edeltävät tässä diplomityössä käsiteltävän järjestelmän kehitystä. Uudemmat versiohallintajärjestelmät on todettu sinänsä jonkin verran paremmiksi. Etujen ei kuitenkaan olla nähty olevan niin merkittäviä, että migraatio haluttaisiin suorittaa.

Jokaiselle asiakkaan ympäristölle on luotu oma tietovarastonsa (repository, repositorio). Sitä käytetään tietokannan tietokantakaavion (database schema) versiohallinnan lisäksi muiden projektin dokumenttien kuten sähkökuvien versiohallinnointiin. Tietokantojen tietokantakaavion versiohallinnointi on haluttu keskittää samaan paikkaan muun materiaalin kanssa.



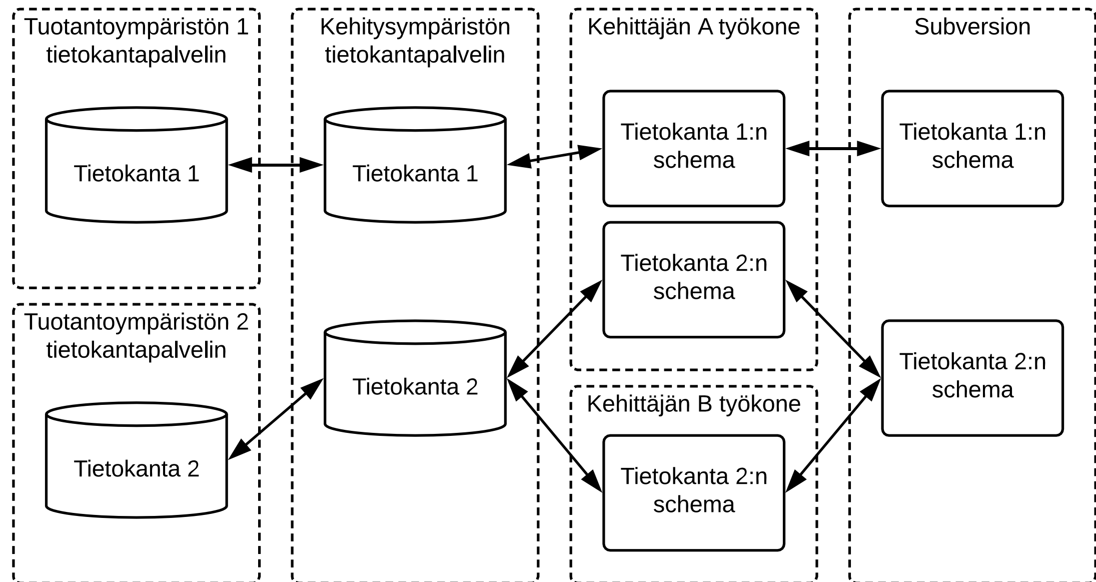
Kuva 3. *Versiohallinnan rakenne tietovarastotasolla.*

Versiohallintaa käytetään lähinnä varmuuskopiona kuvassa 3 kuvatulla tavalla. Ei ole mitään pakottavaa syytä miksei versiohallintaa voisi käyttää tavanomaiseen tapaan kuten ohjelmistokoodin muutosten tarkasteluun tai kehityshistorian tallentamiseen. Käytännössä kuitenkin versionhallintaan vientejä (commit) tehdään liian harvoin, jotta versiohallinnan kaikki hyödyt saataisiin käyttöön. Kehitystyö tehdään kehitysympäristön tietokantapalvelimella, josta myös tuotantoympäristöihin viennit tehdään. Versiohallintaa ei tarvita eri kehittäjien työn integrointiin, koska kehitystietokannat ovat yhteisiä. Tällöin pakottavaa tarvetta viedä ohjelmistokoodin muutoksia versiohallintaan ei tule. Jokainen versiohallintaan vienti on kehittäjälle ylimääräinen työvaihe. Tätä ei helpota se, että tietokantakaavion generointi versiohallinnoitaviksi komentokieleksi (script) vaatii erillisen työkalun käytön.

Koko järjestelmän kehityksen tilaa ei edes pyritä versiohallinnoimaan. Versiohistoria on asiakkaan järjestelmäkohtainen yksittäisen toimitetun järjestelmän tasolla.

2.5 Tietokantapohjaisuus ja sen erityispiirteet

Kaikki muokattava toiminnallisuus kehitetään kehitysympäristön tietokantapalvelimen instansseilla SQL-syntaksilla proseduureihin, funktioihin ja liipaisimiin (trigger). Tyypillisesti ohjelmistokehitys tehdään kehittäjän omalla koneella olevia tiedostoja muokkaamalla [4]. Kaikki versiohallinta perustuu näiden tiedostojen siirtoon ja vertailuun suhteessa muiden kehittäjien tekemään kehitykseen [11]. Tietokantakeskeisyys asettaakin siis useita lisähaasteita versiohallinnan käyttöön ja yleisesti kehittäjien työskentelyyn.



Kuva 4. *Versiohallinnan käyttö alkutilanteessa.*

Tietokannoista voi ottaa varmuuskopioita, jotka sisältävät kaiken tietokantaan tallennetun datan. Monien järjestelmän tietokantojen koko levyllä on gigatavuluokassa eikä näin suurien tiedostojen versiohallinnointi ole helppoa tai tarkoituksenmukaista. Suurin osa datasta ei ole myöskään versiohistoriallisesti kiinnostavaa. Versiohallinta ei käytännössä ole mahdollista, mikäli tietokannan ohjelmistokoodillisia osuuksia ei saada generoitua kehittäjän levylle komentokielisinä tiedostoina.

Komentokielisten tiedostojen generointiin käytetään erillistä *Redgate Softwaren SQL Source Control* -ohjelmistoa [12]. Ohjelmisto yhdistetään kehittäjän koneella olevaan hakemistoon, johon puolestaan on otettu checkout SVN:n tietovarastosta. Tämän jälkeen muutokset voidaan viedä SVN:n tietovarastoon. Tämä ketju on kuvattu kuvassa 4. Eri kehittäjien kehitystyön integrointia ei vaadita tässä vaiheessa, koska kehitystyössä käytettävä tietokanta on kaikille sama. Työkalu mahdollistaa tietokantakaavion versiohallinnoinnin, mutta on silti ylimääräinen vaihe verrattuna tyypilliseen tapaan käyttää versiohallintaa. Se on myös erittäin kallis. [12]

Samaan tietokantaan kehittäminen itsessään aiheuttaa ongelmia. Tietokantakomponentteja muokattaessa niitä itsessään ei varsinaisesti avata vaan työkalu (Microsoft SQL Management Studio) generoi ALTER-komennon. ALTER-komentoja generoidaan kaikille objekteille kuten proseduureille ja funktioille eikä pelkästään tauluille. Mikäli kaksi kehittäjää muokkaisi samaa elementtiä yhtä aikaa, viimeisin ajo kirjoittaa yli aikaisemmat. Tyypillisesti yhteen tietokantaan voikin kehitystä tehdä vain yksi kehittäjä kerrallaan.

2.6 Miten tilanteeseen on päädytty?

Ongelmien juurisyyksi voidaan jäljittää resurssipula. Järjestelmä on alun perin ollut tarve kehittää nopeasti. Se kehitettiin akuutista tarpeesta korvata vanhentunut järjestelmä. Liiketoiminnan tuloksellisuuden paine kuitenkin ajaa samalla myymään liikevaihtoa tuottavia toimitusprojekteja, jotka ovat vieneet kehityskapasiteettia. Samaan aikaan ei olla lähdetty kilpailemaan työvoimasta korkeilla palkoilla. Tästä on seurannut ensinnäkin se, että kehitystiimi on ollut keskimäärin melko kokematonta ja toisaalta se, että työvoimaa ei ole aina ollut saatavilla tarpeen mukaan.

Järjestelmästä on päätetty tehdä tietokantapainoitteinen monen syyn vuoksi. Myös muut tuotteen osat ovat tietokantapainoitteisia. Kehittäjillä on ollut hyvä SQL-osaaminen ja sekä muiden osien että vanhan järjestelmän ohjelmistokoodia on pystytty käyttämään uudelleen. Järjestelmää edeltävät ajurit ja siirto-ohjelmat ovat saaneet parametrinsä tietokannasta eikä niitä ole haluttu kehittää samalla uudestaan.

Tietokantapainoitteisuus on kuitenkin heti alusta alkaen vaikeuttanut järjestelmän versiohallinnointia. Työkalu (SQL Source Control), joka ylipäätään mahdollistaa sen, on kallis. Kalliin investoinnin toteuttamisessa on vienyt aikaa selvityksineen ja liiketoimintapäätöksineen eikä työkalua ole ollut käytössä heti alusta alkaen. SVN:n käyttö sinänsä oli aktiivista jo ennen tämän järjestelmän kehitystä. Mikäli tietokantaa olisi käytetty tavanomaisempaan tapaan välttäen ohjelmistologiikan tallennusta sinne, olisi versiohallinta ollut mukana alusta alkaen.

Kokenut kehitystiimi olisi ehkä osannut haastaa tehdyt ratkaisut ja osannut perustellusti kertoa mitä ongelmia puutteellisesta versio- ja julkaisunhallinnasta seuraa.

Alun perin tilanne on ollut kehitystiimin hallinnassa. Järjestelmä on ollut alkutilanteessa yksinkertainen ja kehittäjien tuoreessa muistissa sekä versiorakenteen että tekniikan osalta. Toimituksia ei ole tehty montaa eikä pieniä muokkauksia, korjauksia tai viilauksia ole kertynyt suurta määrää. Ajan kuluessa ominaisuuksia on kuitenkin kehitetty lisää ja järjestelmän monimutkaisuuden voidaankin katsoa kasvavan jatkuvasti.

Uudet kopiot tietokannoista uusia toimitusprojekteja varten ovat hämärtäneet kehittäjien muistikuvaa siitä mitä kautta nykytilanteeseen on pääty. Viimeinen niitti tälle ovat olleet henkilöstövaihdokset. Nykytilassa kukaan ei enää muista mitä kautta mikäkin toimituksen tietokanta on periytetty.

Kaikki mikä on jätetty tekemättä alussa oikein ja jonka korjaaminen vaatii myöhemmin ylimääräistä työtä, on kertynyttä teknistä velkaa [13]. Tekninen velka tulee myöhemmin maksettavaksi tavalla tai toisella – korkojen kera. Joko virheiden korjaamiseen käytettävä aika on aikaisempaa suurempi tai järjestelmän kanssa toimiminen on ylipäätään vaikeampaa, aikaa vievämpää ja siis kalliimpaa.

3. ONGELMAT LÄHTÖTILANTEESSA

Tässä luvussa käydään läpi lähtötilanteen aiheuttamat ongelmat sekä ohjelmistoteknisestä, projektinhallinnallisesta että liiketoimintanäkökulmasta.

3.1 Ongelmat ohjelmistokehittäjälle

Ohjelmistokehittäjän näkökulmasta perusongelma on se, että tuote on erittäin huonosti ylläpidettävä. Periaatteessa sen toiminnallisuus ei ole monimutkaista, mutta sen huono taso tekee kehittämisestä suhteettoman vaikeata. Kaikki tämä näkyy huonompilaatuisena tuotteena, suurempana työkuormana ja epämieluisampana työnä.

Tuotteen kattava dokumentoiminen on hyvin hankalaa. Ei sinänsä ole epätavanomaista, että tuotteesta on toimitettu eri versioita ja on siis tarve ylläpitää myös eri versioita dokumentaatiosta. Dokumentaation tulisi päivittyä tuotteen version mukaan. Kun päälinjaa ei ole, haarautuisi dokumentaatio täydellisestikin ylläpidettynä kuten järjestelmän eri versiot. Tällainen dokumentaatio olisi niin monimutkainen, että monimutkaisuus itsessään tekee dokumentaation hyödyntämisen mahdottomaksi. Järjestelmää ei olekaan kattavasti dokumentoitu. Tiettyyn toiminnallisuuteen palaaminen täytyy yleensä aloittaa käymällä sen ohjelmistokoodia läpi.

Hyvä ohjelmistokehittäjä pyrkii lähtökohtaisesti tuottamaan laadukasta ja helposti ylläpidettävää ohjelmistokoodia. Kuitenkin tapa kehittää tätä tuotetta aiheuttaa koodikurin murenemista. Useimpien kehitystehtävien kertakäyttöisyydestä johtuen kokonaisrakennetta ei pysähdytä arvioimaan kokonaisuutena, aikaa ei käytetä ohjelmistokoodin siistimiseen ja useimmat muutokset ajatellaan väliaikaisina. Tyypillisesti poistettava ohjelmistokoodi kommentoidaan pois ja jätetään paikalleen – saatetaanhan sitä tarvita myöhemmin. Vuosien jälkeen ohjelmistokoodi onkin täynnä ilman selitystä poiskommentoituja koodilohkoja, pieniä muokkauksia ja spagettikoodia.

Virheiden korjaamiseen ei myöskään motivoi se, että ei ole mitään takuuta korjauksen periytymisestä tietokantojen seuraaviin versioihin. Monesti samoja virheitä korjataan monta kertaa ja välillä korjaukset jäävät kokonaan periytymättä uusiin versioihin. Toisaalta voi hyvin olla, että vahingossa aikaan saadut bugit periytyvät.

Yksi tapa ylläpitää selkeää ohjelmistokoodia on *itsedokumentoituvuus*, jolloin ohjelmistokoodista pyritään tekemään niin selkeää rakenteeltaan ja muuttujanimiltään ettei kommentteja tarvita [14]. SQL komentokielenä on kehitystiimissä todettu syntaksiltaan huonosti itsedokumentoituvaksi verrattuna useimpiin tyypillisiin ohjelmistokieliin.

Yleisesti kehitystiimillä on ollut halu siirtyä automattisten yksikkötestien avulla *testivetoiseen kehitystapaan* (test-driven development, TDD) ja edelleen *jatkuvan integroinnin* (continuous integration, CI) työkalujen käyttöön [15]. Kuitenkin jo yksikkötestien käyttöönotto on todettu hyvin vaikeaksi. Hyviä työkaluja SQL Serveriin ei ole ollut saatavilla ja ylipäätään testien kirjoittaminen vanhaan ohjelmistokoodiin on todettu olevan mahdottoman suuri työ.

Yhteisen kehitystietokannan käyttö aiheuttaa sen, että käytännössä yksi kehittäjä voi työskennellä yhden ominaisuuden parissa kerrallaan. Tämäkään ei täysin estä muiden tekemien muutosten ylikirjoittamista. Microsoftin tietokantatyökalulla on ikävä tapa avata lukuisia ALTER-komentoja kehitettävälle koodille ja antaa ajaa niitä tietokantaan varoittamatta mistään. Työkalu myös pitää aikaisemmin avatut ALTER-komennot auki eri ikkunoissa ellei niitä erikseen sulje.

Järjestelmän versiohallinnollinen ja tekninen rakenne on omiaan kerryttämään suuria määriä teknistä velkaa. Tekninen velka koostuu niistä kaikista versiohallinnan, dokumentoinnin ja ohjelmistokoodin oikaisuista [13]. Ohjelmistokehittäjän näkökulmasta kehitystyö muuttuu jatkuvasti vaikeammaksi. Legacy-järjestelmien ylläpito jo lähtökohtaisesti ei ole ohjelmistokehittäjien lempitehtäviä eikä heille ole motivoivaa korjata muiden vuosia sitten jälkeen jättämiä virheitä. Samalla ajan löytäminen uudelle mielekkäämmälle kehitystyölle muodostuu koko ajan haastavammaksi.

3.2 Ongelmat projektipäällikölle

Projektin käynnistys alkaa ohjelmistokehitysnäkökulmasta yleensä valistuneella arviolla siitä mikä ennestään toimitettu ympäristö olisi mahdollisimman samankaltainen nyt toimitettavan ympäristön kanssa. Mikään toimitus ei kuitenkaan vastaa täysin toista ja räätälöintiä tarvitaan aina. Kenelläkään ei kokemuksesta riippumatta ole koskaan täydellistä käsitystä siitä, missä tilassa valittu pohjajärjestelmä on.

Tyypillisesti arvion on tehnyt projektipäällikkö konsultoiden kehitystiimiä ja myyntiä. Arvion taso riippuu sen tekevien henkilöiden muistista ja kokemuksesta tuotteen kehityksen parissa. Myös sillä kertaa toimitettavan järjestelmän samankaltaisuudella aikaisemmin toimitettuihin on merkitystä. Kokonaan uusien ominaisuuksien arvioiminen on vaikeampaa, koska niitä ei ole vielä kertaakaan toteutettu. Kovin syvälle tässä arvioissa ei myöskään voida mennä. Esimerkiksi kunkin pohjatoimituksen ohjelmistokoodin tai syvemmän teknisen toteutuksen tasoa on hankala arvioida tällä tasolla.

Kopioita otettaessa myös pyrittiin suosimaan mahdollisimman uusia ympäristöjä. Tällöin saadaan yleensä uusimmat kehitetyt ominaisuuden periytettyä. Uudempi ympäristö ei kuitenkaan takaa parempaa laatua vaan päinvastoin, koska voimassa on ollut ohjelmistokoodin laatua huonontava noidankehä.

Projektinhallinta ylipäätään on sitä haastavampaa mitä epäselvempi kuva järjestelmästä on. Toimitusprojekteissa on tyypillistä, että törmätään ongelmiin, joita ei nähty ennalta. Lisäksi projektipäälliköillä on useita toimitusprojekteja vastuullaan samaan aikaan. Pohjajärjestelmät niille voivat olla täysin erilaiset. Monesti yksinkertaiseenkin kysymykseen vastaaminen alkaa selvityksellä siitä mikä tilanne sattui olemaan juuri tässä pohjajärjestelmässä. Näitä vastauksia projektipäällikkö joutuu selvittämään monen eri sidosryhmän tarpeisiin: asiakkaan kysymyksiin, kehitystiimin kehitysjonoon tuotettavien tehtävien taustaselvityksiin tai myynnin esiselvityksiä varten.

3.3 Ongelmat liiketoiminnalle

Tässä diplomityössä käsiteltävä järjestelmä on periaatteessa toiminnaltaan melko yksinkertainen. Loppuasiakkaalle toimitettava versio on vain osittain räätälöity. Kerätty data ei ole erityisen monimutkaista, eikä järjestelmän tässä osassa tehdä sillä monimutkaista laskentaa.

Suurin toimitusprojektin kustannus on kehitystyöhön kuluva aika. Alkuaikoina tehdyt oikaisut ovat aidosti nopeuttaneet toimituksia, eikä niistä aiheutunut tekninen velka ole vielä aiheuttanut merkittäviä ongelmia. Huono dokumentointi ei ollut alussa ongelma kehittäjien muistaessa asiat, mutta ajan kuluessa ja kehittäjien poistuessa tiimistä se on muodostunut ongelmaksi. Hitaasti kehityksen vaikeutuessa on myös kasvanut siihen tarvittava työmäärä ja siis kehityksen hinta. Samaa työtä myös joudutaan tekemään moneen kertaan pelkästään sen vuoksi, että jo tehty työ ei periydy eteenpäin.

Tuotetta myydään pääasiassa urakkamyyntinä, jonka kannattavuuden arvioinnissa työmäärän arviointi on tärkeässä osassa. Tämä arvio tulee olla tehtynä ennen tarjouksen tekemistä tarjottavan hinnan laskemiseksi. Yksinkertaisimmillaan arvio on yhden henkilön tekemä - joko projektipäällikön tai myyjän. Alussa näiden arvioiden taso on ollut kohtuullinen järjestelmän ollessa yksinkertainen. Arvioiden taso on kuitenkin huonontunut samaa tahtia järjestelmän monimutkaistuessa. Paras työmääräarvio saadaan, kun projektipäällikkö on saanut tuotettua kehitystehtävät kehitysjonoon ja kehitystiimi on päässyt kollektiivisesti arvioimaan ne. Tätä tietoa ei kuitenkaan ole vielä tarjousvaiheessa käytettävissä.

Tällöin ei myöskään saada hyvää arviota myytävän työn kuormittavuudesta suhteessa kehitystiimien kapasiteettiin. Toimitusaikaikkuna kuitenkin luvataan tarjouksessa. Arviot epäonnistuvat molempiin suuntiin, mutta aina välillä tulee tilanteita missä aliarvioituja toimitusprojekteja osuu useita päällekkäin, eikä kapasiteettiä niiden tekemiseen olekaan. Tämä aiheuttaa myöhästelyä tai laadun laskemista.

Liiketoiminnalla on paine olla tuottavaa jatkuvasti eikä toimitusprojektien tekemistä voi lopettaa esimerkiksi uuden järjestelmän kehittämiseksi. Toisaalta nykytilanteessa järjestelmän jatkuva monimutkaistuvuus syö näiden toimitusprojektien katteita ja varaa aikaa

kehitystiimin rajallisesta työajasta, josta aika myös uuden järjestelmän kehittämiseksi otettaisiin. Uusien kehittäjien palkkaaminen toisi kehitysaikaa, mutta toisaalta maksaa eikä sopivia kehittäjiä ole aina tarjolla. Liiketoiminnassa joudutaankin tasapainottelemaan monen eri sidosryhmän toiveiden välillä.

Taaksepäin katsottuna kannattavinta olisi ollut myös liiketoiminnan kannalta, mikäli teknistä velkaa ei olisi alun perinkään kerätty. Alussa ja toimitusprojektien aikana tehtyjen oikaisujen välttäminen olisi luultavasti maksanut itsensä myöhemmin takaisin kehitysaikana ja siis myös rahana. Todellisuudessa järjestelmän kehitysaikaan on tietenkin tasapainoiltu samojen paineiden välillä ja nopea kehityssykli on ollut houkuttava tai melkein pakollinen.

4. KIRJALLISUUSKATSAUS JA MUIDEN RATKAISUT

Tässä diplomityössä käsiteltävä järjestelmä ei varmasti ole ensimmäinen, joka on ajautunut samoihin ongelmiin [16]. Tässä luvussa etsitään ohjelmistotuotannon teoreettisia vastauksia ja tutustutaan muiden vastaavaan tilanteeseen päätyneiden ratkaisuihin. Lisäksi hahmotellaan yksi konkreettinen esitys ideaalisesta ratkaisusta, mikäli resurssit ja aika eivät olisi olleet ongelma järjestelmän kehityksen aikaan.

Ensiksi tutustutaan joihinkin saatavilla oleviin tieteellisiin artikkeleihin. Erityisesti paneudutaan IEEE Software -lehdessä vuonna 2007 julkaisuun artikkeliin, jossa käsitellään hyvin lähellä tässä työssä kuvatun järjestelmän kaltaista tapausta. Kyseessä on myös tietokantapainotteinen järjestelmä, jonka versiohallintaan käytetään Subversionia [16]. Tämän jälkeen käydään läpi Redgate Software -yhtiön kirjassaan kuvaama käyttötapaus ratkaisuineen.

4.1 Ohjelmistotuotannon teoreettinen näkökulma

Käsiteltävän järjestelmän versiohallinnan historiallinen muodostuminen on ollut tyypillistä: alun perin kevyistä versiohallinnan käytännöistä ja kehitysympäristöistä on kasvettu ajan kuluessa ulos ja niiden riittämättömyys on todettu vähitellen [16,17]. Pelkkä tietovaraston asentaminen ja käyttöönotto sekä työkalujen koulutus eivät riitä ongelmien ratkaisemiseksi kuten tässä diplomityössä käsitellyssä järjestelmässä oli tehty ennen toimenpiteitä [16]. Myös kirjallisuudessa todetaan tietokantojen versiohallinnoinnin olevan luontaisesti vaikeaa verrattuna tavanomaiseen lähdekoodin versiohallinnointiin [16].

Tietokannan tietokantakaavion (schema) muokkaaminen tietokantaan liitetyn järjestelmän kehityksen yhteydessä on yleistä [18]. Ongelmaksi muodostuu myös kirjallisuudessa tietokantakaavion ja tietokantojen sisältämän datan versiohallinta lähdekoodien yhteydessä [16,19].

Vaikka tietokantojen versiohallintaa käsittelevää ohjelmistotieteellistä tutkimustietoa on olemassa, voidaan yleisesti ottaen todeta, että aihetta on tutkittu vähän verrattuna versiohallinnasta yleisesti saatavilla olevaan tutkimustietoon. Mikään käsitelty tutkimus ei ollut onnistunut tekemään tietokantojen versiohallinnasta yhtä helppoa kuin muun lähdekoodin versiohallinta on. Aihe voi olla edelleen ongelmallinen alalla yleisesti, tai yksinkertaisesti tietokantojen versiohallinnan ongelma yksinkertaisesti ohitetaan erilaisilla teknisillä valinnoilla.

4.1.1 Tyypilliset puutteet ja ongelmat

Monesti tietokantoihin eri kehittäjien toimesta tehtyjä muutoksia on helppo kirjoittaa yli toisten kehittäjien toimesta [16,20]. Osin tietokantojen versiohallinnan vaikeudesta johtuen on monesti helppo päätyä käyttämään yhteisiä kehitystietokantoja, joihin yhtä kehitystä varten tehdyt muutokset rikkovat muiden kantaa käyttävien järjestelmien toimintoja [20]. Myös tietokannan versiohallinnan tarkoituksen tulee olla mahdollistaa eri kehittäjien työn integroiminen ja muu tietokannan palauttaminen tiettyyn tilaan versiohallinnasta [20].

Tietokannan tietokantakaaviolla tulee olla yksi oikean version määräävä sijainti. Ideaalisesti tämän sijainnin tulisi olla versiohallinnan tietovarasto. Mikäli edellä mainittu jaetun tietokannan ongelma vältetään vain sillä, että eri kehittäjät käyttävät omia kehityskantojaan, jää epäselvyys kenen kehittäjän tietokanta on uusimmassa tilassa. Entä mistä tietokannasta tehdään julkaisu? Miten tietokantojen muutokset integroidaan toisiinsa? [20]

Tyypillisesti normaalin lähdekoodin versiohallinnassa versiohallintaan vienneistä saadaan taltioitua aikaleima, sen tehneen kehittäjän nimi ja kommentti [4]. Tietokantojen osalta muutosten seuraaminen ja jäljittäminen tiettyihin kehittäjiin tai aikaan on rajattua [16]. Myös muutosten peruminen on hankalaa tai käyttäjäriippuvaista [16]. Varmuuskopioita saatetaan ottaa esimerkiksi päivityksen yhteydessä saaden jonkinlainen versiohistoria, mutta niiden käyttö unohdetaan välittömän käyttötarkoituksen, kuten muutosten integroimisen tuotantojärjestelmän ohjelmistokoodiin mentyä ohi [16].

Muutosten, kuten päivitysten turvallinen käyttöönotto on hankalaa ilman järjestelmän toiminnan häiritsemistä [16]. Tietokantakaavion versiohallinnoinnin lisäksi data aiheuttaa omat ongelmansa. Strazdins listaa tietokannan datalle tässä kontekstissa neljä ongelmaa: eheys, uskottavuus, jäljitettävyys ja versiohistoria [19]. Tietokantahallintajärjestelmien yhteydessä puhutaan tätä määritelmää lähellä olevasta ACID-periaatteesta: atomisuus (atomicity), eheys (consistency), eristyneisyys (isolation) ja pysyvyys (durability) [21]. Jälkimmäistä kuitenkin käytetään tietokannan transaktioiden eikä niinkään versiohallinnan kontekstissa kuten ensimmäistä. Mielenkiintoisesti kummankin yhteydessä päädytään kuitenkin tosiaan lähellä oleviin periaatteisiin.

4.1.2 Kehitysympäristön parantaminen

Ploskin et al. kirjoittamassa vuonna 2007 IEEE Software -lehdessä julkaistussa artikkelissa kuvataan erään tässä diplomityössä käsiteltävän järjestelmän kaltaisen järjestelmän saattaminen versiohallinnan piiriin [16]. Tässä luvussa käydään läpi erityisesti tätä ratkaisua edeltäviä vaatimuksia ja haasteita.

Versiohallinnan käyttöönottoa tukemaan perustettaisiin uusi kehitys- ja testausympäristö. Tämän ympäristön tulisi muistuttaa läheisesti tuotantoympäristöä laitteisto- ja ohjelmistotasolla. Tässä tapauksessa parhaaksi ratkaisuksi todettiin järjestelmäympäristön asentaminen virtuaalikoneille, jotka pyörivät tätä käyttöä varten hankitulla uudella palvelimella. [16]

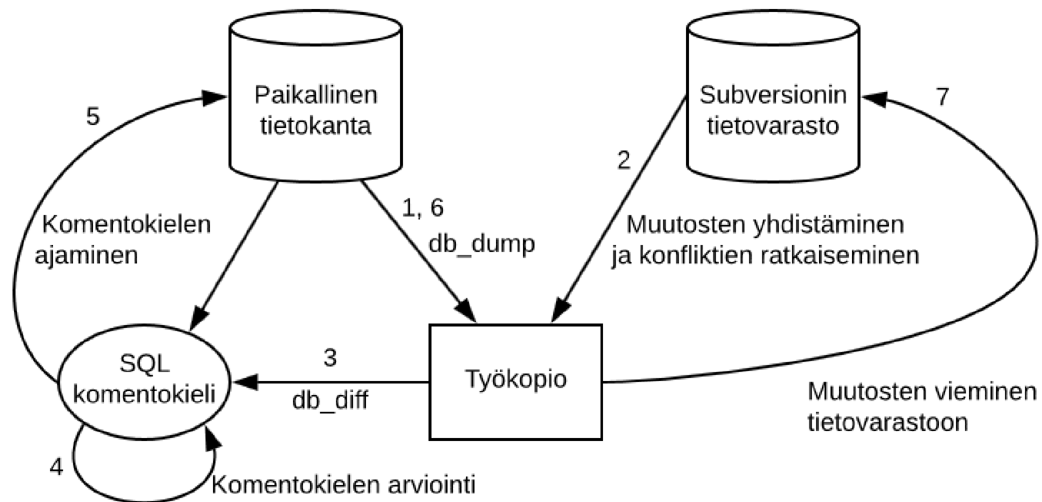
Uuden ympäristön tulisi toimia sekä kehitystyötä tukevana kehitysympäristönä, testiympäristönä, tuotantoympäristön virheenkorjauksen tukena että laskentaympäristönä aikaa kuluttaville kyselyille (eräajoille). Useista rooleista huolimatta ympäristön tuli säilyttää ketteryytensä. Siitä tuli pystyä viemään uusia ominaisuuksia ja korjauksia nopealla aikataululla tuotantoympäristöön. [16]

Monien edellä mainittujen vaatimusten todettiin olevan keskenään ristiriitaisia: käyttö kehitysympäristönä estää käyttöä testaukseen, virheenkorjaukseen ja eräajoihin, koska ympäristö saattaa olla kehityksen aikana ei-eheässä tilassa. Jotta järjestelmää voisi käyttää esimerkiksi tuotantoympäristön virheenkorjaukseen, tulisi se pitää ajan tasalla tuotantoympäristön kanssa ja tätä varten näiden järjestelmien tulisi olla rakenteellisesti samassa tilassa. [16]

Ratkaisu tähän voisi olla monistaa järjestelmää rajoittamattoman paljon eri tarpeiden mukaan, mutta tietokannan koko levyllä rajoittaa tätä. Toisaalta rajatun datajoukon käytön todetaan olevan rajaavan kehittäjien kehitystehtävien tekemistä. Tietokannan datan tulisikin siis olla riittävän kuvaavaa kaikissa tilanteissa. Jos haasteita ei ratkaista ennen uuden järjestelmän käyttöönottoa, todetaan ongelmien määrän kasvavan entisestään. [16]

4.1.3 Ratkaisuja ongelmiin

Ploski et al. esittävät ratkaisuksi työkalujen tukemaa prosessia lähdekoodin ja tietokantakaavion versiohallinnan yhdistämiseksi samaan tietovarastoon. Prosessin olisi oltava yksinkertainen ja se käyttäisi ilmaiseksi saatavilla olevaan tekniikkaa. Tällä saavutettaisiin se, että kehittäjillä ei tarvitsi olla aikaisempaa kokemusta versiohallinnasta, vähennettäisiin koulutustarvetta, mahdollistettaisiin sujuva muutos ja parannettaisiin muutosten vastaanottoa yleisesti. [16]



Kuva 5. *Ploskin et al. artikkelissa kuvattu tietokannan versiohallinnan prosessi [16].*

Tuotantojärjestelmä olisi monistettuna kehitysympäristön virtuaalikoneella. Prosessi toimisi kuvassa 5 kuvatulla tavalla:

1. Paikallisesta tietokannasta otetaan automaattisesti työkopio.
2. Tietovaraston muutokset yhdistetään työkopioon. Samalla ratkaistaan mahdolliset konfliktit.
3. Työkopiosta generoidaan automaattisesti komentokieliset tiedostot.
4. Komentokielisten tiedostojen taso arvioidaan manuaalisesti.
5. Komennot ajetaan paikalliseen tietokantaan.
6. Paikallisesta tietokannasta otetaan taas työkopio. Nyt versiohallinnassa ei pitäisi enää olla uusia yhdistämättömiä muutoksia.
7. Työkopion muutokset viedään versiohallinnan tietovarastoon. [16]

Prosessi otettaisiin käyttöön vaiheittain. Aluksi tuotantoympäristö yksinkertaisesti monistettaisiin kehitysympäristön virtuaalikoneille. Tämän jälkeen oheisjärjestelmän lähdekoodi viedään versiohallintaan ja samalla parannetaan tietokannan konfiguroitavuutta automaattisia työkaluja varten sekä poistetaan tarpeettomia riippuvuuksia kuten kovakoodattuja osoitteita. Tietokanta versiohallinnoitaisiin ensimmäisen kerran manuaalisesti ja tämän jälkeen otettaisiin käyttöön kaksi työkalua versiohallinnan käyttöön automaattisesti: db_dump ja db_diff. Näistä ensimmäinen generoi tietokantakaavion ihmisen ymmärrettäviksi ja versiohallinnoitaviksi tiedostoiksi. Jälkimmäinen vertailee näitä tiedostoja ja generoi erojen pohjalta komentokielisiä tiedostoja erojen synkronoimiseksi tietokannassa. [16]

Toisaalta Allen esittää manuaalisemman tavan, jossa tietokantaobjektit luovat komentokieliset tiedostot luodaan käsin ja versiohallinnoidaan alkutilanteen perustamiseksi [22].

Kun muutoksia halutaan viedä versiohallintaan, luodaan näistä muutoksista omia komentokielisiä muutostiedostoja [23]. Nyt tietokanta voidaan palauttaa haluttuun versioon näiden yhdistelmästä käyttämällä työkalua, joka poistaa kaikki tietokannan objektit ja ajaa komentokieliset tiedostot luoden objektit uudestaan [24]. Versiohallinnan haarojen suurta määrää tulisi välttää ja haarojen ominaisuuksien yhdistäminen toisiinsa tehtäisiin joko luomalla uusia komentokielisiä tiedostoja kohdehaaraan tai yhdistämällä kokonaisia tiedostoja, ei muokkaamalla jo versiohallinnoituja [25].

4.1.4 Muutosten riskejä ja rajoitteita

Kun versiohallinta otetaan käyttöön versiohallinnoimattomaan tietokantaan, on mahdollista aiheuttaa epäjohdonmukaisuuksia eri ympäristöjen ja versiohallinnan välille. Hallinnollinen työ voi lisääntyä ja kehittäjien aikaa kuluu uusien työkalujen ja menetelmien opiskeluun [16]. Nämä ovat ongelmia, joita on todettu myös tässä diplomityössä käsitellyn järjestelmän versiohallinnoinnissa sekä ennen toimenpiteitä että niiden jälkeen.

Muutoksille tulee olla kehittäjien ja liiketoiminnasta vastaavan johdon tuki. Kehittäjien tuki saavutetaan prosessin yksinkertaisuudella ja johdon tuki sen matalalla hinnalla. Toisaalta työkalujen kehitys itse aiheuttaa merkittäviä kuluja ja on siis ristiriidassa jälkimmäisen tavoitteen kanssa. [16]

Ploskin et al. ehdottama prosessi ei tue datamigraatioita kovin hyvin vaan esimerkiksi rakenteeltaan tietovarastossa olevan poikkeavan tietokantataulun sisältöä varten tarvitaan ensiksi sen lisääminen tietokantaan manuaalisilla toimenpiteillä. Lisäksi versiohallinnointi vaatii tietokantakaavion käsittelyn tekstimuodossa levyllä ja kehitystyöhön käytettävien tietokantaversioiden lukumäärä on rajattu niiden suuren koon vuoksi. [16]

4.2 Redgaten ratkaisu

Tietokantojen hallintaohjelmistoja tuottavan Redgate Softwaren työkaluja käytetään tämän järjestelmän kehityksessä runsaasti erityisesti varmuuskopiointityyppisessä versiohallinnassa ja käyttöönotoissa [12]. Työkalujen tarjoamista mahdollisuuksista on kuitenkin käytössä vain pieni osa. Tässä luvussa tutustutaan Redgate Softwaren vuonna 2017 julkaiseman kirjan ”Database Lifecycle Management” kuvaamiin ratkaisuihin tietokantojen versiohallinnassa. Erityisesti paneudutaan lukuihin 5 ja 10: ”Database Version Control” ja ”Database Deployment and Release”.

Kirja paneutuu versiohallintaan DevOps-näkökulmasta sisältäen mm. Continuous Integration (CI) -käytänteitä. Oikeaoppinen versiohallinta on kuitenkin iso osa tätä. Tähän käyttöön ei tämän diplomityön puitteissa mennä. Samalla on kuitenkin hyvä huomioida, että DevOps-käytänteiden käyttöönotolle on olemassa tahtotila myös tämän järjestelmän osalta ja jolle suureksi esteeksi on todettu puutteet tietokantojen hallinnoinnissa.

4.2.1 Tietokantojen versiohallinta

Versiohallinnan käyttö todetaan pakolliseksi kirjan kuvaaman Database Management Lifecycle (DLM) -menetelmän käytössä. DLM kuvataan kirjassa laajempana kokonaisuutena kuin DevOps: DevOps keskittyy kehitys- ja tuotantoympäristöjen sujuvaan yhteystoimintaa ja niiden välisiin nopeisiin käyttöönottoihin, kun taas DLM kuvataan laajempana teoreettisena viitekehyksenä. [26]

Versiohallintaan tallennetaan tietokantakaavion kuvaamia asioita kuten taulujen, tallennettujen proseduurien, funktioiden ja liipaisimien komentokieliset kuvaukset sekä staattinen data, jota tässä diplomityössä kuvattu ylläpitodata olisi. Näiden lisäksi tulee huomioida tietokannan toiminnan kannalta merkitykselliset riippuvuudet. Niitä ovat mm. tietokannan, palvelimen ja tietoverkon asetukset, käyttäjäroolit ja -oikeudet, automaattisesti ajettavat toimenpiteet, tietokannan luomiseen tarvittavat komentokielitiedostot, rajapintakuvaukset, dokumentaatio ja tietokantaan liittyvät tiedostot. [26]

Versiohallintajärjestelmän valinnassa tärkeää on se, että eri kehittäjät pystyvät seuraamaan versiohallinnassa olevien tiedostojen muutoksia helposti. Erityisesti käytettävyyden helppouteen tulee kiinnittää huomiota. Tärkeää on erotella matalan tason versiohallintajärjestelmät kuten Git tai Subversion kattavampia versiohallinta-alustoja tarjoavista tuottajista kuten Github ja harkita versiohallintaa alemmalla tasolla. Kirja toteaa Git:n yleensä parhaaksi valinnaksi työnkulkumallinsa vuoksi. Tässä työssä käsiteltävässä järjestelmässä käytettävän Subversionin todetaan olevan hyvä valinta kooltaan pienille tietovarastoille tai nopean verkkoyhteyden piirissä toimivalle järjestelmälle. [26]

Versiohallinta-alustat tarjoavat versiohallintajärjestelmän toiminnallisuudet ja lisäksi paljon käyttäjäkokemusta parantavia toiminnollisuuksia. Lisäksi tulee kiinnittää huomiota siihen, kuinka helppoja niitä on käyttää mikäli käytöstä tulee huolehtia itse. Kirja argumentoi, että SaaS-tyyppiset palvelut ovat tietoturvaltaan itse ylläpidettyjä järjestelmiä turvallisempia, koska tietoturvallisuuden hallinta useimmissa itseylläpidetyissä järjestelmissä on heikkoa. [26]

Versiohallinnan tulisi olla määräävä totuus versiohallinnoitavan järjestelmän tilasta myös tietokantojen kohdalla. Versiohallintajärjestelmän tulisi sisältää kaikki mitä tarvitaan tietokannan uudelleenrakentamiseen halutussa versiossa. Useita hallintotoimenpiteitä tarvitaan, jotta tietokannan tila voidaan selvittää tietyssä versiossa. Lisäksi tulee olla kyky selvittää mitä muutoksia on tehty, kenen toimesta ja miksi. [26]

Kirja käy läpi useita oleellisia käytäntöjä tietokantojen versionhallinnan toteuttamisessa. Tallaisia ovat versiohallinnan integroiminen avoimien asioiden seurantajärjestelmään (Jira, Trello jne.), yksinkertaisen tietokantakehitystyökalua vastaavan hakemistorakenteen käyttöstandardin käyttäminen, usein toistuvat pienet versiohallintaan viennit versio-

hallinnassa olevan tietokannan eheyden sekä testaamisen varmistamiseksi ja tyhjämerkityksen (välilyönnit, tabulaattorit jne.) käytön yhdenmukaistamisen. Tyhjämerkityksen uudelleenformatoinnit tulisi lisäksi pitää erillään ohjelmistokoodillisesti merkityksellisistä versiohallintaan vienneistä. Tämän tarkoitus on maksimoida merkityksellisten vientien muutosten näkyvyys. [26]

Lisäksi peräänkuulutetaan suunnitelmaa koordinoida tietokannan ulkoisen ohjelmiston ja tietokannan muutoksia. Tietokannalla voi olla useita eritasoisia liityntöjä muuhun ohjelmistoon. Yhdessä ääripäässä tietokanta on täysin irrallaan muusta järjestelmästä, kun taas toisessa tietokanta on tasavertainen komponentti muun ohjelmiston kanssa. Erityisesti jälkimmäisessä tapauksessa tietokanta tulisi olla versiohallinnoitu yhdessä muun ohjelmiston kanssa. Tässä diplomityössä käsiteltävä järjestelmä on hyvin vahvasti ensimmäisen tapauksen kaltainen eikä muuta ohjelmistokoodia (lähinnä kevyttä frontend-toiminnallisuutta) versiohallinnoida yhdessä tietokantojen kanssa. [26]

Kirja peräänkuuluttaa minimaalista haarojen käyttöä, koska jokainen haara lisää tarvetta vastaavassa tilassa olevan tietokannan säilömiseen versiohallinnassa. Tämän neuvon todetaan olevan ristiriidassa monien versiohallintajärjestelmän käyttötapojen kanssa. Haarojen määrä tulisi minimoida kiinnittämällä huomiota haarojen elinkaarien lyhyyteen, organisaatiosta johtuvien syiden poistamisella ja ominaisuuksien päälle kytkemisellä ominaisuuskohtaisten haarojen käytön sijaan. [26]

Lopuksi mainitaan tietokantakonfigurointien versiohallinnointi ottaen huomioon eri järjestelmien erot. Tietokannan versiohallinnan tulisi pitää sisällään myös nämä tiedot. Kuitenkin yhdestä versiohallinnassa olevasta tietokannasta tulisi pystyä tekemään käyttöönottoja erilaisiin ympäristöihin, joissa siis olisi erilaiset konfiguraatiot. Tulisi kuitenkin pyrkiä käyttämään konfigurointitiedostoja ja versiohallinnoimaan ne. Tiedostoista pitäisi karsia pois kaikki ympäristökohtaiset toiminnot ja ulkopuoliset riippuvuutta aiheuttavat ohjelmistot. Ympäristökohtaisia tietovarastoja ei tulisi käyttää. [26]

4.2.2 Tietokantojen käyttöönottoympäristöt ja toimenpiteet

Tietokantojen käyttöönotoissa voi tulla esiin ennakoimattomia puutteita, jotka aiheuttavat viiveitä. Kirjassa kuvataan prosessi, jonka tarkoituksena on saada siirrettyä tietokannan ohjelmistokoodia tuotantoympäristöön kivuttomasti, nopeasti ja testiympäristöjen kautta julkaisuympäristöihin. Kaiken tämän tulisi tapahtua mahdollisimman automaattisesti ja sulavasti halutun nopeuden saavuttamiseksi. [26]

Julkaisuja on harvoin mahdollista tehdä suoraan kehitysympäristöistä tuotantoympäristöihin vaan useimmiten tarvitaan prosessi ohjelmistokoodin toimittamiseen tuotantoympäristöön. Kirjassa keskitytään erityisesti testi- ja käyttöönottoympäristöjen pystyttämiseen, käyttöönottojen automatisointiin ja tuotantoympäristöjen suojelemiseen erityisesti automaattisia käyttöönottomenetelmiä käytettäessä. [26]

Testiympäristö (test environment) on palvelinympäristö, joka emuloi tuotantoympäristön toiminnallisuutta, muttei sen suorituskykyä. Käytännössä tietokannoista puhuttaessa palvelimelle on asennettuna sama versio esimerkiksi tietokannanhallintajärjestelmästä. Testiympäristöllä tavoitellaan kykyä ajaa käyttökelpoisia eli tuotantoympäristöä kuvaavia testejä. [26]

Ympäristön perustaminen on yleensä helppoa ja ongelmaksi muodostuukin ympäristön data. Paras tapa olisi käyttää tuotantoympäristön dataa myös testiympäristössä, mutta useimmiten se ei ole mahdollista. Syitä voi olla useita: datan salassapitovaatimukset, uusien toiminnallisuuksien vaativaa dataa ei välttämättä ole edes olemassa tuotantoympäristöissä tai datan määrä voi yksinkertaisesti olla liian suuri kopioitavaksi. Toimivaa testiympäristöä varten dataa tulee olla riittävän vähän ympäristöjen nopeaan alustamiseen, datan tulee olla toiminnallisuudeltaan tunnettua toistettavuutta varten, edustavaa, lainopillisesti riittävän salaamatonta ja rakenteeltaan tuotantoympäristöä vastaavaa. [26]

Pääasiallisia vaihtoehtoja testiympäristön dataa varten listataan kolme: puhdas varmuuskopio tuotantoympäristön datasta, tuotantoympäristön varmuuskopiosta muokattu data ja erikseen luotu data. Ensimmäisessä vaihtoehdossa kopioidaan tuotantoympäristön data testiympäristöön, toisessa tuotantoympäristön data kopioidaan testiympäristöön muokaten sitä automaattisesti niin että esimerkiksi arkaluontoinen data poistetaan ja kolmannessa luodaan keinotekoista dataa suoraan testiympäristöön. Mitä kauemmas puhtaasta tuotantoympäristön datasta mennään, sitä enemmän kasvavat mahdollisuudet tuottaa ei-kuvaavaa dataa. Lisäksi prosessin automatisointi vaikeutuu. [26]

Kirjassa erotetaan käyttöönottoympäristö (pre-production/staging environment) testiympäristöstä. Kun testiympäristö vastaa konfiguroinniltaan tuotantoympäristöä, mutta ei suorituskyvyltään, tulisi käyttöönottoympäristön vastata tuotantoympäristöä mahdollisimman identtisesti. Sitä kuvataan oveksi tuotantoympäristöön. Siinä missä testiympäristöön siirtymisiä saattaa olla useita vuorokaudessa, käyttöönottoympäristöön tehdään vain yksi onnistunut siirtyminen. [26]

Erityisongelman tietokantojen käyttöönotoissa muodostavat palvelintason objektien kuten palvelinten asennukset ja SQL-agenttien hallinta. Suositeltavaa on asettaa palvelimien asetukset kerran oikeaksi. Tämä ei välttämättä ole kaikissa tilanteissa mahdollista. Tällöin tulee käyttää komentokielisiä tiedostoja, joita ajetaan automaattisesti käsityön sijaan. Komentokieliset tiedostot voidaan viedä versiohallintaan ja ajaa erisisältöisinä eri ympäristöihin. [26]

Tuotantoympäristöjen suojeleminen on tärkeää erityisesti automaattisia käyttöönottomenetelmiä käytettäessä. Kaikista testeistä huolimatta jotain voi mennä väärin ja käyttöönottoa edeltävään tilaan tuleekin voida palata luotettavasti ja nopeasti. Tapoja suojella tuotantoympäristöjä ovat varmuuskopiot koko palvelimesta, varmuuskopiot tietokannasta, A/B-käyttöönotot, komentokieliset paluutiedostot ja kyky korjata virheitä välittömästi.

A/B-käyttöönotossa käyttöönotto suoritetaan virhekorjauksineen toiselle tuotantopalvelimelle toisen jatkaessa käytössä ja käyttöönoton jälkeen suoritetaan vaihto. Kyky korjata virheitä välittömästi (fail forward) tarkoittaa sitä, että palautuksen yrittämisen sijaan virheet korjataan heti käyttäen hyväksi nopeita ja automaattisia kehitysprosesseja. [26]

4.2.3 Käyttöönoton automatisointi

Tietokantojen käyttöönoton automatisointi liittyy tiiviisti CI-prosessin käyttöönottoon samoine ongelmineen. CI-toiminnoissa pystytetään prosessi ja ympäristöt eri kehittäjien työn jatkuvaan integroimiseen ja testaamiseen. Käyttöönotossa halutaan tehdä samat toimet tarkoituksena kehityksen vienti tuotantoympäristöön. [26]

Käyttöönoton automatisointia varten tarvitaan ympäristöt ja työkalut versiohallintaan liittymiseen, käännösten suorittamisen konfigurointiin, käännöshistorian hallintaan, liitynnät käännösten tilasta kertomiseen ja automaattinen testausympäristö [26]. Näiden työkalujen tarkempiin haasteisiin ja sisältöön ei tämän diplomityön puitteissa mennä, vaikka kirja käsitteleekin niitä paljon.

Kun tarvittavat työkalut ja ympäristöt ovat valmiina, käytännössä tietokantojen käyttöönotto suoritetaan automaattisesti ajamalla näillä työkaluilla komentokielisiä tiedostoja. Kirja käsittelee erityisesti T-SQL -komentokieltä, joka on sattumalta tässä diplomityössä käsiteltävän järjestelmän SQL-kieli. Komentokieliset tiedostot parametrisoidaan tarpeen vaatiessa tukemaan vaihtuvia osoitteita, tiedostojen sijainteja tai tietokantojen nimiä. Tiedostot kehitetään komentoriviajoa silmällä pitäen. [26]

4.3 Ideaaliratkaisu

Ottamatta kantaa järjestelmän nykyisestä versiosta pois siirtymisen kustannuksiin tai resurssitarpeisiin, voidaan hahmotella taaksepäin katsoen mikä olisi ollut ideaalinen ratkaisu järjestelmää alun perin kehittäessä. Yksiselitteisiä parhaita valintoja ei ole ja tämä toteutus on vain yksi mahdollinen toteutus nojaten siihen kokemukseen, joka diplomityön kirjoittajalle järjestelmän ongelmista ja yleisesti ohjelmistotekniikan hyvistä käytännöistä on muodostunut.

Yleisesti ottaen ideaaliratkaisussa osa tunnistetuista ongelmista pyritään ratkaisemaan uusia työkaluja hyödyntämällä ja toisaalta ohittamalla osa ongelmista kokonaan järjestelmän teknistä rakennetta muuttamalla. Jälkimmäiset ratkaisut tarkoittavat käytännössä järjestelmän toteuttamista uudelleen ja siis vaativat edellä mainitut rajoittamattomat resurssit.

4.3.1 Siirrytään pois tietokantakeskeisyydestä

Kaikkien tässä ideaaliratkaisussa hahmoteltujen seikkojen toteuttaminen olisi vaikeata tai mahdotonta tietokantakeskeisellä toteutustavalla. Samasta syystä niiden toteuttaminen järjestelmään sen nykyisessä muodossa olisi hankalaa ja ideaaliratkaisu pitäisikin siis toteuttaa kokonaan uutena järjestelmänä.

Object-relational Mapping (ORM) -toteutuksessa tietokantatoiminnallisuus on tyypillisesti piilotettu enimmäkseen käytettävän työkalun taakse ja ohjelmistokoodissa käsitellään tietoa vain olioina [27]. Tällöin ORM-järjestelmä huolehtii olion tietokannan tiedon synkronoinnista. Sen käyttö olisi täysi vastakohta nykyisen järjestelmän tavalle käyttää tietokantoja ja mahdollistaisi logiikan siirtämisen kokonaan pois tietokannasta. Ongelmaksi muodostuu kuitenkin riippuvuus tuotteen muista komponenteista, joihin tulee pystyä siirtämään tietoa myös tässä työssä käsiteltävästä järjestelmästä riippumatta sen toteutuksesta. Tätä varten tarvitaan hyvä kontrolli tietokannan tietokantakaavioon. Tätä ORM-työkalut eivät välttämättä tarjoa eikä voida luottaa siihen, että tieto on tallennettu siinä muodossa missä sen oletetaan olevan muissa järjestelmissä.

Siis vaikka ORM saattaisi olla ideaalitavoite, sen käyttöön ei välttämättä voida mennä. Kaikki järjestelmän logiikka pystytään kuitenkin siirtämään pois tietokannasta valittuun backend-toteutukseen. Siinä käytettäisiin model-view-controller (MVC) -rakennetta, jossa toimintalogiikka on kapseloitu käsittelijään (controller) [28]. Tätä rakennetta käytetään käyttäjälle näkyvästä näkymästä (view), jota puolestaan päivitetään mallista (model). Kaikki tiedon käsittely voidaan nyt kapseloida mallin sisälle. Se olisi ainoa paikka, josta suoritetaan tietokantakyselyitä, se tarjoaisi vain rajapinnat muille komponenteille ja se voitaisiinkin halutessa korvata esimerkiksi ORM-toteutuksella muiden tuotteen komponenttien kehittyessä omilla janoillaan.

Backend-toteutus voidaan toteuttaa melkein millä tahansa hyväksi todetulla viitekehyksellä (framework) ja ohjelmointikielellä. Tärkeintä on päästä tavanomaiseen tapaan kirjoittaa ohjelmistokoodia, jotta päästään eroon tietokantaan kirjoitetun ohjelmistologiikan aiheuttamista ongelmista. Ohjelmistokoodia olisi pakko versiohallinnoida tavanomaisesti jo pelkästään kehittäjien työn integroimiseksi ja toimittamiseksi eikä sen käytöstä olisi mahdollista lipsua. Se olisi myös suoraviivaista, koska ylimääräinen vaihe SQL-ohjelmistokoodin generoimiseksi komentokielisiksi tiedostoiksi ulkopuolisella työkalulla poistuisi. Ohjelmistokoodista saataisiin myös helpommin itsedokumentoituvaa aivan eri tavalla kuin monimutkaisella SQL-kielellä on mahdollista.

4.3.2 Käytetään moderneja tuotteen toimitustapoja

Suoraviivainen ja aktiivinen tapa käyttää versiohallintaa mahdollistaisi monet modernit tavat testata ja toimittaa tuotetta. Näitä ovat jatkuva toimitus (continuous delivery, CD), jossa CD-järjestelmä kääntää, integroi, testaa ja siirtää tuotantoympäristöön jokaisesta

versiohallintaan tehdystä tallennuksesta. Askelta lyhyemmälle jäädään jatkuvassa integraatiossa (continuous integration, CI), jossa järjestelmä toimitetaan vain testiympäristöön ja tuotantoympäristöön siirto vaatii ihmisen toimenpiteen ja se olisikin luultavasti parempi valinta tämän järjestelmän käyttöön, koska välttämätöntä tarvetta viimeiselle askeleen käyttöönotolle ei ole. [15]

Kummankin menetelmän käyttö vaatii automaattisesti ajettuja testejä, jotta voidaan luottaa ohjelmistokoodin toimivuuteen [29]. Erityisen oleellisia ne ovat CD-järjestelmän käytössä. Automaattitestejä on hyvin hankala kirjoittaa jälkikäteen ilman massiivista määrää työtä. Ei riitä, että kirjoitetaan pelkkä testi, koska se ei onnistu ilman jokaisen testattavan ohjelmistokoodin logiikan ymmärtämistä. Tarvitaankin siis testivetoisen kehitystavan (test-driven development, TDD) käyttöönottoa jo järjestelmän kehityksen alkuvaiheessa. Automaattitestit ajetaan jokaisen versiohallintaan tehdyn tallennuksen yhteydessä ja niiden läpimeno on vaatimuksena etenemiselle.

Samassa yhteydessä voidaan hyödyntää säiliöintiä (containerization). Säiliöinnissä ohjelmiston ohjelmistokoodi, komponentit, data ja muut asennukseen liittyvät elementit on säiliöity yhteen pakettiin [30]. Pakettia voidaan jakaa muille tai siitä voidaan tehdä asennuksia esimerkiksi tuotantoympäristöön. Säiliöinti tekee periaatteessa saman työn kuin perinteisempi virtuaalikone, mutta kevyemmin [30]. Virtuaalikoneita käytetään tämän järjestelmän kehityksessä jo runsaasti, mutta ei automaattisissa prosesseissa eikä niitä ole tarkoituksenmukaista ottaakaan laajempaan käyttöön niiden raskauden vuoksi.

Kaikkien näiden menetelmien käyttöönottoa on selvitetty tuotteen kehityksessä useaan otteeseen, mutta tietokantakeskeisyys ja työmäärän suuruus on estänyt niiden käyttöönoton. Vaikka mahdollisia luvussa 4.2 kuvattuja työkaluja on olemassa, ovat ne kalliita ja itsessään jo ennestään suurta työmäärää kasvattavia.

4.3.3 Siirrytään käyttämään Git-versiohallintajärjestelmää

Modernien kehitysmenetelmien käyttöönoton selvityksissä on toistuvasti törmätty siihen, että Subversion-versiohallintajärjestelmästä on aika alkanut ajaa ohi. CD/CI-järjestelmät toteutetaan käytännössä käyttämällä niitä varten suunniteltuja työkaluja, jotka tukevat parhaiten Git-versiohallintajärjestelmää [31,32]. Joko Subversion-tukea ei ole ollenkaan tai mikäli on, on se usein huonompi kuin Git-tuki. Myös kolmansien osapuolien tuki Subversionin käytölle tässä tarkoituksessa on pientä. Suurin osa ohjelmistokehityksestä on ongelmien ratkaisujen etsimistä internetistä ja mikäli käyttäjäkunta on pieni, ei näitä ratkaisuja ole löydettävissä.

Git-versiohallintajärjestelmä ylipäätään on modernimpi tapa tehdä sama asia. Se on avoimen lähdekoodin järjestelmä, jota voitaisiin käyttää ilmaiseksi kuten Subversioniakin, mutta tämän lisäksi olisi tarjolla monia laajalti käytettyä pilvipalveluun perustuvia palveluita kuten Github tai Bitbucket. Yhtä laajaa valinnanvaraa ei ole tarjolla Subversionille.

Git on myös tapa, jota käytetään uusien ohjelmistokehittäjien koulutuksessa ja laajalti muissa yrityksissä, ja se olisikin uusille rekrytoitaville kehittäjille tutumpi.

Versiohallintajärjestelmän vaihdon puolesta puhuu myös se, että yrityksessä vaihto on jo muuten tehty ja valmis Github-ympäristö olisi jo käytettävissä. Git on kuitenkin todettu eduiltaan jo olemassa olevien järjestelmien käytössä niin pieneksi, että migraation vaadittavaa työtä ei olla haluttu tehdä. Uuden järjestelmän kehityksen alkuvaiheessa vaihto kannattaisi kuitenkin tehdä.

4.3.4 Tehdään muutoksia frontend-toteutukseen

Järjestelmä on nyt tehty itse kehitetyllä, kevyellä frontend-viitekehyksellä Web-tekniikoilla toteutettuna käyttöliittymänä siitä huolimatta, että se pyörii normaalia Windows-käyttöjärjestelmää käyttävillä tietokoneilla. Tarvetta käyttää Web-tekniikoita ei siis ole, vaikka tapa onkin aika yleinen. Toisaalta estettä käyttää Web-tekniikoita jatkossakaan ei ole ja suurimmasta osasta käyttöliittymän ongelmista päästäisiinkin eroon jo tietokantakeskeisyydestä poistumalla. Olemassa olevassa järjestelmässä jokaisesta toiminnosta tehdään tietokantakutsu ohjelmistologiikan ajamiseksi mikä tekee käyttöliittymästä tarpeettoman kankean ja latailevan.

Ideaalijärjestelmän käyttöliittymä kehitettäisiinkin siis joko käyttäen mitä tahansa modernia Web-kehityksessä käytettävää viitekehystä tai kokonaan työpöytäohjelmana. Myöskään tässä ei oleellista ole minkään tietyn viitekehyksen valinta vaan ongelmat korjaavia hyviä vaihtoehtoja on monia. Tärkeintä olisi parantaa liittymän käyttäjän käyttäjäkokemusta nopeuttamalla ja modernisoimalla käyttöliittymän toimintaa.

4.3.5 Yhteenveto ideaaliratkaisusta

Käytännössä ideaaliratkaisu olisi suuri arkkitehtuurinen ja erityisesti toimitusputkellinen muutos verrattuna järjestelmän nykytilaan. Ideaaliratkaisun toteuttamiseksi järjestelmä tulisi kehittää uudestaan, koska teknologiat muuttuisivat täysin ja taustalle rakennettavat automaattiset toimitusputket vaatisivat uusien automaattisten testien ajamista.

Ideaaliratkaisussa järjestelmä olisi arkkitehtuuriltaan tavanomaisempi web-sovellus ja huomattavasti vähemmän tietokantakeskeinen. Tietokantaliitosta hoitaisi parhaassa tapauksessa abstraktimpi ORM-järjestelmä ja tietokanta toimisi vain tietovarastona. Ohjelmistologiikka olisi siirrettynä moderneilla tekniikoilla toteutettuihin front- ja backend-komponentteihin.

Erityisen suuri muutos tehtäisiin tapaan kehittää ja toimittaa järjestelmää. Järjestelmä päivittyisi tuotanto- tai vähintään testiympäristöön automaattisesti onnistuneen versiohallintaan viennin jälkeen. Versiohallintajärjestelmänä käytettäisiin modernimpaa Git-järjes-

telmää. Jotta automaattisesti integrointeja ja toimituksia voitaisiin tehdä, tarvittaisiin automaattitestit. Tämä puolestaan vaatisi kokonaan uuden testivetoisen kehitystavan käyttöönoton, jossa testit kirjoitetaan ennen varsinaista ohjelmistokoodia.

5. PÄÄTETYT TOIMENPITEET

Järjestelmän versiohallinnan tilan parantamiseksi on tehty suunnitelma, josta osa valittiin toteutettavaksi ensimmäisessä vaiheessa. Tässä luvussa kuvataan toteutettavaksi valitut toimenpiteet, niin kuin ne oli etukäteen ajateltu toteutettavan. Sitä, miten toteutus todellisuudessa tapahtui, käsitellään luvussa 6, Tulokset.

Myös muut suunnitelman kohdat on todettu hyödyllisiksi, mutta niiden toteuttaminen on jätetty odottamaan henkilöstöresurssien vapautumista tai arviota niiden hyödyllisyydestä suhteessa työmäärään. Tämän diplomityön kirjoitushetkellä ensimmäisessä vaiheessa toteutettavaksi suunnitellut toimenpiteet ovat valmiita. Joihinkin toteuttamista odottamaan jätetyistä kohdista palataan tarkemmin luvussa 7.3, Muutos- ja jatkoehdotukset.

5.1 Tavoitteet

Ennen toimenpiteitä tulee olla selvillä se mitä niillä yritetään saada aikaan. Perusongelma on se, että järjestelmää on paljon vaikeampi kehittää, määritellä ja myydä kuin sen tulisi olla. Se ei tee erityisen monimutkaisia asioita eikä sen tulisi myöskään olla monimutkainen.

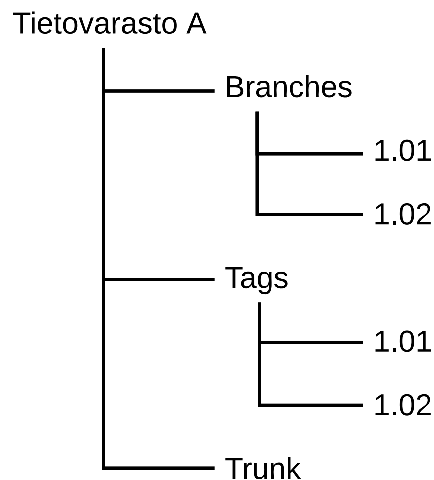
Perusongelma on pääasiassa seurausta järjestelmän tietokantakeskeisyydestä ja kehityksen alkuvaiheessa tehdyistä versiohallinnan käytön puuttumisen yhdistelmästä. Aikaisemmissa luvuissa on käyty läpi syitä, joiden vuoksi tilanteeseen on päädytty ja sen seurauksia. On voitu todeta, että vallalla on noidankehä, joka edelleen huonontaa tilannetta. Alla listatuilla toimenpiteillä pyritään:

1. Vähintään katkaisemaan järjestelmän laatua huonontava noidankehä.
2. Luomaan prosessi, joka kääntää kehityksen positiiviseksi.
3. Saamaan aikaan konkreettisia toimenpiteitä, joilla tilannetta parannetaan.

5.2 Versiohallinnan rakenteen muutokset

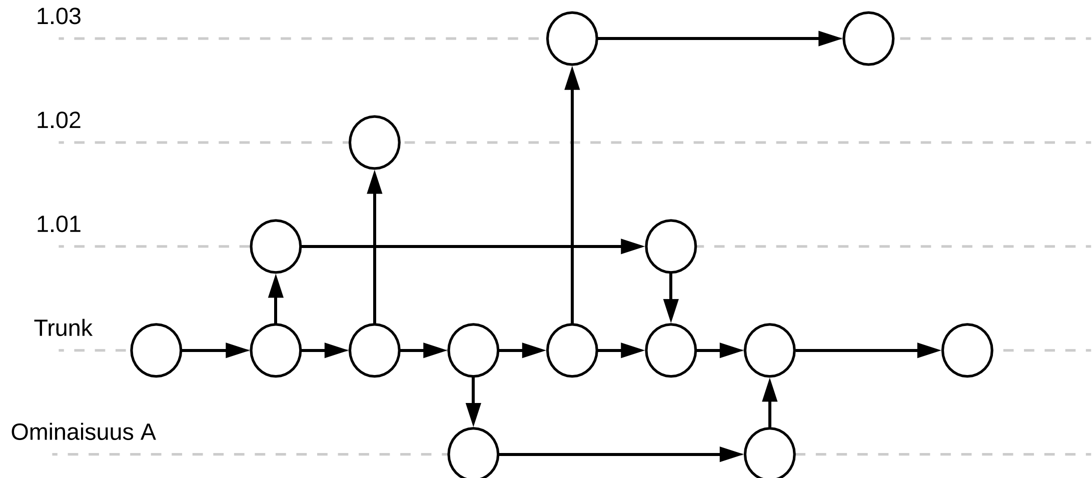
Ensimmäinen päätetty toimenpide oli saattaa järjestelmä versiohallinnan alle. Koko tuotteen versiohallintaan ja myös tämän järjestelmän varmuuskopiotyyliseen versiohallintaan käytettiin jo ennestään Subversionia (SVN). Se olikin luonnollinen valinta versiohallintajärjestelmälle myös laajennetussa käytössä. Kehitystiimillä oli jo ennestään osaamista sen käytöstä, versiohallintapalvelin tietovarastoja varten oli jo olemassa ja käytettyjen työkalujen oltiin todettu tukevan sitä hyvin. Mahdollista migraatiota esimerkiksi Git-versiohallintaan ei ollut tarkoituksenmukaista tehdä samassa yhteydessä – oltiin jo ennestään todettu työtä olevan enemmän kuin aikaa.

Tyypillisessä SVN:n työnkulussa kehitystyötä varten on perustettu oma keskustietovarastonsa, josta kehittäjät hakevat (checkout) työkopion omalla koneellaan olevaan tietovarastoon. Kehittäjät tekevät muutoksia paikallisen työkopion Trunk- eli kehityshaaran tiedostoihin ja vievät ne keskustietovarastoon. Mikäli konflikteja muiden kehittäjien muutoksiin tulee, ratkaistaan ne tässä vaiheessa. Kun tulee aika tehdä ohjelmistosta julkaisu, luodaan kehityshaarasta uusi hakemisto Tag- ja Branch-haaroihin. Tag-haaran ohjelmistokoodi jää versiohallintajärjestelmään tässä muodossa, kun taas Branch-haaraan tehdään julkaisuun kohdistuvia muokkauksia. Tästä muodostuu kuvassa 6 esitetty hakemistorakenne. Kaikista toimenpiteistä saadaan uniikki globaali revisionumero. [4]



Kuva 6. *Tyypillisesti SVN:ssä käytetty hakemistorakenne [4].*

Jokaisesta järjestelmän tietokannasta luotiin oma ydinkantansa kehityspalvelimen tietokantapalvelimen uusimpaan käytössä olevaan instanssiin (SQL Server 2014). Ydinkannan pohjaksi otettiin aikaisempaa toimitustapaa käyttäen uusin ja parhaassa tilassa oleviksi arvioidut tietokannat. Tämän oli tarkoitus olla viimeinen näin tehty haarautuminen ja tästä eteenpäin kaikki uudet toimitukset tultaisiin tekemään näistä ydinkannoista tehtävien julkaisujen kautta. Ydinkantoja muodostettiin yhteensä kuusi kappaletta



Kuva 7. *Sovittu tapa käyttää versiohallintaa.*

Aikaisemmin tietokantoja oli versiohallinnoitu vain projektikohtaisiin tietovarastoihin, jotka luotiin toiminnanohjausjärjestelmän integraation kautta projekteja perustettaessa. Järjestelmälle perustettiin oma toimitusprojekteista irrallinen kehitystietovarastonsa, jonka Trunk-hakemistoon ydinkannoista vietiin Redgaten SQL Source Control -ohjelmalla SQL-komentokieliset tiedostot. Nyt järjestelmän kehityshaara oli luotu. Haara on kuvattu kuvassa 7 ”Trunk”-haarana.

Nyt tästä haarasta voidaan tehdä julkaisuja normaalin SVN:n työnkulun mukaisesti. Lisäksi kuvattiin mahdollisuus käyttää erillisiä haaroja monimutkaisempia uusien ominaisuuksien (feature) kehittämiseen. Nämä haarat luodaan kehityshaarasta omalla erillisellä tavalla nimettyyn hakemistoonsa (Feature A, Feature B, …). Kun ominaisuuden kehitystyö on valmiina, yhdistetään (merge) muutokset kehityshaaraan. Tapa on esitetty kuvassa 7 alimpana.

Myös toimitusprojektien yhteydessä saatetaan kehittää uutta toiminnallisuutta tai tehdä bugikorjauksia. Myös niiden yhdistäminen kehityshaaraan kuvattiin. Tämän ja ominaisuushaarojen käytön on tarkoitus pitää kehityshaaran koodi ehjänä koko ajan. Mikäli kehitys tehtäisiin suoraan kehityshaaraan, olisi kuvassa 7 esitettyyn 1.03-julkaisuun päätynyt rikkinäistä ohjelmistokoodia uuden ominaisuuden kehityksen ollessa vielä kesken.

5.3 Julkaisujen tekeminen

Kun jotain tiettyä toimitusprojektia varten tarvitaan uusi julkaisu, luodaan Trunk-haarasta julkaisun versionumeroa vastaavat Tag- ja Branch-hakemistot, jotka on kuvattu kuvassa 7 yksinkertaistettuna vain versionumerolla. Tag-haara on kuva julkaisuhetkellä olleesta tilasta.

Varsinainen työkopio otetaan Branch-haaran hakemistosta ja kaikki kehitys viedään siis sinne. Julkaisuhetkellä muodostuu uniikki revisionumero. Revisionumero olisi mahdollista hakea pelkästään versiohallinnan historiatiedosta erityisesti kuvaavan kommentin avulla, mutta revisiohistorian haun nopeuttamiseksi se dokumentoidaan myös järjestelmän Wiki-järjestelmään versionumeron lisäksi.

Kaikki toimitusprojektikohtainen kehitys tehdään kunkin julkaisun haaraan ja näin saadaan kehityksen kulusta historia SVN:ään. Kehityshaarasta näihin julkaisuihin on periytynyt kaikki sinne viety uusi kehitys ja bugien korjaukset. Kehitystyö onkin siis lähinnä ei-haluttujen ominaisuuksien poiskytkemistä eikä niiden tuomista muualta tai uudestaan kehittämistä.

5.4 Moduulirakenteen palauttaminen alkutilaan

Järjestelmä oli alun perin suunniteltu hyvin modulaariseksi. Kaikki sen sisältävät toiminnot olisivat tietokannan taulukossa listattuna ja niitä voitaisiin yksinkertaisesti kytkeä päälle ja pois yhdellä bitillä. Joillekin toiminnoille tämä toiminnallisuus toimii vieläkin, mutta useimmat uudet ominaisuudet on kehitetty ohittaen modulaarinen konfigurointitaulu ja kirjoittamalla toiminnot suoraan koodiin.

Kehittäjät ovat kehuneet tätä ominaisuutta. Esimerkiksi koko käyttöliittymän sivujärjestyksen vaihdot ja sivujen poiskytkemiset toimivat tällä tavalla. Pois haluttavan sivun aktivointibitti asetetaan nollaksi eikä muuta tarvitse tehdä. Tapa on todettu erittäin tehokkaaksi, kun se toimii suunnitellusti. Onkin siis päätetty lähteä palauttamaan toiminnon eheyttä kohti alkutilaa myös uusien kehitettyjen toimintojen osalta.

Kaikki toimintoja ei saada istumaan moduulirakenteeseen joko ilman kohtuutonta vaivaa tai monimutkaistamalla itse moduulien valintaa liikaa. Kaikkea ei siis tarvitsekaan muokata tukemaan sitä. Kuitenkin pieni lisävaiva menetelmään soveltuvien ja jatkuvasti päälle tai pois kytkettävien moduulien kohdalla olisi säästänyt vaivaa myöhemmin, ja moduulirakenteen palauttamiseen onkin siis perusteltua pyrkiä pienestä lisätyömäärästä huolimatta.

5.5 Moduulirakenteen käytön laajentaminen

Koska moduulirakenne on todettu olevan hyödyllinen sen toimiessa, on päätetty lähteä laajentamaan sen käyttöä. Moduulilla on aikaisemmin tarkoitettu käyttöliittymän kokonaisia sivuja tai muita suurempia kokonaisuuksia. Nyt lähdetään myös selvittämään pienempien ominaisuuksien toteuttamista samalla tavalla.

Tyypillisesti jonkin moduulin sisäisen toiminnon aktivoimiseksi kehittäjän on tarvinnut käydä sen generoivaa logiikkaa läpi ja tehdä siihen muokkauksia. Jotkut näistä toiminnoista ovat kuitenkin tyyliltään hyvin modulaarisia ja niiden on todettu sopivan hyvin

tämän tyylisen konfiguroitavuuteen. Näiden toimintojen käyttö vaatii uusia rakenteita ja uusia tapoja aktivoida niitä sivujen sisällä ja niitä lähdetään kehittämään samassa yhteydessä.

5.6 Logiikan siistiminen

Järjestelmän ohjelmistokoodin tason on todettu huonontuneen ja huonontuvan edelleen. Useimmat muokkaukset on ajateltu joko kertakäyttöisiksi tai myöhemmin peruutettaviksi. Joskus jopa molempia yhtä aikaa. Esimerkiksi mikäli jokin ominaisuus halutaan kytkeä pois päältä, on se yleensä vain kommentoitu pois ajatellen, että se saatetaan haluta kytkeä takaisin päälle myöhemmin. Mahdollinen korvaava toiminto on saatettu kehittää sen tilalle huolimattomasti. Käytännössä koskaan poiskommentoituja ohjelmistokoodilohkoja ei ole kuinkaan tarvittu myöhemmin.

Nämä muutokset ovat kuitenkin periytyneet tuleviin toimituksiin hallitsemattomasti. Kerran huonontunut koodikuri on ruokkinut itseään ja vuosien saatossa järjestelmän ohjelmistokoodista on muodostunut melko huonotasoista. Kun versiohallinnan käyttö muutetaan, saadaan pysäytettyä ei-toivottujen muutosten periytyminen uudempiin järjestelmiin. Samassa yhteydessä on hyvä aloittaa järjestelmän ohjelmistokoodin tason ja rakenteen parantaminen siistimällä sitä.

Kaikki käytössä olevat proseduurit ja funktiot listattaisiin ja tehtäisiin päätös niiden mahdollisesta uudelleennimeämisestä tai osiin pilkkomisesta. Erityisen hyödyllistä päivittäisessä kehitystyössä on, kun kehitystyökaluna käytettävässä Microsoftin SQL Management Studiossa ne listautuisivat loogisiin paikkoihin. Lista näytetään aakkosjärjestyksessä. Esimerkiksi get-alkuiset proseduurit ryhmitellään vierekkäin. Näiden proseduurien nimet voivat olla sinänsä järkeviä, mutta eri get-toiminnot liittyvät täysin eri asioihin.

Monet käyttöliittymästä kutsuttavat toiminnot on koottu samoihin proseduureihin ja ryhmitelty siellä eri haaroihin kutsun mukaan. Monia näistä haaroista ei enää kutsuta mistään, jotkut ovat väärässä paikassa ja toisaalta esimerkiksi napin painallukselle on useampi proseduuri. Ryhmitellään ja siivotaan nämä uudestaan. Ylipäätään parannetaan vuosian kuluessa huonontuneen koodin tasoa. Jo monimutkaisen koodilohkojen selkokielinen kommentointi auttaisi paljon. Parasta olisi, mikäli ohjelmistokoodi olisi itsedokumentoituvaa, mutta tiimin mielestä SQL kielenä ei sovi tähän kovin hyvin – ohjelmistokoodi on täynnä pitkiä SELECT-lauseita ja monimutkaisia JOIN-rakenteita, joiden ymmärtämiseen tulee aina paneutua lukemista syvemmin.

5.7 Luodaan koodistandardi

Ohjelmistokoodin logiikan siistimiseen liittyy olennaisena osana yhtenäinen tapa kirjoittaa sitä. Järjestelmää kehittäessä ei aikaisemmin ole ollut yhteisesti noudatettavaa, yh-

dessä sovittua tapaa kirjoittaa ohjelmistokoodia. Eri kehittäjät ovat nimenneet ja ryhmitelleet muuttujia, proseduureja ja muita toimintoja toisistaan poikkeavalla tavalla sekä kommentoineet ohjelmistokoodia omalla tavallaan. Erityisesti puutteellisen kommentoinnin on nähty olevan haitallista kehitystyölle.

Nyt kirjoitettaisiinkin ylös yhtenäinen ohjelmistokoodistandardi, jossa päätettäisiin mm. muuttujien nimeämistapa, riittävän kattava tapa kommentoida ohjelmistokoodia, tapa ryhmitellä ohjelmistokoodin haaroja (erityisesti takaisinkutsuhaarat), tapa sijoittaa niitä proseduureihin ja nimetä itse proseduurit tai funktiot. Samalla sovittaisiin tapa kirjoittaa muuta ohjelmistokoodia kuten PHP:ta tai Javascriptiä. Koodistandardin tulisikin siis kattaa myös tuotteen muut komponentit.

5.8 Kirjoitetaan kuvaus tyypillisestä toimitusprojektista

Monesti järjestelmän toimitusprojekteissa tehtävien kuvauksissa tai myynnin toimesta viitataan normaaliin tapaan. Tämä ei ole välttämättä huono tapa: sillä pystyttäisiin periaatteessa kuvaamaan monimutkainen ominaisuus hyvin lyhyesti ja samalla kirjoittamaan kehitystehtävä keskittymällä vain olennaiseen eli muutoksiin. Samalla myös vastataan suoraan kysymyksiin kaikista niistä toiminnoista, joita ei ole erikseen mainittu.

Kuitenkin, koska jokainen toimitus on jossain määrin kustomoitu, hämärtyy se, mitä ”normaali” tarkoittaa. Kokeneemmilla kehittäjillä on kokemattomia tarkempi käsitys sen merkityksestä ja eri kehittäjä voikin päätyä samaa tehtäväkuvausta tehdessään eri ratkaisuun. Kuitenkin Scrum-tyyppisessä kehitystavassa kaikkien kehitysjonon tehtävien tulisi olla kuvattuna niin, että mahdollisimman moni pystyy ne toteuttamaan – eikä kehitystehtävää kirjoittaessa olekaan yleensä tiedossa, kuka sen tulee tekemään [33].

Muutosten yhteydessä siis kirjoitettaisiinkin auki kattava kuva järjestelmän tyypillisestä toiminnasta ja kattavuudesta niin, että kaikki harvinaiset kustomoinnit on poistettu. Kuvaus auttaa myös kehityshaaran toimintojen ja ohjelmistokoodin palauttamisessa eheään muotoon, koska se tarjoaa tiedon siitä missä tilassa minkäkin toiminnon tulisi olla. Samalla myös tarpeettomat toiminnot pystytään tunnistamaan.

5.9 Parannetaan tietokannan tasoa yleisesti

Järjestelmään on aikaisemmin tehty kehitysponnistuksia esimerkiksi taulujen pääavainten (primary key) ja pääavaimille automaattisesti muodostuvien indeksien lisäksi tarpeellisten indeksien luomiseksi. Myös niitä koskevat muut haarautumisesta aiheutuvat ongelmat: ei ole varmuutta ovatko kaikki niistä periytyneet uusimpiin tietokantaversioihin. Myöskään uusille tauluille samaa systemaattista ponnistusta ei ole tehty ja se tehtäisiinkin muutosten yhteydessä.

Järjestelmän tietokannoista ei myöskään ole piirretty ainoatakaan UML-kaaviota. Kaaviot ovat erityisen hyödyllisiä antamaan yleiskuva tietokannan taulurakenteesta ja kuvaamaan taulujen keskinäisiä relaatioita ja siis niiden käyttöä [34]. Muutosten yhteydessä selvitettäisiinkin siis työkaluja kaavioiden piirtämiseen ja ehkä generoimiseen automaattisesti.

Kummankin muutoksen tekeminen vaatii kohtalaista tietokantojen suunnittelun teoreettista osaamista. Esimerkiksi indeksejä ei tule lisätä turhaan, koska jokainen indeksi kasvattaa tietokannan kokoa eikä välttämättä tarjoa mitään tehokkuushyötyä vaan saattaa aiheuttaa jopa hidastumista [1]. Kehittäjillä tulisikin siis olla kyky tehdä tietoon perustuvia arvioita niiden hyödyistä. Heillä on eritasoisia ohjelmistotekniikan koulutustaustoja eikä kaikilla välttämättä ole valmiuksia näiden muutosten toteuttamiseen. Tarvittaessa kehittäjille tuleekin siis tarjota tukea ja koulutusta.

5.10 Luodaan muistilista tehtävistä asioista

Osittain tyypillisen toimituksen kuvaukseen liittyen on tuotu esiin tarve yksinkertaisille muistilistoille. Muistilistalle merkittäisiin tyypillisesti tarkistettavia, tehtäviä tai muistettavia asioita. Sen tulisi olla riittävän yksinkertainen, jotta sen läpikäyminen olisi mahdollista joka kerta siinä missä toimituksen tyypilliseen kuvaukseen turvauduttaisiin vain tarvittaessa. Toisaalta muistilistan tulisi olla riittävän kattava, jotta siinä on kaikki tarpeellinen.

Erityisen hyödyllinen muistilista olisi uusien kehittäjien tullessa tiimiin. Siitä saisi yksinkertaisen tavan kertoa kaikki oleelliset asiat. Silti muistilista olisi hyödyllinen myös kaikille muille. Sillä varmistetaan se, ettei vähitellen lipsuta eri asioiden tekemisessä, ylipäätään estetään asioiden unohtelua ja nopeutetaan niiden läpikäymistä, kun kaikki ovat suoraan käsiteltävällä listalla kunkin muistin sijaan.

5.11 Toimenpiteiden työmääräarvio

Toimenpiteet oli alustavasti suunniteltu ennen niiden toteuttamista ja niiden työmäärä arvioitiin kehitystiimin avulla ennen niiden toteuttamista käyttäen *planning poker* -menetelmää. Menetelmässä tehtävä kuvataan ja jokainen kehittäjä arvioi itsenäisesti sille pistemäärän. Tähän käytetään yleensä fyysisiä pelikortteja tai mobiiliapplikaatiota, jotta jokainen joutuu antamaan ensimmäisen arvion itsenäisesti. Mikäli arviot esimerkiksi sanottaisiin ääneen, vaikuttaisi myöhemmin kerrottuihin arvioihin jo annetut arviot. Ensimmäisten arvioiden antamisen jälkeen tiimi keskustelee arviosta ja pyrkii pääsemään konsensukseen. Muista poikkeavan pistemäärän antaneet kertovat omalle arviolleen perustelut. Lopulta keskustelun jälkeen päädytään yhteen arvioon, jossa on kuultu kaikkia tiimin jäseniä. [35]

Pistemäärä on abstrakti luku, joka ei itsessään vastaa mitään tuntimäärää tai henkilötyöpäivää. Jokainen tehtävä arvioidaan suhteessa muihin: ”…koska tämä tehtävä oli 3 pistettä, täytyy tämän suuremman tehtävän olla 5 pistettä”. Pistemäärät asettuvat tiimikohtaisesti jollekin tasolle ja niille voidaan laskea historiallinen työtuntimäärä tehtäville tehtyjen tuntikirjausten kautta. Tätä kautta voidaan arvioida myös kunkin tehtävän euromääräistä hintaa. Tämän diplomityön tarkoitusta varten riittää kuitenkin pelkkä pistemäärä.

Taulukko 1. *Toimenpiteitä edeltävä työmääräarvio.*

Toimenpide	Kehittäjien ensimmäiset arviot						Arvio
	1	2	3	4	5	6	
5.2 Versiohallinnan rakenteen muutokset	3	3	6	4	6	7	6
5.3 Julkaisujen tekeminen	?	3	1	1	3	5	3
5.4 Moduulirakenteen palauttaminen alkutilaan	?	5	?	5	13	8	13
5.5 Moduulirakenteen käytön laajentaminen	40	20	40	13	20	100	40
5.6 Logiikan siistiminen	40	100	100	100	40	?	100
5.7 Luodaan koodistandardi	2	3	3	5	3	3	3
5.8 Kirjoitetaan kuvaus tyypillisestä toimitusprojektista	1	2	5	8	3	3	3
5.9 Parannetaan tietokannan tasoa yleisesti	3	5	8	20	8	3	5
5.10 Luodaan muistilista tehtävistä asioista	2	2	5	2	3	1	2

Taulukossa **1** on listattu luvun alussa kuvatulla menetelmällä saadut työmääräarviot. Arviot on tehty toimenpiteiden alkuperäisille kuvauksille ja joidenkin niiden mittakaava on muuttunut arvion tekemisen jälkeen. Joitain toimenpiteitä on myös yhdistelty. Tämän vuoksi toimeenpiteen ”5.2 Versiohallinnan rakenteen muutokset” arvio on summa kahdesta eri arviosta.

Ensimmäinen arvio kuvaa kehittäjien ensimmäistä itsenäistä ajatusta työmäärästä. Suuri vaihtelu tai kysymysmerkit niissä kertoo siitä, että työmäärää on vaikea arvioida. Joko työ on kuvattu puutteellisesti ja se on ymmärretty eri tavalla, tai ei ollenkaan, tai työssä on ylipäätään paljon epävarmuuksia. Korkean arvion saanutta toimenpidettä ei voi lähteä toteuttamaan sellaisenaan vaan se tulee jakaa useampaan osaan.

6. TULOKSET

Työn kirjoitushetkellä ensimmäiset toimenpiteet on toteutettu. Tässä luvussa esitellään toimenpiteiden tulokset. Tuloksia verrataan etukäteen tehtyihin suunnitelmiin ja ideaalitilanteeseen. Lisäksi tuodaan esiin muutokset, joita ei ennakoitu, mutta jotka päädyttiin toteuttamaan.

Toteutettaviin toimenpiteisiin vaikuttivat arviot niiden hyödyllisyydestä sekä niille arvioitu työmäärä. Monet toimenpiteet arvioitiin sinänsä hyödyllisiksi, mutta niitä ei päätetty toteuttaa niiden suuren työmäärän vuoksi. Koska kaikkea ei voitu tehdä kerralla, tärkeintä oli aloittaa kaikista hyödyllisimmistä tai niistä, joilla saadaan prosessissa vallalla olevia noidankehiä katkaistua. Tietyt toimenpiteet vaativat lisäksi toisia toteutettavaksi niitä ennen. Esimerkiksi julkaisuja ei voida tehdä ennen kuin versiohallinnan rakennetta on muutettu.

6.1 Toteutetut suunnitellut toimenpiteet

Vain muutama tärkein muutos päädyttiin toteuttamaan ensimmäisessä vaiheessa. Useampiakin olisi haluttu toteuttaa, mutta resurssipaineessa niitä ei pystytty ottamaan mukaan. Tärkeäksi katsottiin kuitenkin päästä toteuttamaan edes joitain muutoksia.

6.1.1 Versiohallinnan rakenteen muutokset

Merkittävin suunniteltu ja toteutettu toimenpide oli muutos versiohallinnan käyttöön. Subversionin ja versiohallinnan käyttö yleisesti oli tiimille jo ennestään tuttua eikä siihen tarvinnut käyttää aikaa.

Tuotteen jokapäiväisessä käytössä oleva dokumentaatio on kerätty sisäiseen Wiki-järjestelmään ja tässä yhteydessä sinne luotiin uusi sivu, jossa kuvattiin kaavioin ja tekstein se tapa, jolla jatkossa versiohallintaa halutaan käyttää. Sivu pyrittiin pitämään kattavana, mutta samalla mahdollisimman kompaktina. Sivulle aloitettiin myös keräämään listaa tehdyistä julkaisuista luvussa 5.3, Julkaisujen tekeminen kerrotuista syistä.

Samassa yhteydessä myös kuvattiin mahdollisuus kehityshaarojen käyttöön. Kehityshaaroilla on tarkoitus estää keskeneräisen työn periytyminen uusiin julkaisuihin. Tämä ongelma on uusi ja johtuu siitä, että julkaisuja ylipäätään aloitetaan tekemään. Myös sen ohjeistus kirjoitettiin Wiki-järjestelmään.

Kuitenkaan tämän diplomityön kirjoitushetkellä yhtään kehityshaaraa ei ole luotu, koska niin suuria uusia ominaisuuksia ei ole kehitetty, että tarvetta niille muodostuisi. Kehityshaarojen käyttöä ollaan kuitenkin ajateltu kokeiltavan pienemmilläkin ominaisuuksilla,

jotta niiden käyttö olisi tuttua oikean tarpeen tullessa eteen. Vähintään olisi hyödyllistä, jos tällöin olisi olemassa ainakin yksi esimerkki hakemistorakenteineen.

Myös korjausten ja uuden kehityksen tuominen kehityslinjaan kuvattiin Wiki-järjestelmään. Tällä varmistetaan se, että samaa kehitystyötä ei tehdä kahdesti. Ei voitu kuitenkaan kirjoittaa mitään erityisen kattavaa kuvausta siitä mitä muutoksia tarkalleen halutaan säilyttää.

Onkin tärkeää, että jokainen kehittäjä pyrkii arvioimaan tekemäänsä työtä ja vaivautuu viemään sen tarvittaessa myös kehityslinjaan. Muutosten tuominen on pitkälti käsityötä ja vaatii SQL Source Control -työkalun käyttöä eli se sisältää myös tämän ylimääräisen vaiheen. Alkuvaiheessa tarkoitus oli vain tuoda muun kehityksen yhteydessä orgaanisesti tehtyjä parannuksia ja korjauksia. Varsinainen kaiken muun logiikan siistimistä käsitellään tarkemmin luvussa 6.2.1, Logiikan siistiminen.

6.1.2 Julkaisujen tekeminen

Kun uusi tapa käyttää versiohallintaa oli kuvattu, päätettiin tehdä ensimmäinen julkaisu heti seuraavasta toimitusprojektista. Varsinainen kehityshaarojen tietokantojen luonti ja versiohallinnointi oltiin päätetty tehdä sitä mukaan, kun kyseisiä kantoja tarvittiin toimitusprojekteissa. Tietovaraston hakemistorakenne haaroille ja julkaisuille oli kuvattu Wiki-järjestelmään kirjoitetussa ohjeessa.

Tämä projekti oli laajuudeltaan melko suuri ja sen yhteydessä saatiinkin käsiteltyä suurin osa järjestelmän tietokannoista. Projektista luotiin julkaisu 1.01 ja seuraava olisi 1.02. Tämän diplomityön kirjoitushetkellä julkaisuja on tehty seitsemän kappaletta ja kaikki järjestelmän kannat on viety kehityshaaraan. Nyt julkaisuihin tehty poikkeuksellinen kehitystyö ei enää periytyisi uusiin julkaisuihin.

6.2 Suunnitellut toteuttamattomat toimenpiteet

Toteutettujen toimenpiteiden lisäksi oltiin suunniteltu monia muita, joita ei päädytty toteuttamaan.

6.2.1 Logiikan siistiminen

Kehitystiimi näki logiikan siistimisen tärkeimmäksi yksittäiseksi toimenpiteeksi, jolla parannetaan järjestelmän ylläpidettävyyttä ja kehitystä. Toisaalta se myös arvioitiin työmäärältään erittäin suureksi, suuremmaksi kuin mikään muu toimenpide. Lisäksi ennen kuin ohjelmistokoodin siistimistä kannattaa aloittaa, tulisi ennen vallalla ollut koodia huonontava noidankehä katkaista. Tämä pakotti toteuttamaan versiohallinnan rakenteen muutokset siistimistä ennen.

Toimenpidettä pitäisi pilkkoa pienempiin osiin, jotta siitä saataisiin ylipäätään toteutuskelpoinen. Sen arvio oli 100 pistettä eikä näin suuria tehtäviä edes voida ottaa scrum-projektinhallintaviitekehyksen kuvaaville kehityssprinteille. Ei myöskään ole mielekästä tehdä pelkkää vanhan ohjelmistokoodin siistimistä koko tiimin voimin pitkää aikaa. Erityisesti ennen näin suurta työtä tulisi tehdä päätös siitä kuinka paljon logiikkaa kannattaa siistiä verrattuna uuden järjestelmän kehittämiseen. Todennäköisesti kyseeseen tulisi vain tärkeimpien asioiden siistiminen.

6.2.2 Luodaan koodistandardi

Vaikka yhtenäiselle koodistandardille nähtiin tarvetta, ei hieman eri tyyliin kirjoitetun ohjelmistokoodin nähty olevan erityisen suuri ongelma. Lisäksi koodistandardin tulisi kattaa koko tuote eli myös muut komponentit kuin tässä diplomityössä käsiteltävä järjestelmä ja niiden kehittämiseen käytetyt ohjelmistokielet. Siis myös kaikki tämän järjestelmän kehitystiimin ulkopuoliset kehittäjät tarvitaan mukaan standardin sopimiseen. Sen tulisi myös jo kattaa tai pystyä laajentumaan kattamaan tulevaisuuden järjestelmien kehitys. Kaikista näistä syistä toimenpidettä ei päätetty toteuttaa ensimmäisten joukossa.

6.2.3 Kirjoitetaan kuvaus tyypillisestä toimitusprojektista

Toimenpidettä ei katsottu tärkeimpien joukkoon vaikkakin se olisi vähätöinen. Päätettiin ensisijaisesti välttää tyypilliseen toteutukseen viittaamista ja mikäli se ei riitä, toteutetaan kuvaus myöhemmin.

6.2.4 Parannetaan tietokannan tasoa yleisesti

Erityisesti hyödyllisten indeksien systemaattinen käyttö todettiin hyväksi. Testeissä hakuja oli saatu nopeutettua merkittävästi. Tällä on järjestelmän käyttöliittymän tietokantoihin takaisinkutsuvasta rakenteesta johtuen suuri hyöty. Toimenpide olisi suuren hyödyn lisäksi pieni työmäärältään.

Toimenpidettä ei kuitenkaan päädytty toteuttamaan ensimmäisten joukossa, koska edelleen tietokantaa ei kannattanut alkaa siistiä ennen versiohallinnan rakenteiden muutoksia eikä kaikilla kehittäjillä välttämättä ollut riittäviä valmiuksia indeksien hyödyllisyyden arviointiin. Toimenpide on kuitenkin suunnitteilla lähitulevaisuudessa.

6.2.5 Luodaan muistilista tehtävistä asioista

Järjestelmästä on koottu useita muistilistoja käyttöönottoa tai palvelinasennuksia varten ja niiden on todettu olevan hyödyllisiä. Muistilista myös kehitystyötä varten katsottiin järkeväksi erityisesti suhteessa sen tekemiseen arvioituun työmäärään. Sitä ei kuitenkaan

katsottu kaikista kriittisimmäksi eikä siis päätetty ottaa ensimmäisessä vaiheessa tehtäviin muutoksiin mukaan.

6.3 Suunniteltujen muutosten lisäksi tehdyt toimenpiteet

Suunniteltujen toimenpiteiden lisäksi päädyttiin toteuttamaan yksi ennalta suunnittelematon toimenpide. Se liittyi suoraan versiohallinnan rakenteen muutoksiin. Toimenpiteen työmäärää ei myöskään arvioitu etukäteen.

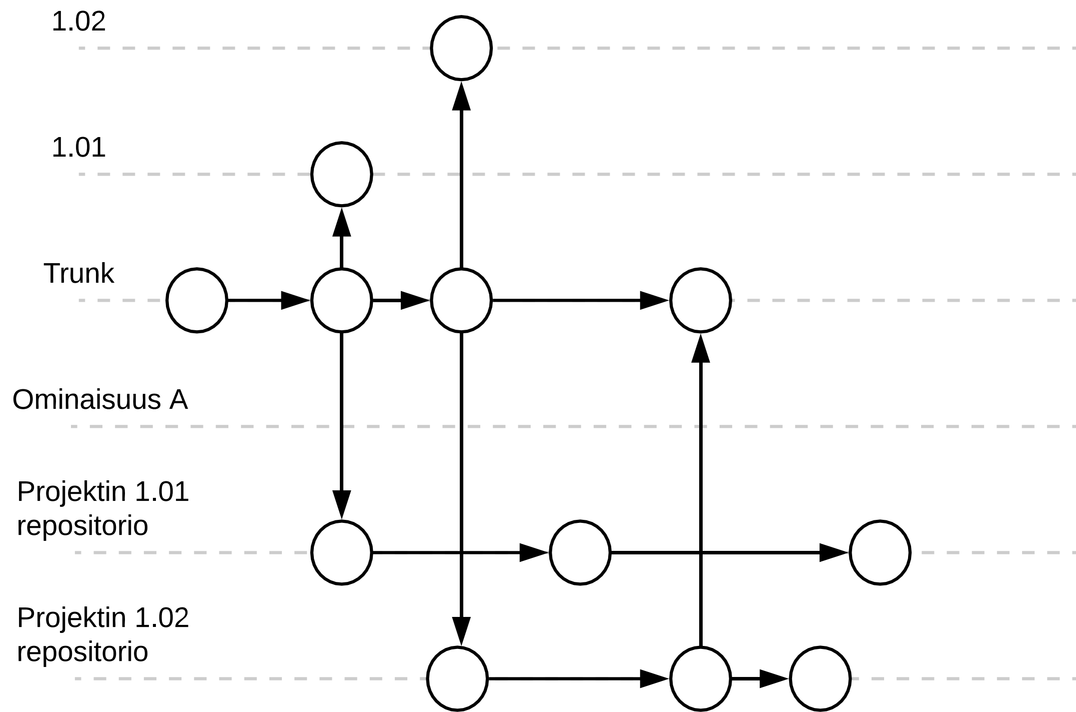
6.3.1 Tuotteen teknisen laadun heikkenemisen lopettaminen

Erityistä painoarvoa päätettiin antaa muutosten tärkeimmälle tavoitteelle eli tuotteen teknisen laadun heikkenemisen noidankehän katkaisemiselle. Sen sijaan, että kuvattaisiin vain itsessään byrokraattisia toimenpiteitä, haluttiin korostaa haluttua lopputulosta ja kannustaa kehittäjiä pyrkimään siihen.

Väliaikaisten muutosten kohdalla painotettiin niiden jäämistä vain kyseisen toimitusprojektin julkaisuun. Toisaalta korostettiin tavoitetta tuoda luvussa 6.1.1, Versiohallinnan rakenteen muutokset kuvatut positiiviset muutokset toimitusprojekteista kehityslinjaan.

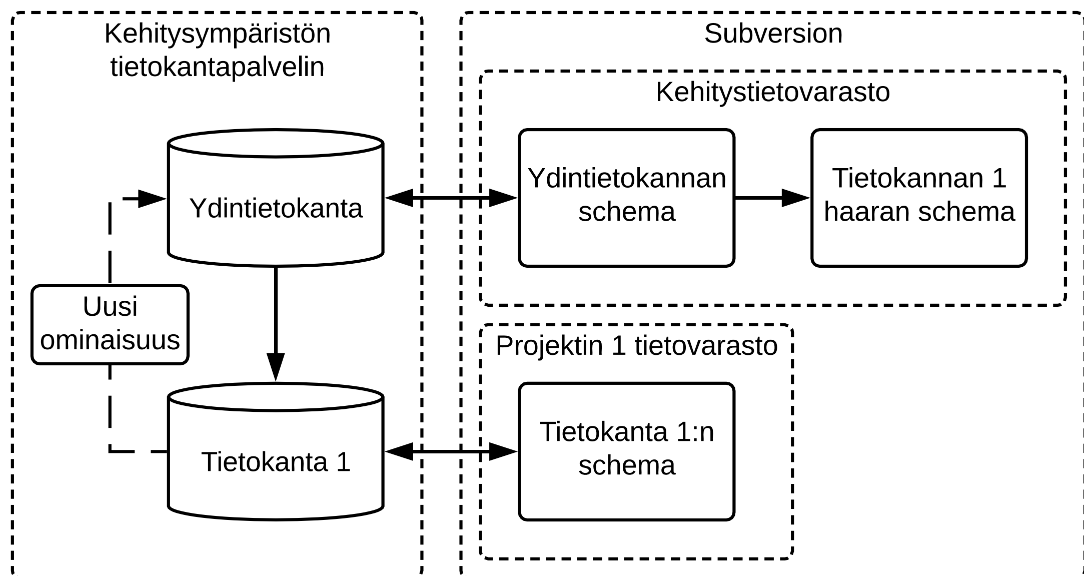
6.4 Tilanne muutosten jälkeen

Kun ensimmäiset toimenpiteet oli saatu toteutettua, huomattiin että oltiin tehty ajatusvirhe versiohallinnan käytössä, eikä todellinen käyttö vastannut ennalta suunniteltua. Samalla oli yritetty saada sisällytettyä kaksi toisensa poissulkevaa asiaa: projektikohtaisten tietovarastojen käyttö ja järjestelmän kehitystietovaraston sisällä tehtävien haarojen luonti.



Kuva 8. *Todellisuudessa toteutunut versiohallinnan käyttö.*

Kuvassa 8 on kuvattu se tapa käyttää versiohallintaa johon todellisuudessa päädyttiin. Versionumeroituja julkaisuja on alettu tekemään järjestelmän kehityshaarasta, mutta niiden lisäksi on tehty kopio projektien tietovarastoihin.



Kuva 9. *Versiohallinnan rakenne muutosten jälkeen.*

Kuvassa 9 kuvataan versiohallinnan rakenne tietovarastotasolla toteutettujen toimenpiteiden jälkeen. Tilannetta on hyvä verrata kuvassa 3 esitettyyn alkutilanteeseen.

Juuri projektikohtaiset tietovarastot on kytketty tietokantakehityspalvelimella oleviin tietokantoihin käyttäen SQL Source Control - työkalua ja siis niihin tehty kehitys on päätynyt vain projektien tietovarastoihin. Haluttu kehitystyö on kuitenkin nyt yhdistetty järjestelmän kehityshaaraan näistä ulkopuolisista tietovarastoista. Käytännössä siis kehitystietovaraston haarat (branch) ovat olleet vain toisia kopioita niiden julkaisuista (tag). On siis päädytty tekemään päällekkäistä työtä.

7. TULOSTEN ARVIOINTI

Tässä luvussa arvioidaan tulosten vaikutusta alkutilanteeseen. Tuloksia verrataan etukäteen suunniteltuihin toimenpiteisiin ja ideaalitilanteeseen. Lisäksi ehdotetaan muutoksia ja hyödyllisiä jatkotoimenpiteitä.

7.1 Ero alkutilanteeseen

Järjestelmän versiohallinta oli jo ennestään monimutkaisempaa kuin vastaavan järjestelmän, jossa ohjelmistologiikkaa ei olla toteutettu tietokannassa. Ylimääräinen vaihe tarvittiin ohjelmistokoodin generoimiseksi tietokannasta levylle tallennettaviin komentokielisiin tiedostoihin.

Tämä tilanne ei ole muuttunut vaan versiohallinnasta on tullut entistäkin monimutkaisempaa. Nyt tarvitaan lisätyötä ympäristöjä perustettaessa toimitusprojekteja varten, kun järjestelmästä tehdään julkaisuja ja jatkuvasti eri julkaisujen kehityksen aikana haluttuja muutoksia ja korjauksia kehityshaaraan tuodessa. Tämä pitää sisällään sekä SQL Source Control -työkalun käyttöä että käsityötä.

Ongelmien juurisyyksi todettu resurssipula ja siitä seurannut kiire ei ole ratkennut, ja on itse asiassa pahentunut muutosten tekemisestä tämän diplomityön kirjoitushetkeen tultaessa. Kehittäjiltä on tullut palautetta ylimääräisestä työstä. Muutosten tarve on sinänsä ymmärretty ja toimenpiteet on arvioitu hyviksi. Niitä ei kuitenkaan olla osattu tehdä kasvattamatta rutiinityön määrää.

Jo ennestään kehittäjiä oli vaikea motivoida huolehtimaan versiohallinnan käyttöön eikä sen käyttötapa vieläkään pakota viemään muutoksia versiohallintaan tuotantoympäristöön tehtävien toimitusten yhteydessä. Paine oikoa sinänsä yhteisymmärryksessä sovitusta käyttötavasta on siis olemassa. Lisäksi versiohallinnan käytön monimutkaistuessa tilaisuudet tehdä virheitä kasvavat.

Nykytilanteessa kehityshaaraan haluttavien ominaisuuksien yhdistämisessä ei voida käyttää Subversionin merge-ominaisuutta ainakaan helposti, koska vietävä koodi on eri tietovarastoissa eri nimisissä tietokannoissa ja siis siitä johtuen eri hakemistorakenteissa. Tällöin ei myöskään saada käyttöön Subversionin versiohistorian etuja yhdistämisiin liittyen. Uuden ominaisuuden yhdistäminen kehityshaaraan näyttää versiohallinnan näkökulmasta samalta kuin kokonaan uutena kehitetty ohjelmistokoodi, koska se on lisätty kehityshaaran kehitystietokantaan käsin.

Tavoite oli vähintään järjestelmän laadun heikkenemisen lopettaminen ja mielellään sen parantaminen, tai ainakin prosessin luominen, jossa se vähitellen parantuu. Kaikkien tavoitteiden tarkoituksena oli helpottaa järjestelmän kehitystä, määrittelyä ja myyntiä, ja vähentää näihin kuluvaa työtä.

Nyt siis käytetään lisätyötä, jotta tuotteen teknisen laadun heikkeneminen on saatu katkeamaan. Ideaalisesti se tapahtuisi orgaanisesti. Kiistatta laadun heikkeneminen on kuitenkin saatu katkaistua. Väliaikaiseksi ajatellut muutokset jäävät aina vain yksittäisen julkaisun (tai oikeastaan toimitusprojektin) tietokantaan, kun versiohallintaa käytetään uudella sovitulla tavalla ja oleellisesti myös silloin, kun sitä ei käytetä niin.

Toimenpiteiden työmäärää arvioitiin vain arvioimalla välittömästi niihin tehtävää työtä. Arvioimatta jätettiin prosessimuutoksista aiheutuva rutiinien aikaansaama työmäärä myöhemmän kehityksen yhteydessä. Tätä arvioita ei olisi pystytty tekemään alkuvaiheessa eikä myöhemminkään merkittävällä tarkkuudella. Selvää on kuitenkin se, että tämä työmäärä on suurempi kuin nyt tehdyissä toimenpiteissä.

7.2 Ero ideaaliratkaisuun

Lähdettäessä toteuttamaan ensimmäisiä toimenpiteitä päätettiin olla edes pyrkimättä ideaaliratkaisuun. Ideaaliratkaisuun pääseminen olisi vaatinut käytännössä järjestelmän kehittämisen uudestaan puhtaalta pöydältä eikä siihen katsottu olevan riittäviä resursseja.

Oleellisin ja muut ideaaliratkaisun seikkojen mahdollistava toimenpide olisi ollut tietokantakeskeisyyden hylkääminen. Nykyisen järjestelmän muokkaaminen tätä varten olisi ollut selvästi kannattamatonta ja toimenpiteiden jälkeenkin järjestelmän logiikka on edelleen toteutettuna tietokannan proseduureissa, funktioissa ja liipaisimissa.

Myöskään toimitustavan muutosta CD/CI-prosessilla ei otettu käyttöön. Tarvittavien työkalujen ja ympäristöjen pystytys lienee mahdollista nykyisessäkin järjestelmässä, mutta jotta automaattisesti integroidun ja toimitetun järjestelmän toimintaan tuotantoympäristössä pystytään luottamaan, tarvitaan kattavat automaattitestit. Testien kirjoittaminen jälkikäteen nykyiseen järjestelmään olisi ollut liian suuritöistä eikä tarvittavia toimenpiteitä päätetty tehdä.

Järjestelmän tavanomaisemman rakenteen käyttöönotto olisi myös vaatinut sen uudelleenkirjoittaminen. Ei ole mahdotonta siirtää proseduureihin, funktioihin ja liipaisimiin kirjoitettu logiikka tietokannan rakenteista pois, mutta nykyinen kevyt backend-ratkaisu ei toteutustapansa puolesta tarjoa kätevää paikkaa sille. Mikäli näitä rakenteita olisi lähdetty toteuttamaan, olisi herännyt myös kysymys tämän logiikan versiohallinnoinnista yhdessä tietokannan logiikan kanssa. Nykytilanteessa tietokannan ulkopuolinen backend on versiohallinnoitu omassa paikassaan yhdessä frontend-toteutuksen kanssa. Mikäli lo-

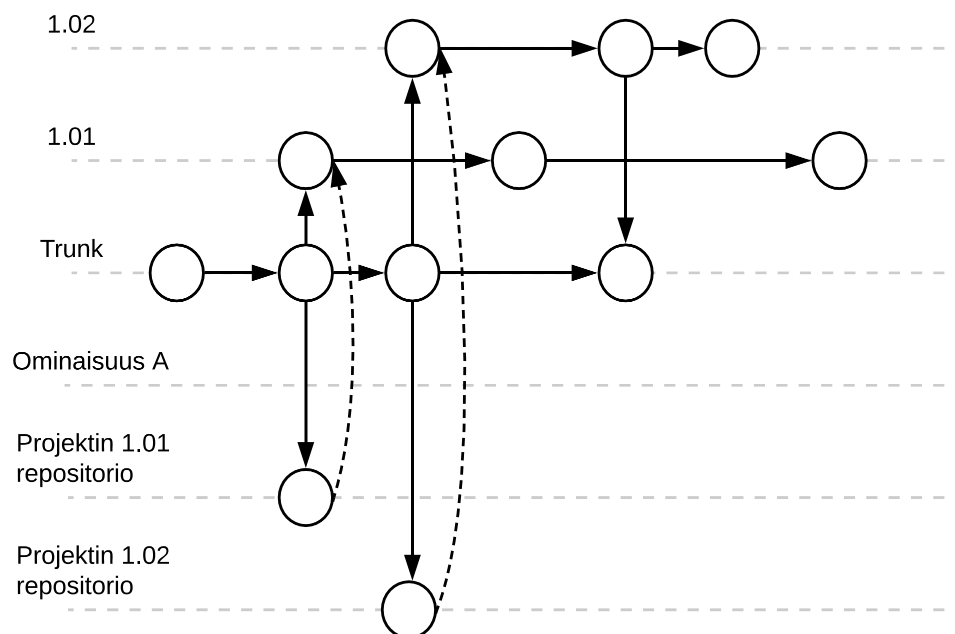
giikkaa lähdettäisiin siirtämään pois tietokannasta, tulisi logiikkaa sisältävä backend versiohallinnoida sen kanssa yhteen sopivan tietokantaversion kanssa, mikä monimutkaistaisi järjestelmän versiohallinnan tilannetta entisestään.

Ainoa nykyjärjestelmään realistisesti ideaaliratkaisusta toteutettavissa oleva toimenpide olisi Git-versiohallintajärjestelmän käyttöön siirtyminen. Tätäkään ei päätetty toteuttaa, koska ideaaliratkaisun toimitustavan muutosten vaatimia työkalujakaan ei tarvittu. Git voisi kuitenkin olla jo yksinään parempi kuin SVN ja sen käyttöä aiotaankin kokeilla toisen järjestelmän kehityksen yhteydessä. Todennäköisesti kaikki uusi tuotteen kehitystyö tullaan jatkossa versiohallinoimaan Git-järjestelmässä.

7.3 Muutos- ja jatkoehdotukset

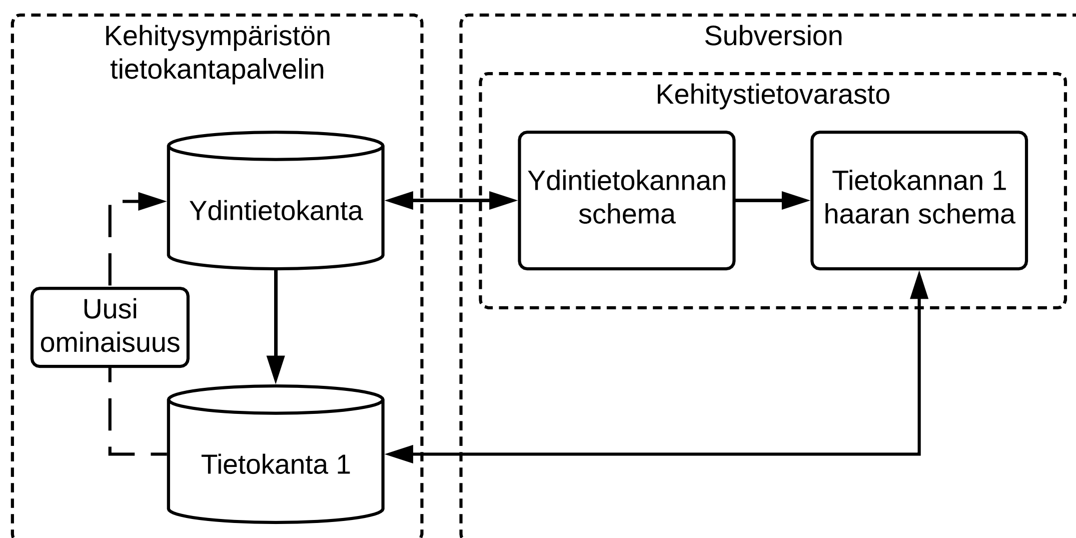
7.3.1 Korjaukset tapaan käyttää versiohallintaa

Kun uusi tapa käyttää versionhallintaa oli otettu käyttöön, todettiin luvussa 6.4 toteutuneen ajatellusta poikkeava tapa. Poikkeavuus johtuu asiakaskohtaisten tietokantojen versiohallinnoinnista edelleen asiakaskohtaiseen tietovarastoon. Muutosten jälkeen on käytössä myös toinen paikka kehitystietovaraston haaroissa. Julkaisuja näihin haaroihin on tehty, mutta todellisuudessa niitä ei kuitenkaan päivitetä vaan päivitykset tallennetaan vain asiakaskohtaisiin tietovarastoihin.



Kuva 10. *Ehdotus korjata versiohallinnan käyttö.*

Muutosten yhdistäminen julkaisuhaaraan asiakaskohtaisesta tietovarastoista olisi mahdollista, mutta suuritöistä jo ennestään monimutkaisessa tavassa käyttää versiohallintaa. Lisäksi ratkaisu ylläpitää kahta eri paikkaa tallentaa koodia on jo lähtökohtaisesti huono. Ehdotetaankin siis tehtäväksi vielä yksi muutos tapaan käyttää versiohallintaa: yksinkertaistetaan prosessia hylkäämällä asiakaskohtaiset tietovarastot kuvassa 10 kuvatulla tavalla.



Kuva 11. *Versiohallinnan ehdotettu rakenne.*

Jatkossa tehtäisiin vain julkaisuhaara kehityshaarasta ja julkaisuun tehtävät muutokset tallennetaan suoraan sinne. Lopputuloksena saataisiin kuvassa 11 esitetty rakenne. Jo tehtyjen julkaisujen muutokset yhdistettäisiin ennestään käyttämättömiin julkaisuhaaroihin käsin ja asiakaskaskohtaisista tietovarastoista poistettaisiin ohjelmistokoodi. Tietovarastojen historia jäisi kuitenkin SVN-järjestelmään talteen. Lisähyötynä saavutettaisiin se, että kehittäjille ei tarvitsisi antaa oikeuksia useisiin tietovarastoihin vaan oikeudet yhteen riittäisivät.

7.3.2 Tehdään suunnitelma uuden järjestelmän kehitysaikataulusta

Oleellinen harkittava seikka legacy-järjestelmän tason parantamisessa on se miten paljon työtä kannattaa käyttää vanhan korjaamiseen verrattuna uuden tekemiseen. Tähän pohdintaan mennään syvemmin luvussa 7.4, mutta ennen kuin mitään arvioita toimenpiteiden tulevasta hyödyllisyydestä voidaan tehdä, tulee olla edes karkea käsitys puhuttavista aikaikkunoista.

Mikäli nykyistä järjestelmää ei aiota korvata uudella koskaan, kannattavat suuritöisetkin toimenpiteet ja toisaalta mikäli uusi järjestelmä aiotaan kehittää alle vuoden sisään, ei kannata käyttää aikaa juuri minkään toimenpiteen tekemiseen. Vaikka uutta järjestelmää

on päätetty olla kehittämättä toistaiseksi, tullee se kuitenkin ajankohtaiseksi lähitulevaisuudessa. Ehdotetaankin siis konkreettisen ja realistisen suunnitelman tekemistä sekä tämän järjestelmän että tuotteen muiden komponenttien elinkaarista.

7.3.3 Siistitään tärkein osa ohjelmistokoodista

Ohjelmistologiikan ja -koodin siistiminen on toistuvasti mainittu kehittäjien suunnalta tulleessa palautteessa tärkeäksi. Nyt tehdyillä versiohallinnan käytön toimenpiteillä on saatu vaikutettua siihen, ettei ohjelmistokoodi enää jatka huononemista, mutta toisaalta uusi prosessi ei itsessään paranna vanhaa ydinversioon päätynyttä huonoa ohjelmistokoodia.

Luvussa 5.11, Toimenpiteiden työmääräarvio logiikan siistiminen arvioitiin työmäärältään suurimmaksi yksittäiseksi toimenpiteeksi, eikä sitä siksi päätetty toteuttaa ensimmäisessä vaiheessa. Työmääräarvio tehtiin kaiken järjestelmän ohjelmistokoodin siistimisille. Näin suuritöinen toimenpide tulisi joka tapauksessa paloitella osiin ja ensimmäisessä vaiheessa ehdotetaan eriteltäväksi kaikkein oleellisin osa ohjelmistologiikasta sekä tehtävän toimenpidesuunnitelma sen siistimiseksi.

Tallaista logiikkaa on ohjelmistokoodi, johon tulee tarve tehdä muokkauksia erityisen usein tai ohjelmistokoodi, joka on niin huonoa tai monimutkaista ettei sitä voi ilman suurta työtä käyttää uudelleen ollenkaan. Toisaalta taas ohjelmistokoodi, johon kosketaan harvoin voi jäädä nykytilaan, vaikka se olisikin melko huonoa.

Käytännössä ohjelmistokoodin tasoa parannettaisiin järjestelemällä sitä paremmin poistamalla pitkiä poiskommentointeja, kommentoimalla ohjelmistokoodia huolellisemmin, purkamalla monimutkaisia IF-ELSE -haaroja sekä pilkkomalla niitä eri proseduureihin ja funktioihin sekä yleisesti siistimällä ohjelmistokoodin rakennetta ja nimeämisiä. Ohjelmistokoodin parannukset joko tehdään suoraan kehityshaaraan tai toimitusprojektin yhteydessä julkaisuhaaraan, josta ne yhdistetään kehityshaaraan.

7.3.4 Selvitetään parempia tapoja käyttää kehitystietokantoja

Nykytilanteessa kehitettävät tietokannat sijaitsevat kehityspalvelimella ja kehittäjillä on niihin yhteinen pääsy. Tästä aiheutuu jonkin verran ongelmia kuten päällekkäisten muutosten tekeminen yli kirjoittaen toisten kehittäjien tekemiä muutoksia, tietokannan datan muuttuminen kesken kehityksen ja testiluontoisen ohjelmistokoodin päätyminen häiritsemään kehitystä.

Ehdotetaankin selvitettävän olisiko kannattavampaa siirtyä kehittäjien työasemilla pyörivien tai jollain muulla tavalla hajautettujen tietokantojen käyttöön kehitystyössä. Tällöin muutosten integroimiseen vaadittaisiin versiohallinnan käyttöä ja muutokset päätyisivät sinne automaattisesti. Samalla tulisi selvittää kuinka vaikeata käytännössä on integroida

ohjelmistologiikan muutoksia eri tietokantojen välillä pitäen mielessä se, että jo monimutkaista prosessia ei ole järkevää monimutkaistaa lisää.

Versiohallinnan muutoksilla on saatu vaikutettua kehitystietokantojen aiheuttamiin ongelmiin, kuten esimerkiksi siihen missä oleva ohjelmistokoodi määrää ajantasaisen tilanteen. Nyt voidaan todeta versiohallinnassa olevan koodin tekevän sen. Entä ylläpitotyyppinen data? Olisi selvitettävä myös mahdollisuutta versiohallinnoida valikoitua dataa – kaiken datan tallentaminen veisi liikaa tilaa eikä ole tarpeenkaan.

Lisäksi tulisi selvittää ratkaisuja eri tietokantainstanssien versio-ongelmiin. Kehityshaaran tietokannat ovat SQL Serverin uusimmassa tietokantahallintajärjestelmän versioinstanssissa, jota käytetään kehitystyössä. Nykytilanteessa saattaa tulla vastaan tilanne, jossa on tarve tehdä uusi julkaisu vanhaan ympäristöön ja siis vanhaan tietokantahallintajärjestelmän versioon. Tukea tähän ei ole vaan ainoastaan vanhemmasta uudempaan [36].

7.3.5 Muutetaan organisaatiota kehitystiimien ajan myymiseksi

Useimpien ongelmien merkittävimpänä juurisyynä voidaan pitää kroonista resurssipulaa, joka on saanut aikaan sen, ettei asioita ole alun perin ollut aikaa tehdä riittävän hyvin. Koko juurisyy voitaisiin ohittaa myymällä kehitystiimien aikaa tuotteen sijaan.

Nykytilanteessa toimitusprojekteja myydään urakkahinnoittelulla, joissa myytävä tuote ja hinta on määritetty etukäteen. Ketterää ohjelmistokehitystä on kuitenkin mahdollista myydä myös tuntilaskutteisesti niin, että asiakas ostaa osaavan tiimin aikaa ja asiantuntemusta. Tyypillisesti ei luvata mitään toimitusaikaa vaan järjestelmää kehitetään iteratiivisesti, kunnes se on tiiviisti kehityksessä mukana olevan asiakkaan mielestä valmis [33]. Tällöin resurssiongelmaa ei ole – aikaa ja rahaa kehitystyön tekemiseen kunnolla loppuun asti on automaattisesti.

Järjestelmää kehitetään jo Scrum-mallilla, joka itsessään sopii hyvin tämän tyyppisiin sopimuksiin [37]. Tiimit ovat kuitenkin organisoitu arkkitehtuurillisesti niin, että mitään yksittäistä tiimiä ei voida irrottaa asiakkaan käyttöön, koska muut tiimit eivät osaa samoja asioita eikä mikään yksittäinen tiimi osaa kaikkia asioita.

Ehdotetaankin että kehitystiimit organisoidaan uusiksi niin, että jokainen pystyy tekemään toimituksia kaikista tuotteen osista. Tällöin mahdollistetaan ketterä myynti. On luultavaa, että kaikki raskaan teollisuuden asiakkaat eivät halua ostaa tällä tavalla eikä

mitään estettä myydä urakkamuotoisia projekteja saa muodostua vaan organisaation tulee pystyä molempiin.

7.3.6 Teknisen velan määrän arvioiminen ja tekeminen näkyväksi

Tässä diplomityössä on jo paneuduttu teknistä velkaa aiheuttaviin syihin ja korjaustoimenpiteiden työmäärän arviointiin sekä edelleen työmäärien ja hyötyjen arvioinnin vaikeuteen. Useimmat ongelmista seuranneet konkreettiset asiat ovat käytännössä teknistä velkaa ja tulisikin siis tehdä erillisiä ponnistuksia tai prosessimuutoksia sen määrän tekemiseksi näkyväksi.

Pyrittäisiin kvantifioimaan tai muodollisesti arvioimaan nykyisessä järjestelmässä olevan teknisen velan määrää. Käydään ohjelmistokoodia joko erikseen tai muun kehitystyön yhteydessä läpi ja pyritään tunnistamaan ongelmia. Aloitetaan luomaan konkreettisia korjaustehtäviä tuotteen backlogille merkittynä tekniseksi velaksi. Tehtävät voidaan arvioida kuten muutkin kehitystehtävät ja näin saadaan numeraalinen arvio velan määrästä.

Suurin osa teknisestä velasta on kuitenkin näkymätöntä [13]. Näkymättömän teknisen velan tekemiseksi näkyväksi pyrittäisiin selvittämään ohjelmallisia työkaluja ohjelmistokoodin käsittelemiseksi. Pelkkä ohjelmallinen läpikäyminen ei kuitenkaan riitä näkymättömän teknisen velan tunnistamiseksi, koska työkalut eivät tunnista rakenteellisista tai arkkitehtuurillisista valinnoista johtuvaa teknistä velkaa [13]. Kuitenkin osa voidaan tunnistaa ohjelmallisesti ja se voi olla selvästi kustannustehokkain tapa.

Pelkkä yksittäinen kehitysponnistus ei riitä tähänkään vaan tarvitaan prosessimuutoksia, jotta jatkossa kertyvä tekninen velka saadaan systemaattisesti identifioitua – tietenkin niiden prosessimuutosten lisäksi, joilla pyritään alun perin ehkäisemään teknisen velan aiheutumista. Käytetty ketterä kehitysmenetelmä mahdollistaa periaatteessa teknisen velan ripeän korjaamisen, mutta vain kun se on tunnistettu [13].

Kun vanhan teknisen velan määrästä on saatu edes jonkinlainen arvio, voidaan sen korjaamiseen liittyvää työtä ottaa hallitusti kehityksen alle tai tietoa voidaan käyttää hyödyksi arvioitaessa tarvetta ja aikataulua mahdollisen korvaavan järjestelmän kehittämiseksi.

7.4 Kannattiko vanhan korjaaminen?

Tässä yhteydessä legacy-järjestelmän korjaamisen järkevyyttä arvioidaan kokonaan uuden, korvaavan järjestelmän kehittämiseen. Kolmas vaihtoehto olisi ajaa järjestelmän toimitus ja käyttö alas kokonaan, mutta tämä ei ole liiketoimintanäkökulmasta vaihtoehto eikä muutenkaan tarkoituksenmukainen vertailukohde. Olemassa olevan järjestelmän

korjaamiseen voidaan nähdä monta eri näkökulmaa: liiketoimintanäkökulma, kehittäjien tyytyväisyysnäkökulma tai tuotteen ohjelmistoteknisen laadun näkökulma.

Kaikkien näkökulmien kvantifioiminen on vähintäänkin erittäin hankalaa, ellei mahdotonta. Perusajatus on se, että nyt oikein tehty työ säästää enemmän työtä jatkossa, koska kehittämisestä saadaan tehtyä helpompaa eikä samaa työtä tarvitse enää tehdä moneen kertaan. Tämä ajatus lienee selkeä, mutta hyötyjen arvioinnissa on paljon epävarmuutta. Liiketoimintanäkökulmassa pyritään arvioimaan korjauksiin käytettyä aikaa suhteessa myöhemmin toimitusprojekteissa saataviin aikahyötyihin.

Kuitenkaan mitään hyvää arviota tulevaisuuden hyödyistä ei saada laskettua. Henkilöstön tyytyväisyyden ja tuotteen laadun mittaaminen on vielä hankalampaa. Voidaan tehdä henkilöstökyselyitä tai esimerkiksi mitata huoltotehtävien määrää, mutta minkään yksittäisen toimenpiteen suoraa vaikutusta ei pystytä erottamaan muusta kohinasta.

Kuinka pitkällä aikavälillä hyötyjä tullaan saamaan? Mikäli aikaväli on liian lyhyt, ei välttämättä saadakaan alun aikainvestointia vastaavaa hyötyä. Aikavälin selvittämistä sivuttiin luvussa 7.3, ”Muutos- ja jatkoehdotukset” eikä sitä tiedetä vielä tätä diplomityötä kirjoittaessa. Kaikkia järjestelmän yksittäisiä asioita ei tarvita yhtä usein. Jos korjauksia tehdään kaikkiin niistä yhtä suurella työmäärällä, jotkut korjaukset voivat kannattaa ja jotkut eivät. Entä jos korjauksissa epäonnistutaan eikä tulevaisuuden hyötyjä saadakaan?

Kysymyksiin ei ole yksiselitteistä ja selkeää vastausta. Toimenpiteet arvioitiin työmäärän mukaan ja niitä päätettiin toteuttaa alkaen kaikista hyödyllisimmiksi arvioiduista. Voidaan olla melko varmoja, että ensimmäisessä vaiheessa toteutetut toimenpiteet olivat kannattavia, mutta sitä missä kohtaa raja menee ei pystytä nykytiedoilla arvioimaan. On kuitenkin selvää, että nykyinen järjestelmä tullaan joskus korvaamaan uudella ja siis raja on olemassa jossain kohtaa.

Yllä oleva pohdinta liittyy lähinnä liiketoimintanäkökulmaan, jossa aika on sama asia kuin raha. Entä henkilöstön tyytyväisyys- ja laatunäkökulmat? Kehittäjiltä toimenpiteistä saatu palaute on ollut pääasiassa positiivista, mutta kaikista mieluiten he kehittäisivät kokonaan uuden järjestelmän tyhjältä pöydältä haluamillaan tekniikoilla. Toisaalta jos uusi järjestelmä kehitetään, ei varmasti voida tarjota kaikkia miellyttäviä ratkaisuja teknisesti ja muuten toteutuksellisesti.

Painetta järjestelmän kehittämiseen tulee siis monista eri suunnista: paine kehittää tuotteen laatua, paine pitää henkilöstö tyytyväisenä ja paine pystyä tekemään taloudellisesti kannattavia asiakastoimituksia järjestelmästä. Vastapainona tälle paineelle toimii yleinen resurssipaine ja tarve ottaa nykyiseen järjestelmän versioon tehdystä kehitystyöstä kaikki taloudellinen hyöty irti. Kun selviä vastauksia ei saada, tulee arvioinnin olla jatkuvaa. Tulee pyrkiä tiedostamaan arvioinnin sokeat pisteet ja tulee miettiä tapoja avata niitä.

Mikäli juurisyitä ei korjata, saattaa olla, että samoja tai uusia ongelmia aiheuttavia virheitä tehdään myös uuden järjestelmän kehitystyössä. Vaikka uuden järjestelmän kehitystyö olisi kehittäjille mieluisaa, ei ole itsestään selvää, että valmistuessaan uusi järjestelmä on kehittäjille nykyistä mielekkäämpi tehdä asiakastoimituksia tai ohjelmistoteknisesti laadukkaampi. Nykyinen henkilöstä ei ole merkittävästi osaavampaa, kokeneempaa eikä kiireettömämpää kuin nykyisen järjestelmän aikanaan kehittänyt henkilöstö.

7.5 Miten lähtötilanteeseen päätyminen voidaan välttää?

Lähtökohtaisesti kaikkien ongelmien voidaan todeta juontuvan resurssiongelmasta: järjestelmän kehitystyöhön ei ollut aikanaan riittävästi aikaa riittävän osaavalla kehitystiimillä. Nykyjärjestelmän ensimmäinen versio kehitettiin hyvin nopeasti. Kun saadaan käytettyä aikaa asioiden tekemiseen oikein heti alussa, vältytään tekniseltä velalta heti alkuunsa ja kaikista niistä ongelmista, joita siitä seuraa. Tällöin saadaan vältettyä noidankehä, josta on erittäin vaikea irtautua. On kuitenkin monta sudenkuoppaa, jotka painostavat oikomaan.

Kehitystiimillä tulee olla riittävästi kokemusta nähdä ongelmat etukäteen. Kokemus tulee virheitä tekemällä ja niistä oppimalla. Lisäksi kokemus antaa varmuutta kyseenalaistaa ratkaisuja. Luultavasti kuka tahansa juuri koulusta valmistunut osaa ihmetellä versiohallinnan puutteita, mutta ei välttämättä lähde kyseenalaistamaan niitä samalla tavalla kuin ongelmien kanssa työskennellyt kehittäjä. Yrityksellä tuleekin olla halu pitää henkilöstöstä kiinni – mikäli kehittäjät päästetään karkaamaan kokemuksen kertyessä ei saada riittävästi kriittisiä ääniä vaikutuksen aikaansaamiseksi. Kokemattomien kehittäjien palkkaaminen on sinänsä mahdollista, kunhan tiimissä on sopiva yhdistelmä kokemusta ja uutta verta.

Myös projekti- ja teknologiajohdon tulee olla riittävän kokenutta. Tyypillisesti tämän järjestelmän projektijohtoon on nostettu henkilöitä tiimistä eikä rekrytoitu ulkopuolelta. Tälle on vahvat syynsä kuten laajan tuotteen tuntemuksen tarve, mutta samalla voidaan siirtää kehittäjien kokemuksen puutetta ylöspäin tasolle, jolla päätökset tehdään. Toisaalta tällä tavoin voi myös saada sitoutettua henkilöstöä pysymään pidempään kehitystiimissä ja kerryttämään arvokasta kokemusta. Tulee kuitenkin huolehtia siitä, että projektijohdon näköalat pysyvät riittävän monipuolisina. Kun sama henkilö on työskennellyt saman järjestelmän parissa pitkään, voi muodostua sokeutta tehtyihin ratkaisuihin eikä vaihtoehtoja välttämättä edes osata nostaa arviointiin. Kehitystiimien kaikilla tasoilla tulee olla riittävä kokemus ja näkemys perustella virheiden välttämistä tai ne päädytään tekemään.

Projektijohto myös tuntee resurssipaineen eri tavalla kuin kehittäjä. Heillä on vastuu toimitusprojektien toteutuksesta aikataulussa ja kannattavasti. Tällöin on helppo nähdä tarve lyhytnäköisille oikaisuille, jotka saattavat välittömästi säästää aikaa. Vaatii myös suurta

vaivannäköä lähteä perustelemaan lisäresurssien käyttöä heti alkuvaiheessa. Monesti kyseessä on taistelu tuulimyllyjä ja liiketoimintarealiteetteja vastaan. Tämä taistelu tulisi kuitenkin aina pystyä käymään sekä projektijohdon omien resurssien puolesta ja pystyä raivaamaan tilaa pitkäjänteiselle kehitystyölle.

Kehitystyössä tulee olla avoin uusille tavoille tehdä asioita. Se, että teknologiat tunnetaan ja osataan, on sinänsä pätevä peruste käyttää niitä, mutta samalla tulisi kyetä ottamaan harkintaan vaihtoehtoja ja tekemään valistuneita päätöksiä niiden väliltä. Tässä työssä kuvatussa järjestelmässä on päädytty ylikäyttämään SQL-osaamista. Kehitystiimillä tulisikin olla tarvittaessa valmiudet siirtyä muiden tekniikkojen käyttöön tarvittaessa. Valmiudet kasvavat kokemuksen kertyessä ja tätä kautta palataan taas kehitystiimin kokemuksen tärkeyteen.

Lähtökohtaisesti siis moni ongelma lähtee resurssiongelmasta. Hyvällä HR-asioiden hoidolla päästään jo pitkälle ongelmien syntymisen ehkäisyssä. Sillä saadaan henkilöstö pysymään tiimissä pidempään eivätkä resurssiongelma pääse kasaantumaan. Samalla henkilöstö myös ehtisi kerryttämään enemmän kokemusta ja olisi motivoituneempaa kehittämään tuotetta tulevaisuudessakin. Monesti kehittäjän piilevät lähtöaikeet huomaa ensimmäiseksi siitä, että enää ei yritetäkään muuttaa asioita.

Liiketoiminnasta vastaavan johdon tulee pystyä näkemään alkuvaiheessa tehtyjen virheiden euromääräinen arvo pitkällä aikavälillä. Tämän arviointi on erittäin hankalaa, koska tilanne kehittyy hitaasti. Alussa virheiden tekeminen saattaa jopa näkyä positiivisena tuloksena. Tarvitaankin hyvää, pitkäjänteistä näkökykyä ja strategia, joka tukee kauaskantoisia ratkaisuja.

8. YHTEENVETO

Tässä diplomityössä esiteltiin tietokantakeskeinen legacy-järjestelmä, jossa ei oltu aluksi käytetty ollenkaan versiohallintaa ja myöhemminkin versiohallinnan rooli oli ollut vain varmuuskopiotyylinen. Järjestelmän ohjelmistologiikka oli kehitetty pääasiassa tietokannan proseduureihin, funktioihin ja liipaisimiin. Tämän logiikan versiohallinnointi oli osoittautunut huomattavasti normaalia versiohallinnan käyttöä vaikeammaksi. Muut järjestelmän osat olivat toteutettuna hyvin kevyesti ja vaikka ne olivat versiohallinnoituna paremmin, oleellisin osa ohjelmistologiikkaa ei ollut.

Toimitusten vaatimia julkaisuja oli tehty ottamalla kopioita vanhoista ympäristöistä ja muokkaamalla uutta tarkoitusta varten. Ajan kuluessa näin tehdyistä haarautumisista oli muodostunut unohdettu puurakenne. Tilanteeseen oli alun perin päädytty resurssipulan ja siitä seuranneen kiireen vuoksi. Järjestelmä oli aikanaan kehitetty hyvin nopeasti. Osatekijöiksi voitiin katsoa kehittäjien kokemattomuus ja väärät tekniset valinnat, mutta juurisyy kaikelle oli resurssipula.

Vallitsevasta tilanteesta todettiin aiheutuvan paljon ongelmia monesta eri näkökulmasta. Ohjelmistokehittäjän työstä tuli vaikeampaa ja työläämpää. Kehittäjä joutui etsimään tarvittavia toimintoja eri julkaisuista ja integroimaan niitä käsin toisiin. Jo tehtyä työtä väistämättä hukkui ja sitä päädyttiin tekemään moneen kertaan. Monien harvinaisten ominaisuuksien olemassaolo oli täysin kehitystiimin muistin varassa ja kun henkilöstöä on vuosien varrella vaihtunut, oli tätä tietoa hävinnyt.

Yrityksen organisaatiossa projektipäällikkö tuottaa kunkin projektin toteuttamiseksi tarvittavat kehitystehtävät kehitystiimien toteutettavaksi. Järjestelmän tekninen rakenne on yhtä monimutkainen heille. Lisäksi asioiden ymmärtäminen vaikeutuu entisestään, koska projektin ymmärrettävä laajuus kasvaa teknisestä tasosta enemmän asiakasvaatimusten toteuttamiseen – kehittäjä saattaa pystyä miettimään tiettyä toiminnallisuutta sellaisenaan, kun projektipäällikön tulee ymmärtää mitä sillä halutaan saada aikaan. Liiketoiminnallisesti puolestaan kaikki ylimääräinen työ näkyy tarpeettomasti kasvaneina kuluina liiketoiminnan tuloksessa. Pahinta kaikissa ongelmissa on ollut se, että tilanne on huonontunut hitaasti ja huomaamatta – järjestelmä on alkutilanteessa ollut yksinkertainen ja hyvin sen kehittäneiden kehittäjien muistissa, ja siis hallinnassa.

Ongelmien korjaaminen aloitettiin tekemällä kirjallisuuskatsaus ohjelmistotieteen teoriaan yrittäen löytää teoriatietoa aiheesta ja esimerkkejä käytännön toimenpiteistä vastaavassa tilanteessa. Teoriatiedosta pyrittiin keräämään aiheeseen liittyviä korjaavia toimenpiteitä, hyviä käytäntöjä ja tietoa yleisistä ongelmista sekä riskeistä. Lisäksi käytiin tarkemmin Redgate Software -yhtiön kaupallinen ratkaisu ja hahmoteltiin ideaaliratkaisu

olettaen, että ongelmia voidaan korjata tyhjiössä ilman oikean maailman väistämättömiä resurssipaineita.

Voitiin kuitenkin todeta, että aiheesta on saatavilla huomattavasti vähemmän tutkimustietoa kuin versiohallinnasta yleisesti. Aihetta voisikin olla aiheellista tutkia enemmän. Tietokanta on oleellinen osa useissa nykyaikaisissa järjestelmissä. Onko sen versiohallinta yleisesti yhtä huonosti hoidettu? Onko ongelma yksinkertaisesti ohitettu erilaisilla teknisillä valinnoilla?

Seuraavaksi päätettiin joukko käytännön toimenpiteitä. Toimenpide-ehdotukset arvioitettiin kehitystiimillä käyttäen *planning poker* -menetelmää pisteinä. Tietäen kehitystiimin historiallinen kapasiteetti näinä pisteinä ja pisteiden tekemiseen käytetty aika, voidaan myös arvioida toimenpiteiden euromääräistä hintaa. Lisäksi pyrittiin tunnistamaan kaikista hyödyllisimmät ja toteutuskelpoisimmat toimenpiteet. Lopulta toteutettavaksi valittiin niistä kaikista hyödyllisimmät ja pienitöisimmät eli siis kustannustehokkaimmat.

Osa suunnitelluista toimenpiteistä ehdittiin toteuttaa tämän diplomityön kirjoitushetkeen mennessä ja osa niistä jäi odottamaan resursseja. Lisäksi päädyttiin toteuttamaan muutamia suunnittelemattomia toimenpiteitä. Tässä vaiheessa työtä käytiin läpi oikeasti toteutuneet tulokset.

Tämän jälkeen arvioitiin tulosten vaikuttavuutta ongelmiin ja tehtiin joukko jatkokehitysehdotuksia. Tulosten kvantifioimisen todettiin olevan haastavaa. Pääasiassa kuitenkin kehittäjien palaute oli hyvää, mutta myös kritiikkiä muutoksia kohtaan esitettiin. Erityisesti nostettiin esiin lisääntynyt hallinnollinen työmäärä, jota tarvittiin versiohallinnan haarojen ylläpitoon ja dokumentointiin. Riski tähän oli tullut esiin jo kirjallisuuskatsauksessa ja se todettiin myös käytännössä.

Toimenpiteet kuitenkin onnistuivat estämään hallitsemattoman haarautumisen ja niiden jälkeen päällekkäistä työtä ei tarvitse enää tehdä vaan kaikki halutut ominaisuudet periytyvät uusiin julkaisuihin automaattisesti ilman eri toimenpiteitä. Kovin lähelle ideaalitilannetta ei päästy, mutta ottaen huomioon todellinen resurssitilanne siihen ei edes yritetty päästä.

Jatkokehitysehdotuksia annettiin sekä pohjautuen toteuttamatta jääneisiin toimenpiteisiin, korjauksina jo toteutettuihin ja esitettiin kokonaan uusia seikkoja. Koska juurisyyn todettiin olevan resurssipula, jatkotoimenpide-ehdotusten täytyi mennä pelkkää versiohallinnan käyttöä pidemmälle kuten organisaatiomuutoksiin.

Lopuksi pyrittiin arvioimaan sitä, kannattiko vanhan järjestelmän korjaaminen ylipäätään. Missä pisteessä on kannattavampaa vain kehittää järjestelmä kokonaan uusiksi? Tarkkaa rajaa tähän ei pystytty osoittamaan. Nyt toteutettujen toimenpiteiden voitiin sanoa olevan selvästi järkeviä, mutta toisaalta kaikista suuritöisimmät toimenpiteet olisivat selvästi kannattamattomia.

Samalla myös pyrittiin osoittamaan toimia, joilla lähtötilanteeseen päätyminen voidaan estää jo alkuvaiheessa. Se, kuinka paljon vanhan järjestelmän korjaaminen kannattaa on sinänsä tärkeä kysymys, mutta uuden version kehittäminen järjestelmästä on väistämättä edessä jossain vaiheessa. Tämä vaihtoehto on tuotu esiin tässäkin työssä korjaustoimenpiteenä ja se on erityisesti ohjelmistokehittäjille mieluisa vaihtoehto. Entä jos juurisyitä ei ole korjattu ennen uuden versio kehitystä? On mahdollista, että virheistä ei olla osattu oppia ja on käytetty paljon resursseja uuden järjestelmän kehittämiseen, jota vaivaavat edelleen samat ongelmat.

LÄHTEET

[1] Davidson L, Moss JM, Books24x7 I. Pro SQL Server 2012 Relational Database Design and Implementation.; 1 ed. Berkeley, CA: Springer Verlag; 2012.

[2] Microsoft. Microsoft Data Platform. Saatavissa (viitattu 9.3.2019): https://www.microsoft.com/en-us/sql-server/.

[3] The Apache Software Foundation. Apache Subversion. Saatavissa (viitattu 9.3.2019): https://subversion.apache.org/.

[4] Pilato C, Collins-Sussman B, Fitzpatrick BW. Version Control with Subversion, 2nd Edition. 2nd ed.: O'Reilly Media, Inc; 2008.

[5] The PHP Development Team. PHP: Hypertext Preprocessor. Saatavissa (viitattu 9.3.2019): http://www.php.net/.

[6] Mozilla Foundation. JavaScript | MDN. Saatavissa (viitattu 9.3.2019): https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[7] World Wide Web Consortium. Introduction to HTML. Saatavissa (viitattu 9.3.2019): https://www.w3schools.com/html/html_intro.asp.

[8] World Wide Web Consortium. CSS Introduction. Saatavissa (viitattu 9.3.2019): https://www.w3schools.com/css/css_intro.asp.

[9] The Apache Software Foundation. Welcome! - The Apache HTTP Server Project. Saatavissa (viitattu 9.3.2019): https://httpd.apache.org/.

[10] National Instruments Corporation. What is LabVIEW? Saatavissa (viitattu 9.3.2019): https://www.ni.com/fi-fi/shop/labview.html.

[11] Hinsen K, Läufer K, Thiruvathukal GK. Essential Tools: Version Control Systems. Computing in Science & Engineering 2009;11(6):84-91.

[12] Red Gate Software Ltd. SQL Source Control. Saatavissa (viitattu 6.2.2018): https://www.red-gate.com/products/sql-development/sql-source-control/.

[13] Kruchten P, Nord RL, Ozkaya I. Technical Debt: From Metaphor to Theory and Practice. IEEE Software 2012;29(6):18-21.

[14] McConnell S. Code complete: a practical handbook of software construction. Redmond (WA): Microsoft Press; 1993.

[15] Ståhl D, Bosch J. Modeling continuous integration practice differences in industry software development. The Journal of Systems & Software 2014;87(1):48-59.

[16] Ploski J, Hasselbring W, Rehwinkel J, Schwierz S. Introducing Version Control to Database-Centric Applications in a Small Enterprise. IEEE Software 2007;24(1):38-44.

[17] Cioranu C, Cioca M, Novac C. Database Versioning 2.0, a Transparent SQL Approach Used in Quantitative Management and Decision Making. Procedia Computer Science 2015;55:523-528.

[18] Roddick JF. A survey of schema versioning issues for database systems. Information and Software Technology 1995;37(7):383-393.

[19] Strazdins G. Data Version Control for Relational Databases: Small and Start-up Business Perspective. Baltic Journal of Modern Computing 2016;4(4):978.

[20] Allen KS. Three Rules for Database Work. Saatavissa (viitattu 30.1.2019): https://odetocode.com/blogs/scott/archive/2008/01/30/three-rules-for-database-work.aspx.

[21] Haerder T, Reuter A. Principles of transaction-oriented database recovery. ACM Computing Surveys (CSUR) 1983;15(4):287-317.

[22] Allen KS. Versioning Databases – The Baseline. Saatavissa (viitattu 30.1.2019): https://odetocode.com/blogs/scott/archive/2008/01/31/versioning-databases-the-baseline.aspx.

[23] Allen KS. Versioning Databases – Change Scripts. Saatavissa (viitattu 30.1.2019): https://odetocode.com/blogs/scott/archive/2008/02/02/versioning-databases-change-scripts.aspx.

[24] Allen KS. Versioning Databases – Views, Stored Procedures, and the Like. Saatavissa (viitattu 30.1.2019): https://odetocode.com/blogs/scott/archive/2008/02/02/versioning-databases-views-stored-procedures-and-the-like.aspx.

[25] Allen KS. Versioning Databases – Branching and Merging. Saatavissa (viitattu 30.1.2019): https://odetocode.com/blogs/scott/archive/2008/02/03/versioning-databases-branching-and-merging.aspx.

[26] Skelton M, Fritchey G, Brewer W, Elliott E, Jürgensen K. Database Lifecycle Management. Redgate Software Ltd; 2017.

[27] Keith M, Schincariol M, Keith J, Books24x7 I. Pro JPA 2 : Mastering the Java Persistence API. 1st ed. Berkeley, CA: Apress L. P; 2011.

[28] Gamma E. Olio-ohjelmointi: suunnittelumallit : design patterns. Helsinki: Edita, IT Press; 2001.

[29] A hundred days of continuous integration. Proceedings - Agile 2008 Conference; 2008; pp. 289-293.

[30] Piccolo SR, Frampton MB. Tools and techniques for computational reproducibility. GigaScience 2016;5(1):30.

[31] Jenkins. Jenkins. Saatavissa (viitattu 9.3.2019): https://jenkins.io/.

[32] JetBrains s.r.o. TeamCity: the Hassle-Free CI and CD Server by JetBrains. Saatavissa (viitattu 9.3.2019): https://www.jetbrains.com/teamcity/.

[33] Schwaber K, Sutherland J. The Scrum Guide. ; 2017. Saatavissa (viitattu 4.4.2019): https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf.

[34] Song E, Yin S, Ray I. Using UML to model relational database operations. Computer Standards & Interfaces 2007;29(3):343-354.

[35] Mahnič V, Hovelja T. On using planning poker for estimating user stories. The Journal of Systems & Software 2012;85(9):2086-2095.

[36] Microsoft. MSSQLSERVER_948 - SQL Docs. Saatavissa (viitattu 12.2.2019): https://docs.microsoft.com/en-us/sql/relational-databases/errors-events/mssqlserver-948-database-engine-error?view=sql-server-2017.

[37] Opelt A, Gloger B, Pfarl W, Mittermayr R. Agile contracts: creating and managing successful projects with Scrum. Hoboken, N.J: Wiley; 2013.