KETTERÄ TIETOKANNAN KEHITYS JA VERSIONHALLINTA VERKKO-OHJELMISTOPROJEKTEISSA

Antti Nevala

Opinnäytetyö
Joulukuu 2012

Mediatekniikan koulutusohjelma
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) NEVALA, Antti	Julkaisun laji Opinnäytetyö	Päivämäärä 10.12.2012
	Sivumäärä 27	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty (X)

Työn nimi
KETTERÄ TIETOKANNAN KEHITYS JA VERSIONHALLINTA VERKKO-OHJELMISTOPROJEKTEISSA

Koulutusohjelma
Mediatekniikka

Työn ohjaaja(t)
MANNINEN, Pasi

Toimeksiantaja(t)
Protacon Solutions Oy

Tiivistelmä

Tehtävänä oli tutkia Protacon Solutions Oy:n ohjelmistokehitykseen sopivia menetelmiä tietokannan kehitykseen ja versionhallintaan. Tietokantojen kehitys tarvitsi ketteriin menetelmiin sopivia prosesseja. Tarkoitus oli myös tutkia ketteriin menetelmiin sopivia versionhallintatapoja, joilla saataisiin tehostettua ryhmän ohjelmistokehitystä.

Opinnäytetyössä käydään läpi yleisesti ohjelmistokehityksessä käytettyjen ketterien menetelmien yhteisiä tekijöitä, ja millaisia tapoja on kehitetty soveltamaan näitä menetelmiä tietokannan kehitykseen, sekä miten ryhmän roolit jakautuvat menetelmiä sovellettaessa. Tarkoitus oli saada sekä ohjelmiston että tietokannan kehityksestä mahdollisimman sujuvaa. Versionhallinnasta tutkittiin projekteissa käytettäviä menetelmiä ja erillisiä versionhallintaratkaisumalleja. Ratkaisumalleihin liittyi automaattisten versionhallintaratkaisujen tutkiminen. Näissä keskitytään erityyppisten ratkaisujen toimintaperiaatteisiin ja eroihin.

Tuloksina löytyi käytäntöjä tietokannan ketterään kehittämiseen ja tämän mahdollistavia prosesseja. Mihin kantoihin kehitys tehdään ja kuinka muutokset viedään toisille kehittäjille. Millaisia kantamuutoksia on ja mitä tulee ottaa huomioon näitä tehdessä. Versionhallintaratkaisuissa tuli esille erilaisia malleja suorittaa versionhallintaa. Myös ratkaisujen toimintatavat tulivat selville. Näitä tuloksia voidaan käyttää ketterissä verkko-ohjelmistoprojekteissa pohjana kannan kehitysprosessien luomiseen.

Avainsanat (asiasanat)
Ketterät menetelmät, ohjelmistokehitys, prosessit, tietokanta, verkko-ohjelmisto, versionhallinta

Muut tiedot

Author(s)
NEVALA, Antti

Type of publication
Bachelor´s / Master's Thesis

Date
10.12.2012

Pages
27

Language
Finnish

Permission for web publication
(X)

Title
AGILE DATABASE DEVELOPMENT AND VERSION CONTROL IN WEB DEVELOPMENT PROJECTS

Degree Programme
Media engineering

Tutor(s)
MANNINEN, Pasi

Assigned by
Protacon Solutions Oy

Abstract

The objective was to find methods for agile database development and version control for Protacon Solutions Oy to be used in web application development. Traditional database development methods do not work well with agile software development. Additionally, the objective was to find version control possibilities for databases.

The thesis discusses commonly used agile software development methods, and what kind of possibilities there are to apply these methods to the development of the database. The aim was to run both software and database development as smoothly as possible for programmers. The database version controllers were studied for different approaches and techniques.

The results are used to enable agile development for database. These include development databases, ways of moving changes and publishing versions as well as approaches to change management, testing and project flows. The automated version control procedures are crucial for easy errorless management.

Keywords
Agile methods, database, procedures, software development, version control, web development

Miscellaneous

SISÄLTÖ

KUVIOT

1 TARKOITUS JA RAJAUS

1.1 Opinnäytetyön tilaaja

Protacon Solutions Oy on Protacon Group Oy -konserniin kuuluva jyväskyläläinen asiakaskohtaisiin ohjelmistoihin ja ratkaisuihin keskittyvä ohjelmisto ja ICT- yritys. Yritys käyttää kehitysprosesseissa ketteriä menetelmiä ja ajantasaisia työkaluja. Verkko-ohjelmistoihin keskittyvän ohjelmistotarjonnan lisäksi Protacon Solutions tarjoaa myös ylläpito- ja käyttöpalveluita. Yrityksessä työskentelee 60 henkilöä pääasiassa Jyväskylän toimistolla. (Protacon Solutions Oy 2012.)

1.2 Tavoitteet

Tavoitteena oli tutustua, tutkia ja löytää verkko-ohjelmistojen ketterään tietokantojen kehitykseen sopivia käytäntöjä ja prosesseja, jotka helpottavat kehittäjien toimintaa. Sopivien prosessien puuttuminen aiheuttaa sen, että jokaisessa projektissa asiat tehdään erilailla, ja jopa projektin sisällä eri henkilöt eivät toimi samoin. Tällöin aikaa tuhlaantuu tiedon kadotessa ja jokainen muutos on niin sanotusti yhden henkilön takana. Yhteisillä sovituilla prosesseilla kehittäjien ei tarvitse miettiä, miten asiat tehdään.

Jotta tietokantojen kehitys kävisi mahdollisimman luontevasti, yhden käyttäjän muutokset eivät saa vaikuttaa muihin, ennen kuin ne on otettu käyttöön. Muutosten tulisi myös olla helposti jaettavissa, toistettavissa ja tarpeen tullen myös peruttavissa. Opinnäytetyössä pyrittiin löytämään tapoja tämän prosessin toteuttamiseen ja tutkimaan mahdollisuuksia prosessin automatisointiin, jolloin kehittäjälle jäisi enemmän aikaa itse kehittämiseen.

Tämä opinnäytetyö keskittyy verkko-ohjelmistoprojekteihin, joiden projektiryhmä koostuu noin kolmesta vakituisesta henkilöstä ja tietokanta on luotu yhden sovelluksen käyttöön.

1.3 Verkko-ohjelmistokehitys ryhmässä

Ohjelmistokehityksessä, oli projekti mikä tahansa, työskennellään normaalisti erikokoisissa ryhmissä. Suuremmissa projekteissa voi työskennellä useita erikokoisia ryhmiä eri osioiden tai projektiin liittyvien kokonaisuuksien kanssa. Yksinkertaisemman

projektin tai palvelun kanssa työskentelee yleensä kahdesta neljään henkilöä. Itse projektiryhmien lisäksi projekteihin liittyy useita sidoshenkilöitä, tai ryhmiä, kuten tarvittavien palvelimien hallinnoijat, asiakkaan ja mahdollisten ulkoisten liitosten yhteyshenkilöt, johto yms.

Tietokannan kehitys ei myöskään rajaudu pelkästään ohjelmistoa kehitysvaiheessa kehittävän ryhmän harteille. Jos projekti on laaja, samaa kantaa voi kehittää usea eri ryhmä. Ohjelmiston jatkokehityksessä voi olla mukana täysin eri henkilöitä, jotka eivät ole olleet mukana aiemmin. Tämä tekee metatiedoista, versioinnista ja muutosten hallinnasta arvokasta projektin sulavan kehityksen kannalta.

Ketterät menetelmät tekevät kehityksestä ryhmässä monipuolista, kun henkilöitä ei suoraan ole sidottu tiettyihin tehtäviin, vaan kukin tekee osaamisensa mukaan. Jotta ketterät menetelmät toimivat käytännössä, tarvitaan prosesseja kehitykseen. Prosessien tulee olla tarpeeksi joustavia, jottei kehittäjien tarvitse käyttää turhaa aikaa prosesseihin, vaan prosessit voidaan muokata projektiin ja kehittäjille sopiviksi.

2 KETTERÄ VERKKO-OHJELMISTOKEHITYS

2.1 Yleistä

Ketterä ohjelmistokehitys on ylätermi eri ohjelmistotuotannossa käytettäville menetelmille, jotka perustuvat toimivan ohjelmiston korostamiseen, suoraan viestintään ja nopeaan muutoksiin reagointiin.

Verkko-ohjelmistona voi pitää verkossa selaimen kautta käytettyä palvelua tai ohjelmaa, joka tarjoaa vuorovaikutteisuutta käyttäjän kanssa. Tällainen ohjelma voi esimerkiksi korvata perinteisen työpöytäsovelluksen tai tarjota dynaamista sisältöä.

2.2 Ketterät menetelmät

Ketterät menetelmät pilkkovat ohjelmistokehityksen lyhyisiin iteraatioihin, joissa jokaisen iteraation jälkeen tulisi olla käytettävissä toimiva versio ohjelmasta. Iteraatiot kestävät normaalisti yhdestä neljään viikkoa, ja jokainen sisältää uusien ominaisuuksien julkaisuun tarvittavat vaiheet: määrittelyn, suunnittelun, ohjelmoinnin, testauksen ja dokumentoinnin. Iteraatioiden välissä käydään läpi projektin tilanne ja tavoitteet, arvioidaan uudelleen prioriteetit ja päätetään seuraavan iteraation sisältö. (Agile software development 2012.)

Ketterissä menetelmissä työskennellään tiimeissä, joiden henkilöt pyrkivät työskentelemään samassa tilassa menetelmien suosiessa suoraa viestintää. Projektitiimissä on kehitystiimiin kuuluvien koodaajien lisäksi kaikki henkilöt, joita ohjelmiston valmiiksi saattamiseen tarvitaan. Tämän vuoksi myös asiakkaan henkilöitä toimii projektitiimissä toimeksiantajan roolissa. Usein nämä jäsenet ovat löyhemmin mukana itse tiimin työskentelyssä, eivätkä henkilöt ole välttämättä samassa tilassa. Näin kuitenkin pyritään pitämään kaikki tarvittavat henkilöt mukana projektin jokaisessa vaiheessa.

Näitä tapoja käyttäen ketterät menetelmät pyrkivät päätavoitteenaan luomaan toimivan ohjelmiston. Dokumentointia ei tehdä pelkästään dokumentoinnin vuoksi, vaan normaalina sivutuotteena tarpeeseen. Toimivaan ohjelmistoon pyritään pitämällä mahdollisimman alusta iteraatioiden tuotokset julkaisukelpoisena ohjelmistona. Muutoksia ei nähdä negatiivisena asiana, vaan ohjelman asiakkaalle tuoman arvon parantamisena.

Loppujen lopuksi ketterät menetelmät ovat enemmänkin ajatusmaailma kuin valmiita työskentelymenelmiä.

2.3 Työskentely ryhmässä

Ketterien menetelmien mukaan ryhmätyöskentely voi oikein toteutettuna nostaa jokaisen yksilön työtehoa ja motivaatiota verrattuna yksin tehtyyn työhön. Tämän saavuttamiseksi tulee kuitenkin sopia ja sovittaa toimintatavat ryhmään ja projektiin sopiviksi. Ryhmä koostuu jäsenistä ja heidän osaamisestaan. Kun ryhmän koko pidetään tarpeeksi pienenä, ketteryys ja kommunikaatio pysyvät hyvinä. Ketterät menetelmät panostavat joustavuuteen, eivätkä määrittele tarkkoja tekotapoja, jolloin ryhmä pystyy itse luomaan tilanteeseen sopivat menetelmät, joita tarvittaessa vaihdetaan iteraatioiden välillä. Tietynlaiset prosessit on kuitenkin hyvä olla valmiina, jottei näitä tarvitse aina uudelleen miettiä ja jotta eri projektit ja ryhmät olisivat yhteensopivia keskenään. Näin kehittäjän ei tarvitse aina vaihtaa ja opiskella perustoimintojen tekemistä uudelleen jokaisen projektin yhteydessä.

Toimeksiantajalla ohjelmiston kehittämiseen ryhmässä käytetään versionhallintaratkaisuja ohjelman koodille. Kehittäjä tekee muutokset koodiin omassa kehitysympäristössään, jossa he voivat turvallisesti testata ohjelmistoa, muita kehittäjiä haittaamatta. Testatut muutokset viedään keskitettyyn varastoon. Täältä muut kehittäjät voivat tuoda muutokset omiin kehitysympäristöihinsä. Ryhmän jäsenten tulee pystyä kehittämään ja testaamaan samaa sovellusta muihin tahattomasti vaikuttamatta. Tämä pätee myös tietokannan kehittämiseen.

2.4 Tietokannan kehitys

Verkko-ohjelmistojen tietokannoille on yleensä yhteistä, että ne on suunniteltu vain yhtä ohjelmaa silmällä pitäen, eikä kannalle yleensä tule muita käyttäviä sovelluksia. Tämä antaa tiettyjä vapauksia kannan suunnittelemisen kanssa, muttei saa olla tekosyy huonoon kantasuunnitteluun.

Ohjelmiston ja tietokannan kehitys on vanhastaan mielletty erillisiksi prosesseiksi, joista vastaavat eri henkilöt, joilla ei ole yhteistä kieltä tai tarkkaa tuntemusta toistensa alueista projektissa. Kaikki kantaan liittyvä on pitänyt erikseen viedä tietokannasta vastaavan henkilön, DBA:n kautta. Lisäksi kannan kehityksessä on usein edel-

leen käytössä tapa, jossa kanta suunnitellaan ensin ja ohjelmisto liitetään tähän mahdollisimman vähällä kannan rakenteen muuttamisella. Kannan kehitystä ei siis ole pidetty ketteränä, ja mallin tarkka miettiminen alussa aiheuttaa muutosvastaisen ajatusmaailman. (Ambler 2003.)

Verkko-ohjelmistokehityksessä kannan kehittämisestä vastaavat usein samat henkilöt, jotka kehittävät ohjelmaa. Tämä helpottaa kannan ja ohjelmiston välisten yhteyksien ja vaatimusten ymmärtämistä, mutta vaatii kehittäjiltä taitoa tietokannan suunnitteluun. Ketterässä kannan kehityksessä kannalle suunnitellaan alussa perusrakenne, mutta tarkemmat taulujen sisällön ja relaatioiden määrittely jätetään iteraatioihin. Usein projektin jäsenillä ei vielä projektin alkuvaiheessa ole tarkkaa kuvaa siitä, millaista kantarakennetta ohjelmisto tulee tarvitsemaan. Kannan rakenne ei myöskään saa jarruttaa muutoksen hyväksymistä ohjelmiston kehittämisessä käytettäessä ketteriä menetelmiä. Jo olemassa olevien kantarakenteiden kanssa kehittäjien pitää asennoitua hyväksymään muutokset, kun niihin on tarvetta. Tärkeintä on muistaa soveltaa ketterän ohjelmistokehityksen ajatusmaailmaa myös tietokantaan.

2.4.1 Kehitysprosessit

Yleisellä tasolla projektin kannat voidaan jakaa kehitys, testaus ja tuotantokantoihin, joissa muutokset kulkevat kehityskannasta kohti tuotantoa. Todellinen kantojen määrä ja muutosten kulku on hieman monimutkaisempi. (Ambler 2003.)

Ketterään kehitykseen kuuluu, että kehittäjä voi rauhassa kokeilla muutoksiaan, ilman että se vaikuttaa muiden kehitykseen. Tämän mahdollistaa oma kehitysympäristö jokaiselle kehittäjälle, jossa kehittäjällä tulee olla oma kanta, johon kehittäjällä on täydet oikeudet. Kantojen luomisprosessista tulee tehdä tarpeeksi yksinkertainen ja automatisoitu, jotta jokaiselle kehittäjälle, jokaiseen projektiin on helppo saada oma kanta tarvittaessa. Kätevää on, jos kehittäjä voi itse luoda kannan ilman suurempaa tuntemusta tietokannan hallintajärjestelmästä. Tällöin kannassa olisi valmiiksi hoidettu käyttäjiin ja oikeuksiin liittyvät asiat ja käyttäjällä olisi heti pääsy kantaan. Vastaavasti kehittäjän tulisi olla mahdollista myös poistaa vanha turhaksi käynyt kanta. Kun näihin operaatioihin on luotu automatisoitu prosessi, myös tietokannan hallintajärjestelmästä vastaavan henkilön resursseja säästyy. (Sadalage 2007, 7-9.)

Jotta jokaisella oleva oma tietokanta toimii, kannan yhteysasetuksien tulee olla helposti muokattavissa erikseen jokaisen kehittäjän työskentely-ympäristöön. Aiemmin ajatuksena oli että omista kehityskannoista viedään muutokset keskitettyyn kantaan, josta muut kehittäjät voivat päivittää omat kantansa, kuten koodin versionhallintaratkaisuissa. Tämä ratkaisu toimii erillään kehitettävien tietokantojen kohdalla. Myöhemmin kannan muutoksien versionhallinta on liitetty yhteen koodin versionhallinnan kanssa kun on kyse tietyn ohjelman kannasta. Jotta tällainen muutosten siirto on mielekästä, se tulee automatisoida. (Fowler 2003, 2006.)

Kantaan voi tehdä kahdenlaisia muutoksia. Muutoksia, kuten sarakkeen lisääminen tauluun, jotka eivät riko olemassa olevaa koodia ja muutoksia, jotka rikkovat. Ensimmäiset muutokset voidaan viedä keskitettyyn kantaan ilman riippuvuuksia koodiin, mutta jälkimmäisissä muutoksissa tulee ottaa huomioon myös koodimuutokset. Mitä väljemmin ohjelmakoodi ja kanta ovat sidoksissa toisiinsa, sitä helpompi kumpaankin on tehdä muutoksia. Väljyyttä voi saavuttaa kannan enkapsulaatiolla ja koodin data-layerilla. Enkapsuloimalla kannan tiettyjä toimintoja ja tauluja piilotetaan rajapintojen taakse. Data-layer auttaa siirtämällä kannan kanssa kommunikoivan koodin yhteen paikkaan, jolloin muualta koodista voidaan kutsua tätä kerrosta tietyillä komennoilla ilman tarkkaa tietoa kannasta. Kummassakin tapauksessa pitää kannan rakennetta muutettaessa muistaa muuttaa kannan kanssa kommunikoivaa tasoa. Nämä tavat eivät poista tarvetta muokata koodia, mutta ne helpottavat muutosta. Suurempien rikkovien muutosten, jotka koskevat paljon käytettyjä tauluja, kohdalla voi olla hyvä jättää muutos iteraation alkuun. (Ambler 2003.)

Kun kehittäjä tekee muutoksia tietokantaan, on mahdollista että hän tarkoittamattaan rikkoo jonkin ohjelman ominaisuuden toiminnan. Tämän vuoksi eri kehittäjien koodin ja kannan muutosten sekä ohjelman osien jatkuva yhdistäminen on tärkeää. Martin Fowler (2006) puoltaa jatkuvaa integrointia julkaisussaan Continuous Integration. Hän ehdottaa automaattisten integraatiopalvelimien käyttöä. Kun versionhallintaan tulee muutoksia, palvelin luo päivitetyn toimivan ohjelmiston ja ajaa tälle määrättyjä toimenpiteitä, kuten automaattitestejä. Kun testi luo versiota vastaavan kannan ja testaa ohjelman toiminnallisuutta sitä vastaan, saadaan tietoa mahdollisista ongelmista. Tämä myös helpottaa kehittäjän työtä, kun hänen ei tarvitse testata

kaikkea. Tärkeät tietokantamuutokset tulee kuitenkin ensin varmistaa kannasta tai toiminnoista vastaavilla henkilöillä, jottei turhia virheitä synny. (Fowler 2006; Sadalage 2007, 28.)

2.4.2 Versionhallinta

Kantojen versionhallinta toimii ajatukseltaan samoin kuin keskitetyn versionhallinnan koodit. Kehityskantojen muutokset viedään keskitettyyn varastoon, jonka kautta muiden muutokset jaetaan takaisin kehittäjille. Tällä tavalla jokaisella kehittäjällä on ajantasainen kanta. Keskitetystä versionhallinnasta muutokset viedään kerralla isommissa erissä testaukseen ja tuotantoon projektin julkaisutavan mukaan esimerkiksi iteraation tai jokaisen muutoksen jälkeen. Myös kannasta tehdään versiohaaroja julkaisuja varten. Erona koodin haaroihin kannan voi haarauttaa myös datasisällön mukaan. Tällainen tilanne on testikannan kanssa, jossa kannassa on hyvä olla tarpeeksi dataa kattavaa testausta varten. Tärkeä ero koodin versiohallintaan nähden on tietokannan sisältämä data. Muuttuvaa, ohjelman käytössä syntyvää dataa ei voi tallentaa versionhallintaan, mutta sen pysyvyys muutoksien yhteydessä täytyy ottaa huomioon tuotantoympäristössä. Ohjelman käyttämiseen tarvittava data sen sijaan pitää viedä versionhallintaan. (Fowler & Sadalage 2003; Hunt 2011.)

Martin Fowler ja Pramod Sadalage (2003) ehdottavat julkaisussaan Evolutionary Database Design automaattista päivitystä kehityskannoille keskitetyn kannan päivittyessä. Tämä eroaa perinteisestä koodin versiohallinnasta, mutta on heidän mukaansa toimiva ratkaisu. He eivät kerro, miten mahdolliset konfliktit ilmaistaan näissä tapauksissa. (Fowler & Sadalage 2003.)

Käytännössä kannan muutokset tallennetaan skriptitiedostoihin, joilla muutoksen voi ajaa tai mahdollisesti myös peruuttaa. Näiden tiedostojen käsittelyyn ja ajamiseen luodaan joko itse automaattinen ratkaisu tai käytetään olemassa olevaa. Tiedostot itsessään toimivat kannan muutoslokina, mutta näihin tai näiden lisäksi voidaan tuottaa muutakin metadataa.

Ohjelmakoodi ja tietokanta ovat toisiinsa sidottuja. Kummankin pitää olla tietyssä versiossa ohjelman toimimista varten. Jotta tietokannasta voidaan saada ohjelman versiota vastaava versio, kannan luontiskriptit pitää sisällyttää samaan versionhallin-

taan koodin kanssa. Kanta luodaan alusta näiden skriptien mukaan. Jos kannasta löytyy edellinen versio, se tuhotaan ensin. Tarkoituksena on yleensä uuden ohjelmiston pystyttäminen, kehitys- tai testiversio, joten vanhaa kantaa ei tarvita. Jos kanta on jo olemassa, sen nykyisestä tilasta tulee luoda nämä skriptit. Tuleva kehitys ja muutokset tehdään aina ensin skripteihin. Skriptien hallinnan helpottamiseksi erityyppiset tietokantaobjektit tulee tallentaa omiin tiedostoihinsa, esimerkiksi taulujen luonti, indeksien luonti, stored prosedures ja datan lisäys. (Sadalage 2007, 7-9.) Muiden muassa Mike Hadlow (2006) suosittelee blogissaan julkaisemassaan artikkelissa How to do database source control and version controls luomaan jokaiselle objektille oman tiedostonsa, jolloin objekti on helpompi löytää ja sen versiohistoriaa helpompi seurata (Hadlow 2006).

Tietokannan skeeman luonti tiettyyn versioon on hyvä automatisoida ajettavan komennon taakse (Fowler 2003). Tällöin kehittäjän ei tarvitse käsin ajaa skriptejä, jolloin prosessi nopeutuu ja virhemahdollisuus pienenee. Kannan päivittämiseen on kaksi lähtökohtaista tapaa. Ensimmäisessä olemassa oleva kanta tuhotaan ja luodaan tietyn version skripteistä uudelleen. Tätä tapaa käytetään yleensä uutta ympäristöä luotaessa tai kehitys- ja testiympäristöissä, joissa datan säilyvyydellä ei ole merkitystä. Toisessa tavassa olemassa olevaa kantaa päivitetään muutostiedostojen mukaan eteen- tai tarvittaessa taaksepäin. Kannan versio tai päivityshistoria tulee tallentaa kantaan, jotta kannan kulloinenkin tila tunnetaan ennen päivittämistä. Tätä tapaa käytetään ympäristöissä, joissa datan säilyvyys on tärkeää. Yleensä tämä on tuotantoympäristö, mutta tämä tapa sopii myös muihin ympäristöihin. Toisen tavan kanssa täytyy ottaa huomioon kehitysympäristössä, että myös aiemmista kokeiluista unohtuneet muutokset pysyvät kannassa ja voivat aiheuttaa odottamattomia ongelmia. Scott Allen (2008) pitää säännön arvoisena että ratkaisusta riippumatta on tärkeää kehittäjälle tietää mistä voi saada nykyisen tietokantarakenteen, ja pystyä ottamaan sen itselleen helposti käyttöön myös kesken projektiin tullessaan. (Fowler 2003; Allen 2008.)

Skriptien, metatiedon ja versionhallinnan rakenne vaihtelee automatisoiduista ratkaisuista riippuen. Näitä tutkitaan tarkemmin luvussa 3.

2.4.3 Ryhmän jäsenten roolit

Perinteisesti ohjelmiston kehittäjät ja tietokantaa hoitava DBA ovat olleet erillisiä henkilöitä, jotka ovat toistensa kanssa tekemisissä vain tarvittaessa. Ketterä kehitystiimi ei välttämättä tarvitse erillistä DBA:ta, vaan roolissa voi toimia kykenevä tiimin jäsen tai jäsenet. Tämä helpottaa muutosten tekemistä ja varmistaa, että kannasta vastaavalla on kattava tieto projektista. Tärkeintä kantamuutoksissa on kehittäjien ja DBA:n kommunikaatio. (Fowler 2003.)

Kannan tarkemman rakenteen muodostuessa ja muuttuessa kehityksen aikana kehityksestä vastaavat pääasiassa ohjelmistokehittäjät. Kaikilla ei kuitenkaan tarvitse olla täyttä tietämystä kannan toiminnasta tai kehittämisestä, kun DBA voi hoitaa nämä muiden tiimin jäsenten puolesta. Kehittäjillä tulisi tästä huolimatta olla perustiedot kannasta ja sen kehittämisestä, jotta he osaavat ottaa nämä asiat huomioon ja kommunikoida tarpeensa. Sama pätee myös erillisen DBA:n tapauksessa ohjelmiston ja peruskoodauksen tuntemiseen. Varsinkin suuremmissa tai olemassa olevaa ohjelmiston toimintaa muuttavissa muutoksissa muutokset tulee käyttää DBA:n kautta. DBA voi tuoda esiin erillisen näkökulman tai mahdolliset konfliktit muutoksessa. (Fowler 2003.)

Ketterässä kehityksessä erillisen DBA:n toimintaan ei niinkään kuulu kannan skeeman tai datan ylläpito, vaan toiminta kantojen hallitsijana, joka pitää huolen hallintajärjestelmästä ja ympäristöstä. DBA:n tehtävänä on pystyttää, päivittää ja pitää huoli, että nämä pysyvät käynnissä. Tämän henkilön takana ovat myös pääsy kantaan ja muut yleisemmät oikeudet. Kun prosessit ovat selkeät ja tarpeellinen automaatio monimutkaisille ja toistettaville tehtäville luotu, myös kesken projektiin tulleet tiiminjäsenet pääsevät mukaan huomattavasti helpommin (Fowler 2003; Sadalage 2007, 27).

2.5 Jatkuva integraatio

Edellä esiteltyjä ketterien menetelmien versionhallintaa, prosesseja ja testausta varten projektin tasolla on kehitetty jatkuvan integraation periaate. Pääpiirteissään tämä tarkoittaa aina kehittäjien tekemien muutosten jälkeen kaikkien ohjelman osien yhdistämistä valmiiksi ohjelmaksi, joka voidaan testata ja periaatteessa vaikka julkaista. Näin toteutetaan ketterien menetelmien periaatetta julkaisukelpoisesta oh-

jelmistosta ja ennen kaikkea testataan ohjelman osat ja niiden yhteensopivuus mahdollisimman nopeasti muutosten tekemisen jälkeen. Jatkuvaan integraatio toteutetaan yleensä erillisillä integraatiopalvelimilla ja tähän tarkoitukseen on useita ohjelmia hoitamaan integraation vaatima automaatio. Tietokannan kannalta on hyvä, että integraatiopalvelimet antava antavat luoda tietokannan käyttäjän valitsemalla tavalla. On siis mahdollista käyttää valitsemaansa ratkaisua tietokannan rakentamiseen ja asettaa tämä ratkaisu ajettavaksi integroinnin yhteydessä, jolloin versiohallitut kannat saadaan myös mukaan jatkuvaan integrointiin. Tietokannoille on olemassa myös täysin omia jatkuvan integraation ratkaisuja kuten DB-Ghost tai Red Gaten työkalut, joissa testataan kannan luonti ja toiminta muutosten jälkeen. (Fowler 2006; Wikipedia 2012.)

Jatkuvaan integraatioon kuuluu tiukasti versiohallinta. Martin Fowler pitää parhaana pitää versionhallinnassa kaikki ohjelmiston rakentamiseen tarvittava. Kun kehittäjä on lähettänyt muutoksensa versionhallintaan, integraatiopalvelin tekee joko heti tai ajastetusti ohjelmasta käytettävän version. Tämän jälkeen ohjelmalle ajetaan automaattitestit. Mahdollisimman kattavat integraatiotestit ohjelmalle testaavat samalla tietokantaa ohjelman toimintojen kautta. Testauksessa voi olla montakin vaihetta, mutta tärkeää on testata ohjelmaa mahdollisimman paljon julkaisuympäristöä vastaavassa ympäristössä. Läpäissyt ohjelmaversio voidaan asettaa käytettäväksi, jotta myös ihmiset voivat testata sitä tai käyttää esittämään ohjelman toimintoja. Integroinnissa pitää pyrkiä nopeaan aloittamiseen ja toimivan ohjelman rakentamiseen muutoksien jälkeen, jotta tieto virheistä saadaan heti kehittäjille. (Fowler 2006.)

3 TIETOKANTOJEN VERSIONHALLINTARATKAISUT

3.1 Yleistä

Ketterän tietokantojen kehityksen levitessä on kehittynyt tapoja ja työkaluja versionhallinnan helpottamiseksi. Iso tekijä versionhallinnassa on helppouden tavoittelu, joka saavutetaan viemällä automatisointi mahdollisimman pitkälle. Valitettavasti tietokannan versionhallinta ei ole yhtä yksinkertaista kuin koodilla. Tuotannossa tulee ottaa huomioon datan säilyminen muutoksien yhteydessä. Erilaisia versionhallintaratkaisuja on käsitelty luvuissa 3.2 – 3.6.

3.2 Tietokantojen vertailuohjelmat

Kehittäjälle helpompi ratkaisu olisi kantamuutosten jälkeen käyttää kantoja vertailevaa ohjelmaa muutosskriptin luontiin. Tällä muutosskriptillä kannat saadaan yhdenmukaisiksi. Kehittäjän ei siis tarvitse erikseen kirjoittaa skriptejä, vaan hän voi haluamallaan tavalla muokata tietokantaa. Tällaisessa menettelyssä on kuitenkin tiettyjä huonoja puolia. Luvussa 4.4 tarkemmin läpi käydyn Liquibasen kehittäjät ovat käsitelleet näitä puolia artikkelissaan The Problem With Database Diffs. Heidän mukaansa tietokantojen eroja tarkkailevien ohjelmien huonot puolet ylittävät niistä saadut edut. Pääasiallinen etu on yksinkertaisuus käyttäjälle. Kehittäjä tekee tarvittavat muutokset tietokantaan ja muodostaa vertailuohjelmalla muutosskriptin keskitettyyn kantaan tai skripteihin vertailemalla. Vertailun ongelmat ovat muutoksien taustojen katoaminen ja datan vertailu. Vertailevat ohjelmat eivät pysty selvittämään, onko esimeriksi sarake korvattu toisella, vai onko se vain uudelleen nimetty kadottaen näin sarakkeen datan. Datan vertailu ei taas ole monessa tilanteessa järkevää mahdollisen testidatan ja tehokkuuden takia. (The Problem With Database Diffs 2007.)

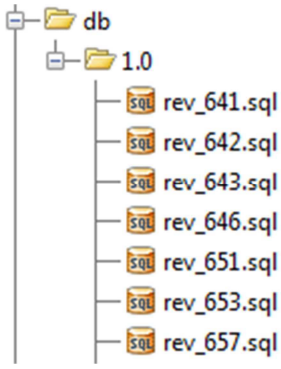
Ongelmista huolimatta vertailuohjelmia käytetään laajalti. Usein esille tuotu ongelma uudelleennimeämisen kanssa kierretään vaatimalla käyttäjää puuttumaan operaatioon. Käyttäjän pitää joko käyttää ohjelman omaa uudelleennimeämistä, tai luoda erikseen ajettava skripti tätä varten. Käyttöliittymän omaavalla vertailuohjelmalla pystyy näkemään nopeasti omat ja muiden muutokset, varmistamaan että ne ovat ajettavissa ja ajaa muutokset kantaan. Innovartiksen SQL Serverille kehittämä DB Ghost tarjoaa kattavaa vertailuun perustuvaa automaatiota. Käyttäjä pystyy Ghostilla

luomaan olemassa olevasta kannasta tietokantaobjektien mukaiset skriptitiedostot jotka asetetaan versionhallintaan. Kehittäjä muokkaa näitä tiedostoja ja ajaa niiden muutokset tietokantaansa Ghostia käyttäen. Ghostin idea on kuitenkin käyttää sitä myös jatkuvan integraation osana. Myös Red Gaten ratkaisut SQL Compare ja Data compare SQL Serverille tarjoavat samankaltaiset visuaaliset työkalut tietokannan muutosten hallintaan ja toimivat osana laajempaa tuoteperhettä jatkuvan integraation välineinä. Kehittäjä voi vertailla tietokantaskeemoja ja hallita muutosten synkronointia käyttöliittymän avulla.

3.3 Oma ratkaisu

Tietokannan versionhallintaan ei välttämättä tarvita erillistä valmista ratkaisua. Alkuun pääsee yksinkertaiselle itse tehdyllä muutosskriptien hallinnan automaatiolla. Myös monet valmiit ratkaisut perustuvat tälle pohjalle.

Harrie Verveer (2011) esittää Techportalin verkkojulkaisussa Database Version Control esimerkin tyypillisestä tavasta hoitaa tietokannan versionhallinta. Tietokantaan ajettavat muutosskriptit, kuten taulun luonti, muokkaus tai rivin lisäys, tallennetaan scriptitiedostoihin omaan kansioonsa, jotka lisätään koodin versionhallintaan normaalin ohjelmistokoodin mukana. Näin muut kehittäjät saavat tietokantamuutokset koodin päivityksen mukana ja voivat ajaa ne omaan kantaansa. Tiedostojen tulee noudattaa juoksevaa nimeämiskäytäntöä, jotta muutokset ajetaan oikeassa järjestyksessä. (Ks. Kuvio 1.) Muutoksia tietokantaan ajettaessa, tiedossa pitää olla myös kannan nykyinen versio, jotta vain uudet muutokset osataan ottaa huomioon. (Verveer 2011.)



KUVIO 1. Koodin version mukaan nimettyjä muutostiedostoja

Kannan päivitys voidaan toteuttaa automaattisesti luomalla skripti, joka ajaa nämä muutostiedostot määrättyyn kantaan. Tällöin kehittäjän täytyy vain ajaa skripti koodin päivityksen jälkeen ja kanta päivittyy ajan tasalle. Kannan versiotieto voidaan tallentaa itse kantaan, josta skripti voi tarkistaa, mitkä muutokset ajetaan kantaan. Versiotiedon täytyy sisältää viite viimeisimpään ajettuun päivitykseen. Tämän lisäksi voidaan pitää tarkempaa muutoshistoriaa ajetuista muutoksista. (Mt.)

Ratkaisun luominen menee monimutkaisemmaksi, jos otetaan huomioon tarve peruuttaa muutoksia, mahdollisuus päivittää kanta tiettyyn vanhempaan versioon. Tätä varten tarvitaan peruutusskriptit jokaista muutostiedostoa kohden. Nämä skriptit tekevät päinvastaisen operaation päivityksen sisältämään tiedostoon nähden. Jos päivityksessä luodaan uusi taulu, peruutustiedostossa poistetaan se. Peruutustiedostojen luominen tuottaa lisää työtä, mutta on tarpeellista, jos halutaan mahdollistaa muutosten peruuttaminen. Yksi mahdollisuus on luoda kanta uudelleen alusta siihen pisteeseen asti, johon halutaan peruuttaa. Tällöin ei tarvita peruutustiedostoja. (Mt.)

Koodista poiketen jotkut kannan versiomuutokset ovat peruuttamattomia. Sarakkeen tai taulun poistaminen tuhoaa sen sisällön. Tämä ei välttämättä ole suuri ongelma kehitysvaiheessa, mutta tuotannon päivittämisessä kannan varmuuskopiot ovat tärkeitä. Suoraan datan kanssa toimiminen ei myöskään ole yksinkertaista. Aiemmin lisätyn datan tarkistaminen uutta lisätessä voi olla erittäin monimutkaista. Poistettua dataa ei välttämättä ole mahdollista palauttaa. (Mt.)

Toinen versiopäivityksessä huomioon otettava asia on kehityshaarojen yhdistäminen. Kun eri koodihaaroihin tehdään tietokantamuutoksia, tulee huomioida, etteivät versiotiedostojen nimet mene päällekkäin ja että muutokset ajetaan oikein kantaan. Nimeämiseen on monia ratkaisuja, kuten omiin haaraa vastaaviin kansioihinsa tallentaminen. Jos koodin versiotiedot pysyvät toisistaan erillisinä haarojen välillä, kuten Subversionissa, voidaan käyttää tätä tietoa tiedoston nimenä. Eräs toinen mahdollisuus on käyttää tarpeeksi tarkkaa aikaleimaa. Haarojen yhdistämisessä muutosten oikea ajo muodostuu ongelmaksi. Eri haaroissa kanta ei välttämättä ole samassa tilassa ennen muutosta. Myös kantaan ajetuista päivityksistä tarvitaan viimeistä ajettua versiota tarkempi tieto, jotta toisesta haarasta tulleet muutokset osataan ajaa.

Käytännössä tällainen yhdistäminen vaatii kehittäjää varmistamaan, ettei konflikteja synny. Kaikki toisen haaran muutokset voidaan myös yhdistää yhteen versiotiedostoon ja jättää huomiotta haarautuminen automaatioskriptissä. (Mt.)

Menetelmän hyvä puoli on toteutustavasta riippuen yksinkertaisuus, räätälöinti omiin prosesseihin ja tietokantamuutosten sitominen koodimuutoksiin. Toteutus ja prosessit pitää päättää erikseen, mikä vaatii oman aikansa. Pelkkä päivittäminen on lopulta yllättävän yksinkertainen toteuttaa, mutta järjestelmästä voi tehdä niin monimutkaisen ja automatisoidun kuin itse haluaa.

3.4 Valmis ratkaisu

3.4.1 Yleistä

Monien valmiiden ratkaisujen toiminta perustuu pitkälti samoihin periaatteisiin kuin oman ratkaisunkin. Jotkut tekevätkin pelkästään skriptitiedostojen ajon kantaan nykyisen version perusteella. Monimutkaisemmissa ei käytetä kannan käyttämää kyselykieltä, vaan ratkaisun omia komentoja muutosten ajamiseen. Tällöin samoja päivitystiedostoja on mahdollista käyttää eri kannanhallintajärjestelmissä ja muutoksiin voidaan liittää lisää logiikkaa, kuten automaattisesti luotavat peruutusskriptit.

3.4.2 Liquibase

Liquibase on kehittynyt Javalla toteutettu valmis ratkaisu tietokannan versionhallintaan. Liquibase toimii esimerkkinä sen monipuolisuuden, helppokäyttöisyyden ja hyvän dokumentaation takia. Samanlaisia toimintaperiaatteita löytyy muistakin ratkaisuista, pienin muutoksin.

Liquibase käyttää muutosten tallentamiseen XML-tiedostoja. Muutokset kirjoitetaan XML-muodossa, jotta Liquibase käsittää muutoksen sisällön ja pystyy myös peruuttamaan monia muutoksia ilman erillisiä peruutusskriptejä. Yksi XML-muutoslista voi sisältää useita muutoksia, joille jokaiselle asetetaan käsin listakohtainen yksilöllinen id ja muutoksen tehnyt kehittäjä. Kuviossa 2 on kuvattu taulun luominen XML-muutoslistaa käyttäen. Nämä yhdistettynä itse tiedostoon yksilöi jokaisen muutoksen ja muodostaa muutoslistan. Muutoslistat tallennetaan muun koodin mukana versionhallintaan, josta ne jaetaan muille käyttäjille. (Liquibase 2012.)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
  xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-2.0.xsd">

    <changeSet id="1" author="ane">
        <createTable tableName="Users">
            <column name="id" type="int">
                <constraints primaryKey="true" nullable="false"/>
            </column>
            <column name="Name" type="varchar(50)">
                <constraints nullable="false"/>
            </column>
            <column name="Email" type="varchar(100)">
                <constraints nullable="false"/>
            </column>
            <column name="active" type="boolean" defaultValueBoolean="true"/>
        </createTable>
    </changeSet>

    <changeSet id="2" author="ane" context="test">
        <insert tableName="Users">
            <column name="id" value="1"/>
            <column name="Name" value="testaaja"/>
            <column name="Email" value="test@example.com"/>
        </insert>
    </changeSet>
</databaseChangeLog>
```

KUVIO 2. Taulun ja testidatan lisäys Liquibasella

Listan muutokset ajetaan kantaan Liquibasella. Tämän prosessin voi liittää useisiin ohjelmistohallintatyökaluihin tai palvelimen käynnistykseen. Muutokset voi ajaa myös suoraan komentorivillä. Liquibase luo kantaan itselleen kaksi taulua, joilla se hallitsee muutoshistoriaa. Taulu databasechangelog pitää sisällään kaikki ajetut muutokset muutoslistoilla. Tämän perusteella Liquibase tietää, mitkä muutokset kantaan on jo ajettu. Taulu databasechangeloglock pitää huolen siitä, että kantaan voi ajaa kerrallaan muutoksia vain yksi Liquibase-instanssi. Muutoslistoja on mahdollista sisällyttää muihin listoihin, jolloin voidaan luoda esimerkiksi lista, joka sisältää kaikki ohjelman julkaisuversiot tai muuten eriyttää listoja sisällön mukaan ja ajaa ne yhtä listaa kutsumalla. (Mt.)

Muutoslistoissa on mahdollista käyttää normaalia SQL:ää Liguidbasen XML-syntaksin sijaan tai ladata ulkoisia SQL-tiedostoja. Liquibasen sisäinen syntaksi ei siis rajoita mahdollisia muutoksia. Stored proseduret ajetaan tätä menetelmään käyttäen. Kehittäjän ei myöskään ole pakollista opiskella Liquibasen XML-syntaksia käyttääkseen sitä kannan versionhallintaan. Huonona puolena kehittäjä menettää automaattisen muutosten peruuttamisen, ellei kehittäjä kirjoita itse peruutusskriptiä. (Mt.)

Muutoslistoihin on mahdollista asettaa ehtoja itse listaan tai yksittäiseen muutokseen. Muutos ajetaan vain, jos ehto on tosi. Ehdoilla voi tarkastella muun muassa kantamoottoria, kannan käyttäjää, muita muutoslistoja, kannan rakennetta tai jopa kannan dataa. Ehtojen lisäksi listoja voi hallita konteksteilla. Muutoksia ajettaessa käyttäjä voi määrittää, minkä kontekstin omaavat muutoslistat ajetaan. Liquibasen dokumentaatiossa suositellaan testeihin liittyvien muutosten ajamista kontekstia käyttäen. Kuviossa 2 lisätään tauluun dataa kontekstin ollessa test. (Mt.)

Muutoslistan aikaisempia muutoksia ei normaalisti tule muuttaa. Jos muutoksen poistaa listalta tai siihen tekee muutoksia, voi seurata ongelmia, jos tulevat muutokset ovat tästä riippuvaisia. Liquibase auttaa huomaamaan virheelliset muutokset aikaisempiin listan muutoksiin tarkastelemalla näitä kannan lokissa oleviin hajautusarvoihin. On mahdollista asettaa muutokselle myös aja muutettaessa -valinta, jolloin kohta päinvastoin ajetaan, kun siihen on tullut muutoksia. (Mt.)

Liquibase mahdollistaa tietokannan merkitsemisen tietyllä versiotekstillä, jolloin esimerkiksi tietokannalle voi määritellä julkaisuversiot ja näihin voi helposti päivittää. Toinen ominaisuus on kahden tietokannan välisten erojen selvittäminen ja muutoslistan luominen, jotta toinen kanta saadaan samaan tilaan kuin toinen. Vertailussa tulee muistaa, että käytössä ei ole muutoshistoriaa, jolloin Liquibase ei voi tietää, onko esimerkiksi sarake nimetty uudelleen vai poistettu ja luotu uusi. Käyttöön ottaessa on kätevää, että Liquibase osaa tehdä olemassa olevasta kannasta muutoslistan, jolla kanta voidaan luoda. Myös JavaDoc-tyylisen muutoshistorian luominen kannan ja muutoslistojen perusteella on mahdollista. Tämä helpottaa muutosten tarkastelemista esimerkiksi virhettä etsittäessä. Stored prosedures, funktiot ja triggerit, luomi-

nen ja muutokset pitää yleensä hoitaa käsin. Myös kannoista automaattisesti luotaviin muutoslistoihin nämä pitää muistaa lisätä erikseen. (Mt.)

Liquibaselle kehitetään graafisia IDE-käyttöliittymiä Eclipselle ja IntelliJ IDEA:lle. Näissä muutosten hallinta on yhdistetty kannan rakenteen graafiseen muokkaamiseen. Suunnitelmissa on tuoda Liquibaseen uusia tietokannanhallintajärjestelmiä, lisää refaktorointitageja ja automaattinen testidatan luonti ja testien hallinta kannalle. Ominaisuuksien listalla on myös tietokantadiagrammin luonti, joka tekisi Liquibasesta entisestään kattavamman ratkaisun tietokannan käsittelyyn. (Mt.)

3.4.3 Dbdeploy

Javalle, .NETille ja PHP:lle julkaistu Dbdeploy on huomattavasti yksinkertaisempi ratkaisu. Se vastaa luvussa 3.3 esiteltyä omaa ratkaisua yksinkertaisimmillaan. Dbdeploylle annetaan ajettavien tiedostojen sijainti ja kanta johon muutokset tehdään. Dbdeploy tekee aiemmin ajamattomissa tiedostoissa olevista kyselyistä yhden skriptin, ajaa sen kantaan ja tallentaa tiedon ajetuista muutoksista omaan tauluunsa. Tämän taulun luomiseen dbdeploy tarjoaa komennon, mutta muita ominaisuuksia siitä ei löydy. Muutostiedostot ajetaan järjestyksessä nimen mukaan. Ratkaisu on erittäin yksinkertainen ja helppokäyttöinen. (dbdeploy 2012.)

3.5 ORM työkaluihin integroidut ratkaisut

Ohjelmakoodin ja tietokannan sidonnaisuutta voidaan vähentää käyttämällä DBAL- ja ORM-menetelmiä. ORM eli Object-relational mapping tarkoittaa tietokannan datan ja rakenteen abstraktointia ohjelmoinnin objektien kautta. Abstraktoinnin toimintaa varten kehittäjä luo tietokannan rakennetta vastaavan rakenteen ORM-järjestelmän ymmärtämällä tavalla, usein ohjelmointikielen mukaisina luokkatiedostoina, tai erikseen määritetyllä merkintäkielellä. Koska kannan rakenne pitää toteuttaa näin, osa ORM-ratkaisuista tarjoaa mahdollisuuden luoda kannan skeema suoraan ORM-kartoituksen pohjalta, säästäen kehittäjän aikaa. Näin tehdessään kehittäjän tulee olla tarkka, sillä ohjelmoinnissa käytettyjen olioiden rakenne ei saa toimia tekosyynä kannan rakenteen suunnittelun laiminlyönnille, kuten Josh Berkus varoittaa luennossaan Ten Ways to Wrek Your Database [Berklus 2009]. Kartoituksen pohjalta ORM luo kannan datasta olioita. Ohjelma käsittelee näitä olioita ja ORM hoitaa niiden ha-

kemisen ja tallentamisen tietokantaan. Käytännössä yhtä kannan riviä vastaa olioinstanssi. ORM poistaa tarpeen SQL-koodin käytölle ohjelman koodissa, jolloin kantamuutokset voidaan hoitaa käsiteltävien olioiden ja niiden käsittelyn muutoksilla. Koodi ei myöskään ole sidottu tiettyyn tietokannanhallintajärjestelmään, vaan ORM:n mahdollistaa kaikkien tukemiensa kantojen käytön. Monissa verkko-ohjelmistokehyksissä on sisäänrakennettu ratkaisu tietokantaoperaatioille, joka käytännössä on yleensä tietyntyyppinen ORM toteutus. (Object-relational mapping 2012.)

Sisäänrakennettuihin tietokantaratkaisuihin on lisätty myös tietokannan päivitysratkaisuja. Toiminta on pääpiirteittäin samanlainen kuin erillisissä toteutuksissa, sisältäen liitoksia käytettyyn tietokantaratkaisuun. Yleensä tämä tarkoittaa, että käsin tehtyjen muutosskriptien lisäksi, kehittäjä voi tehdä muutokset ORM-kartoitukseen ja luoda skriptin näiden muutosten pohjalta. Kuten kartoitus, muutokset on monessa tapauksessa tallennettu käytetyn kielen mukaiseksi luokaksi, tai merkintäkielellä joka yleensä käännetään luokaksi. Luokka sisältää metodit päivitykseen ja sen peruuttamiseen. Näihin metodeihin toteutetaan ratkaisusta riippuvat kutsut, joilla tehdään muutokset. Kuten erillisratkaisuissa, myös näissä on tavallista antaa mahdollisuus ajaa suoraan SQL-kyselyitä täydentämään toiminnallisuuksia. Suorat kyselyt sitovat muutoksen tiettyyn kantaan, mutta mahdollistavat kannanhallintajärjestelmille yksilöllisien toimintojen käyttämisen. (Doctrine 2 Migrations documentation 2012; Ruby on Rails Migrations 2012.)

Doctrine on ORM:iin ja DBAL:iin keskittyvä PHP-kirjastojoukko. Muutostenhallintaan se tarjoaa Migrationsin osana tietokantakirjastoaan, tai erikseen asennettavana ratkaisuna, jolloin komentoja on helppo ajaa palvelinympäristöissä. Muutokset kirjoitetaan php-luokkina, tai muodostetaan kannan kartoitukseen tehtyjen muutosten pohjalta. Kuviossa 3 lisätään uusi taulu tietokantaan Doctorinea käyttäen. Muutokset voi ajaa suoraan kantaan tai tallentaa SQL-tiedostoksi. Muutokset pystyy myös peruuttamaan. Doctrine pitää kirjaa ajetuista muutoksista kannassa olevassa taulussa. Doctrine 2 migrationsin verkkodokumentaatio on kevyt ja käsittelee yleisen käytön. Vanhemmassa dokumentaatiossa on enemmän tietoa käytöstä. Doctrineä käytetään muun muassa Symfony – ohjelmistokehyksessä. Tämän lisäksi PHP:lle löytyy migra-

tions - kirjastoja muistakin ohjelmistokehyksistä kuten Yiistä. Pääpiirteittäin käyttö on samanlaista. Muutosluokasta löytyy up- ja down -metodit, mutta funktiot muutosten tekemiseen ja komennot ajamiseen ovat erilaiset. (Doctrine 2 2012; Symfony 2012; Yii 2012.)

```php
namespace DoctrineMigrations;

use Doctrine\DBAL\Migrations\AbstractMigration,
    Doctrine\DBAL\Schema\Schema;

class Version20121208174411 extends AbstractMigration
{
    public function up(Schema $schema)
    {
        $table = $schema->createTable('Users');
        $table->addColumn('Name', 'string');
        $table->addColumn('Email', 'string');
    }

    public function down(Schema $schema)
    {
        $schema->dropTable('Users');
    }
}
```

KUVIO 3. Taulun lisäys Doctrine Migrationsilla

Ruby on Rails – verkko-ohjelmistokehyksessä on vastaava toteutus samalla nimellä. Tämä Migrations nopeuttaa muutosluokkien luontia, tarjoamalla generaattorin, jolle annetaan taulun rakenne parametreina. Generaattori luo parametrien perusteella luokan, jota voi käyttää kannan päivittämiseen. Railsin Migrationissa on mahdollista käyttää up- ja down - metodeita, joihin muutos ja sen peruutus luodaan, tai change-metodia, joka osaa itse tehdä peruutuksen. Kuviota 3 vastaava taulu lisätään tietokantaan käyttäen change – metodia kuviossa 4. Taululle luodaan myös lisäys ja muutosajan sarakkeet käyttäen Migrationsin timestamps – komentoa. Foreign keyt, triggerit ja tallennetut proseduurit eivät ole tuettuja Migrationin komennoissa, vaan ne pitää erikseen määritellä SQL-lauseilla. Tällä on haettu logiikan pitämistä ohjelmoin-

tipuolella, jolloin kanta pysyisi pelkkänä datavarastona, ilman erikoisempia rajoituksia, tai logiikkaa. (Ruby on Rails Migrations 2012.)

```
class CreateUsers < ActiveRecord::Migration
    def change
        create_table :Users do |t|
            t.string :Name
            t.string :Email

            t.timestamps
        end
    end
end
```

KUVIO 4. Taulun lisäys Ruby on Rails Migrationsilla

3.6 Tietokannan kehitystyökaluihin liitetyt ratkaisut

Tietokannan kehitystyökaluilla kehittäjä pystyy suunnittelemaan ja hallitsemaan tietokannan skeemaa visuaalisella käyttöliittymällä. Varsinkin kaupallisia tietokannanhallintajärjestelmiä kehittävät yritykset kuten Microsoft ja Oracle tarjoavat omia visuaalisia kehitys- ja hallintatyökaluja tietokannoilleen. Niin maksullisille kuin ilmaisille tietokannoille löytyy kuitenkin työkaluja kannan suunnitteluun ja kehittämiseen.

Microsoft tarjoaa tietokannanhallintajärjestelmän nimeltä SQL Server, johon Visual Studiolla, tai SQL Server Developer Toolsilla on mahdollista luoda tietokantaprojekti, jossa kehittäjä voi hallita tietokantaobjekteja visuaalisesti. Visual Studio mahdollistaa synkronoinnin tietokannan kanssa, joten muutokset voidaan ajaa kantaan suoraan ohjelmasta. Microsoft suosittelee käyttämään tuotantoympäristön päivitykseen muutoksista generoitua skriptiä, jonka kehittäjä käy läpi, suoran synkronoinnin sijaan. Versionhallinnan voi toteuttaa samaan tapaan kuin koodilla, asettamalla tietokantaprojekti versionhallintaan.

Kehitystyökaluihin on myös mahdollista liittää versionhallintaratkaisuja. Red Gate on kehittänyt SQL Server Management Studioon liitettävän työkalun SQL Source Cont-

rol. Tällä saadaan tietokannan kehitykseen versionhallintalähtöisyys alusta asti kun Source Control yhdistää muokattavat skeematiedostot versionhallintaan ja tarjoaa näille muun muassa mahdollisuuden konfliktien ratkaisuun. Red Gaten vertailutyökaluihin kuuluvat SQL Compare ja SQL Data Compare hoitavat tietokantojen päivittämisen skeematiedostojen mukaan. Source Control lisää versionhallintaan omia metatiedostojaan mutta pitää skeematiedostot normaalissa muodossaan, jolloin näitä voidaan käsitellä myös ilman Source Controllia.

Kaikissa graafisissa työkaluissa ei ole otettu huomioon versionhallintaa, jolloin ne saattavat tallentaa skeeman yhteen tiedostoon. Tiedoston voi asettaa versionhallintaan, mutta silloinkin vain yksi kehittäjä voi tehdä tiedostoon muutoksia kerrallaan, koska versionhallinta korvaa binääritiedoston kokonaan uudella versiolla. Jos kaksi kehittäjää tekee muutoksia samaan tiedostoon, toisen on pakko korvata tiedostonsa versionhallinnasta tulevalla ja tehdä muutokset uudelleen. Versiohaarojen kanssa on mahdollista käyttää eri haaroille eri tiedostoja, mutta haarojen yhdistämisessä muutokset pitää viedä käsin toiseen tiedostoon. Versionhallinta tällaisia tiedostoja käyttäen ei siis ole hyvä ratkaisu.

4 YHTEENVETO

Nykyaikana ei pitäisi olla enää esteitä tietokannan ketterälle kehitykselle. Jos ohjelmistokehitykseen käyttää ketteriä menetelmiä, vanhanaikainen tietokannan kehitys vain haittaa itse ohjelmiston kehitystä. Monessa 2000-vuosikymmenellä julkaistussa aihetta käsittelevässä lähteessä valitettiin valmiiden versionhallintaratkaisujen puutetta. Tämä puute on pitkälti korjaantunut, ja valittavissa on useita ratkaisuja eri kannanhallintaohjelmille. Monet ratkaisut ovat jo sen verran kehittyneitä, että pahimmista lapsuksista on päästy eroon. Useat on liitetty kattavampiin ohjelmistotyökaluihin, joten niiden tulevaisuutta voidaan pitää suhteellisen turvattuna. Jos sopivaa valmista ratkaisua ei löydy, oman luominen ei ole läheskään niin monimutkainen työ kuin voisi kuvitella. Monet valmiit ratkaisutkin perustuvat samoihin lähtökohtiin, joiden mukaan voi muodostaa omanlaisensa ratkaisun. Käytetystä ratkaisusta riippumatta, tärkeää on prosessin muodostaminen tietokannan muutoksien tekemiseen ja ketterän ajattelutavan omaksuminen myös tietokannan kanssa työskenneltäessä.

Tärkeää on saada jokaiselle kehittäjälle oma projektikohtainen kanta ja valita yrityksen kehitysprosesseihin soveltuva versionhallintaratkaisu. Omien kantojen tekemisessä kätevää olisi, jos kehittäjä itse pystyisi luomaan ja poistamaan kantoja, mutta ei saa täysiä oikeuksia kannanhallintajärjestelmään. Yksi mahdollisuus olisi luoda verkkopalvelu, josta kehittäjät voivat hallita kantojaan. Tämä säästäisi kannoista vastaavien henkilöiden työtä. Tärkeintä on kuitenkin saada kanta helposti käyttöön ja tämän tehtävän voi hoitaa tietokannoista vastaava henkilökin.

Joissakin ratkaisuissa lisätyötä aiheuttaa muutosten kirjoittaminen erikseen ratkaisussa käytetyllä kielellä tai käyttämään ratkaisun omia komentoja muodostamaan tietokantaan ajettavat muutokset. Kehitystyökaluihin tai ORM-tasoon liitetyt ratkaisut auttavat tähän ainakin jonkin verran, kun muutoksista voi luoda valmiin skriptin. Tietenkin voi käyttää myös hyväkseen kantaa vertailevaa ohjelmaa muutosskriptin pohjan tekemiseen ja korvata käsin kohdat, joita vertailuohjelma ei pysty muodostamaan oikein. Joka tapauksessa kehittäjien pitää opetella käyttämään valittua ratkaisua ja sen käytöstä pitää tehdä projektille sopivat käytänteet. Aivan yhtä helppoa ei kannan versionhallinnasta saa koodin versionhallintaan verrattuna.

Versionhallintaratkaisuissa Liquibase oli kattava ja kehittynyt ratkaisu. Versiolistoissa käytettävä logiikka tuo lisää mahdollisuuksia kattavaan käyttöön, mutta lisää samalla monimutkaisuutta ja opeteltavaa. Jos tietokannan kehitystyökaluihin voi liittää versionhallinnan, tämä vaikuttaa prosessin kannalta helpoimmalta ratkaisulta ja muistuttaa eniten normaalia koodin versionhallintaa. Jos taas ohjelmassa käytetään DBAL-ratkaisua, jossa on myös kannan päivittäminen, kannattaa tähän ainakin tutustua. Näissä käytetyt ohjelmoinnin metodeilla ajettavat muutokset vaikuttivat selvemmiltä, kuin esimerkiksi Liquibasen XML-ratkaisu. DBAL ja siihen liitetty kannan päivittäminen löytyvät monista verkko-ohjelmistokehyksistä, jolloin ratkaisu on valmiina käyttöönotettavaksi. Jos projektien välillä käytetään eri ohjelmistokehyksiä, erillisratkaisu yhtenäistää prosessia. Protacon Solutionsilla on käytössä oma ratkaisu, jonka käyttötavat ovat päässeet sirpaloitumaan. Tärkeää on selventää käyttötapa kaikille, jotta ratkaisua voi käyttää huoletta. Ohjelmiston ollessa kehitysvaiheessa voi olla parempi käyttää koko kannan versionhallintaan tallennettua päivitettävää skeemaa, kuin erillisiä päivitystiedostoja muutosten suuresta määrästä johtuen. Ensimmäisen tuotantoversion jälkeen muutostiedostojen käyttö tulee aiheelliseksi.

Vaikka ratkaisut kehittyvät, moni asiaa käsittelevästä lähdemateriaalista on julkaistu viime vuosikymmenellä ja perustuu paljolti Martin Fowlerin 2000-vuosikymmenellä tehtyjen julkaisujen ympärille. Ilmeisesti tuohon aikaan alettiin huomata tietokantojen kehityksen jääminen jälkeen ketterien menetelmien yleistyessä. Uudempaa lähdemateriaalia ei juuri löytynyt ja löytyneetkin viittaavat usein Fowleriin. Toivottavasti tämä tarkoittaa, että kannan kehitys on normaalisti nykyään ketterää, eikä siitä tarvitse erikseen kirjoittaa. Huonompi vaihtoehto on, että asiaan ei edelleenkään puututa, vaikka projekteissa ongelmat tulisivatkin vastaan.

LÄHTEET

Agile software development. 2012. Wikipedia-artikkeli. Viitattu 17.11.2012. http://en.wikipedia.org/wiki/Agile_web_development.

Ambler, S. 2003. Agile database techniques.

Atwood, J. 2006. Is Your Database Under Version Control? Codinghorror. 12.12.2006. Viitattu 11.11.2012. http://www.codinghorror.com/blog/2006/12/is-your-database-under-version-control.html.

Allen, K.S. 2008. Three Rules for Database Work. OdeToCode.com. 31.1.2008. Viitattu 10.12.2012. http://odetocode.com/blogs/scott/archive/2008/01/30/three-rules-for-database-work.aspx.

Berkus, J. 2009. Wrecking Your Database. Toolbox.com 5.8.2009. Viitattu 1.12.2012. http://it.toolbox.com/blogs/database-soup/wrecking-your-database-33298.

Comparison of web application frameworks. 2012. Wikipedia-artikkeli. Viitattu 1.12.2012. http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks.

Continuous integration. 2012. Wikipedia-artikkeli. Viitattu 10.12.2012. http://en.wikipedia.org/wiki/Continuous_integration.

Database Migration. 2012. Yii guide. Viitattu 10.12.2012. http://www.yiiframework.com/doc/guide/1.1/en/database.migration.

dbdeploy. 2012. Viitattu 10.12.2012. http://dbdeploy.com/.

Doctrine 2 Migrations documentation, 2012. Viitattu 1.12.2012. http://docs.doctrine-project.org/projects/doctrine-migrations/en/latest/index.html.

Fowler, M. & Sadalage, P. 2003. Evolutionary Database Design. 1.2003 Viitattu 9.11.2012. http://martinfowler.com/articles/evodb.html.

Fowler, M. 2006. Continuous Integration. 1.3.2006. Viitattu 24.11.2012. http://www.martinfowler.com/articles/continuousIntegration.html.

Hadlow, M. 2006. How to do database source control and builds. 12.9.2006. Viitattu 25.11.2012. http://mikehadlow.blogspot.fi/2006/09/how-to-do-database-source-control-and.html.

Hunt, T. 2011. Continuous Integration for SQL Server Databases. Simple-talk. 1.3.2011. Viitattu 10.12.2012. http://www.simple-talk.com/sql/sql-tools/continuous-integration-for-sql-server-databases/.

Liquibase. 2012. Viitattu 17.11.2012. http://www.liquibase.org.

Object-relational mapping. 2012. Wikiedia-artikkeli. Viitattu 2.12.2012. http://en.wikipedia.org/wiki/Object-relational_mapping.

Ruby on Rails Migrations. 2012. RailsGuides. Viitattu 2.12.2012. http://guides.rubyonrails.org/migrations.html.

Sadalage, P. 2007. Recipes for Continuous Database Integration.

The Problem With Database Diffs. 2007. Liquibase-artikkeli. 20.6.2007. Viitattu 16.11.2012. http://blog.liquibase.org/2007/06/the-problem-with-database-diffs.html.

Verveer, H. 2011. Database Version Control. Techportal 11.01.2011. Viitattu 11.11.2012. http://techportal.inviqa.com/2011/01/11/database-version-control/.

Visual Database Tools. 2012. MSDN. Viitattu 2.12.2012. http://msdn.microsoft.com/en-us/library/y5a4ezk9.aspx.