TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MASI KAJANDER

OHJELMISTON ITSEPÄIVITYSTOIMINNALLISUUDEN TOTEUTUS

Diplomityö

Tarkastaja: Professori Kari Systä

Tarkastaja ja aihe hyväksytty 31. lokakuuta 2018

TIIVISTELMÄ

MASI KAJANDER: Ohjelmiston itsepäivitystoiminnallisuuden toteutus
Tampereen teknillinen yliopisto
Diplomityö, 49 sivua
Marraskuu 2018
Tietotekniikan tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: Professori Kari Systä

Avainsanat: ohjelmistopäivitys, jatkuva toimitus, automaattinen päivitys

Jatkuvan toimituksen avulla on mahdollista toimittaa ohjelmistotuotantoprosessissa tuotettua arvoa asiakkaalle hyvin nopeasti ohjelmistopäivitysten muodossa. Paikallisesti ajettavan työpöytäsovelluksen tapauksessa asiakkaan saataville toimitettu ohjelmistopäivitys tuottaa hyötyä kuitenkin vasta kun se on otettu paikallisesti käyttöön. Itsepäivitystoiminnallisuuden avulla voidaan automatisoida ja nopeuttaa päivitysten käyttöönottoa ja siten nopeuttaa arvontoimitusta.

Tässä työssä esitellään ohjelmiston itsepäivitystoiminnallisuuden suunnittelu ja toteutus ammattikäyttöön tarkoitettuun Windows-työpöytäsovellukseen. Sovelluksella konfiguroidaan ja tarkkaillaan automaatiojärjestelmää. Se ja työssä toteutettu toiminnallisuus on toteutettu Qt-ohjelmistokehyksen avulla. Sovellusta kehitetään automaattisen toimituksen mahdollistavassa ohjelmistokehitysprosessissa. Toiminnallisuus kytkeytyy tämän prosessin jatkeeksi verkossa olevan päivityspakettivarastonsa kautta. Kehitysprosessissa luodaan päivityspaketteja pakettivarastoon, josta toteutettu toiminnallisuus lataa ne Microsoftin NuGet-paketinhallintaohjelmaa käyttäen.

Työssä esitellään kehityksen kohteena oleva sovellus ja tausta itsepäivitystoiminnallisuuden kehittämiselle. Samalla asetetaan tavoitteet toiminnallisuudelle. Lisäksi käsitellään tutkimukseen perustuvaa teoriaa ohjelmistopäivitysten suunnittelun tueksi. Pääsisältönä käydään läpi työkaluvalinnat suunnitteluvaiheessa ja merkittävimmät toteutusratkaisut.

Tuloksena saadaan vaatimukset täyttävä itsepäivityksen toteuttava ohjelmakomponentti, joka on otettu asiakkaalla käyttöön. Työssä on arvioitu toteutetussa toiminnallisuudessa käytettyjä ratkaisuja ja esitetty käyttökontekstin ja alan kirjallisuuden perusteella jatkokehitysehdotuksia. Toiminnallisuuden kehittäminen jatkuu edelleen.

ABSTRACT

MASI KAJANDER: Implementation of a software autoupdate mechanism
Tampere University of Technology
Master's thesis, 49 pages
November 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Kari Systä

Keywords: software update, continuous deployment, feedback loop, lead time, automatic update

With a constant delivery software development process, one can deliver value to the customer very rapidly by providing software updates. In the case of a desktop application, however, value is considered delivered only when an update has been taken into use on the desktop system. With an autoupdater the deployment of updates can be automatized, speeding up the delivery of value.

This thesis presents the design and implementation of an autoupdate functionality for a professional tool running on a Windows desktop PC. The tool is used for configuring and monitoring an automation system. It, and the autoupdate functionality, have been implemented using the Qt framework. The application is being developed with a process that supports constant delivery. The process is extended with an update repository, hosting update packages created in the process. The autoupdate functionality downloads updates from the repository using Microsoft's NuGet package manager.

In the thesis, the application under development and the background for the conception of the autoupdate functionality are presented, laying out requirements for the functionality. Also research-based theory backing the design is discussed. The main part is going through the tool selection and design process and the most significant solutions in the implementation.

The result is a requirement-fulfilling component of the application. It has been taken into use at the customer. In this thesis the implementation is evaluated and, based on the context and theory, further development ideas are proposed. The development of the functionality still continues.

ALKUSANAT

Tässä diplomityössä käsitelty päivitystoiminnallisuus on tehty Wapice Oy:lle asiakasprojektiin. Työ on osa projektissa tapahtuvaa ohjelmistotuotantoprosessin kehittämistä. Uuden toiminnallisuuden luominen on ollut hyvin motivoivaa ja sopivan haastavaa.

Kiitos työni ohjaajana toimineelle ja edelleen projektissa esimiehenäni toimivalle Otto Bothakselle. Otto on antanut projektin joka vaiheessa arvokasta palautetta ja tukea. Kirjoittamisvaiheessa häneltä saamani palaute on auttanut sisällön ja kirjoittamisessa jaksamisen kanssa. Iso kiitos myös projektissa tutorina ja taistelijaparinani toimineelle Vesa Valkoselle. Merkittävä osa suunnittelusta, osa ohjelmistoon toteutetusta toiminnallisuudesta ja palvelinpään ratkaisut ovat hänen ansioitaan. Kiitos myös Toni Mattilalle ohjelmiston saloihin perehdyttämisestä ja avusta vaikeissa paikoissa.

Kiitos työni tarkastajalle professori Kari Systalle kehittävästä palautteesta ja nopean kirjoitusaikataulun mahdollistamisesta. Ilman Karin ohjausta ja nopeaa palautetta työ ei olisi valmistunut tavoitellussa aikataulussa ja laajuudessa.

Kiitos ystäville, Tampereen TietoTeekkarikillalle ja ylioppilaskunnalle unohtumattomista kokemuksista ja opeista elämään. TTY:llä vallitseva hyvä henki ja vanhempien tieteenharjoittajien jakama viisaus ovat motivoineet sekä valmistumaan että tekemään sen Hervannan teknillisestä korkeakoulusta.

Lopuksi kiitos perheelleni. Vanhempieni tuki on ollut minulle korvaamattoman arvokasta.

Tampereella, 21.11.2018

Masi Kajander

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

Apache 2.0	Apache Software Foundation -järjestön luoma salliva ohjelmistolisenssi.
Backlog	Työjono ketterässä ohjelmistokehityksessä.
BAT	Windowsin komentosarjatiedostotyyppi
BSD2, BSD3	Berkeley Software Distribution -lisenssi, 2 tai 3 ehtoa sisältävä salliva ohjelmistolisenssi
Capture	Säännöllisten lausekkeiden yhteydessä kohteena olevasta merkkijonosta lausekkeen talteen kaappaama osa
DBADPL	Don't be a dick public license, salliva ohjelmistolisenssi
GET	HTTP-pyyntö, jolla haetaan palvelimelta pyynnön mukainen resurssi
HTML	Hypertext Markup Language, seittisivuilla käytetty merkkauskieli
HTTPS	Hypertext Transfer Protocol Secure, HTTP-protokolla salatun TLS-protokollan yli
Kanban	Työtehtävien seurantataulu lean-tuotannossa
Lead time	Tuotannossa aika jonkin prosessin aloittamisen ja loppuunsaattamisen välillä
Lean	Toyotalla kehitetty prosessinkehitysmalli, jossa pyritään poistamaan turhaa työtä ja kehittämään prosessia tehokkaammaksi
Lib	Library, kirjasto
LGPL	GNU-projektin Lesser General Public License, salliva ohjelmistolisenssi.
MIT-lisenssi	Massachusetts Institute of Technologyssä kehitetty salliva ohjelmistolisenssi
REST	Representational State Transfer, HTTP-protokollalla toteutettu rajapinta-arkkitehtuuri
Rollback	Paluu aiempaan versioon
URL	Uniform Resource Locator, seittiosoite
XML	Extensible Markup Language -dokumenttistandardi

1. JOHDANTO

Ohjelmistotuotannon tehokkuus on alan yleisen kehityksen lisäksi kasvanut viime vuosina ketterien menetelmien omaksumisen myötä. [23, s. 395–398] Ketterän kehityksen tiivistahtinen ja syklinen prosessi mahdollistaa muutosten nopean toimittamisen ja siten nopean pääsyn toimitetuista muutoksista saatavaan palautteeseen. Kun päästään nopeasti palautteen perusteella arvioimaan toteutetun muutoksen käyttäjälle tuomaa arvoa, voidaan päätellä, mitä kannattaa tehdä seuraavaksi. Jos muutos toi käyttäjälle arvoa, se oli muutos oikeaan suuntaan; jos ei, muutoksen eteen tehty työ Lean-ajattelun mukaisesti turhaa. Tällainen nopeampi palautteen takaisinkytkentä parantaa ohjelmistoprosessin laatua ja tekee siitä tehokkaamman. [23, s. 398] [15, 3]

Nykyaikaisessa ohjelmistokehityksessä ketteristä menetelmistä seuraava ja niiden päälle rakentava ideologia DevOps pyrkii entisestään nopeuttamaan kehityssykliä automatisoimalla mahdollisimman monen tuotantovaiheen. Esimerkiksi ohjelmiston yhteen komponenttiin tehty muutos voidaan ottaa heti mukaan koko ohjelmistoa koskeviin automaattisiin testeihin, joiden mentyä läpi ohjelmistosta luodaan automaattisesti asennuspaketti ja toimitetaan se saataville verkkoon. Vasta tehty muutos on tällä tavalla hyvin pian käyttäjien saatavilla. [9]

Jos ohjelmistopäivitystä ei ole automatisoitu eikä ohjelmisto osaa päivittää itse itseään, uuden version käyttöönotto on keskimäärin hitaampaa. Käyttäjiä ei välttämättä kiinnosta, onko uutta versiota saatavilla. Lisäksi käyttäjillä on suuri kynnys ryhtyä manuaalisiin päivityksen tarkistus-, lataus- ja asennustoimiin. Kun mitataan uuden version julkaisemisen ja sen käyttöönoton välistä aikaa, nämä seikat muodostavat suurimman osan siitä. [18]

Ohjelmiston päivittäminen ilman päivitystoimintoa voi esimerkiksi mennä niin, että käyttäjä käy selaimella tarkistamassa ohjelmiston kotisivuilta, onko päivitystä saatavilla. Jos päivitys on saatavilla, käyttäjä lataa uuden version asennusohjelman koneelleen ja käynnistää sen. Lopuksi käyttäjä vastaa asennusohjelman kysymyksiin ja odottaa sen latautuvan loppuun. Vasta tämän jälkeen käyttäjä voi käynnistää ohjelmiston uuden version. Kaikkeen tähän kuluu merkittävä määrä aikaa, ja käyttäjältä vaaditan joka vaiheessa aktiivisia toimia.

Tässä työssä käsitellään suurta ammattikäyttöön tarkoitettua ohjelmistoa, jota kehitetään ohjelmistoyrityksessä asiakasyrityksen asiantuntijoiden käytettäväksi. Kyseisessä ohjelmistoprojektissa jatkuvan toimituksen mahdollistama asennuspaketin luominen toistuu keskimäärin kymmenen kertaa päivässä. Tällöin, jos se asetettaisiin aina automaattisesti saataville, käyttäjillä olisi mahdollisuus enimmillään kymmenen kertaa päivässä ottaa käyttöön uusi versio ohjelmistosta, todeta sen laatu ja nauttia mahdollisista parannuksista.

Käyttöönotto ei tosin automatisoidu kuin muut vaiheet ja voi muodostaa hyvinkin suuren osan sykliin kuluvasta ajasta.

Päivityksen käyttöönoton kynnystä voidaan madaltaa ja siten helpottaa käyttöönottoa automatisoimalla se. Ohjelmisto voi vaihtelevilla automaation tasoilla ottaa itse päivityksen käyttöön. Vähimmillään se poistaa käyttäjältä erilliseen päivityksen käsin asentamiseen kuluvan vaivan. Enimmillään se päivittää ohjelmiston uuteen versioon niin, että käyttäjä ei edes huomaa päivitystä tapahtuneen. [13]

Tässä työssä päivitysten käyttöönottoa nopeuttamaan suunniteltiin ja toteutettiin ohjelmistoon itsepäivitystoiminnallisuus, jolloin ohjelmisto voi asentaa itsestään uudemman version käyttöön. Tämä työ on jatkoa itsepäivitystoiminnallisuuteen liittyneelle kandidaatintyölle [13]. Toiminnallisuus noudattaa soveltuvin osin kandidaatintyössä havaittuja periaatteita. Päivitysten saatavuus tarkistetaan ohjelmiston käynnistyksen yhteydessä. Jos uusi päivitys on saatavilla, siitä ilmoitetaan käyttäjälle ja päivitystä tarjotaan asennettavaksi. Jotta käyttäjä voi tehdä valistuneen valinnan, ottaako päivityksen käyttöön vai ei, ilmoituksesta saa myös auki tarkemmat tiedot päivityksen sisällöstä.

Päivitysten latauskoon pienentämiseksi päivitystoiminnallisuus hyödyntää ohjelmiston rakenteen erityispiirrettä. Ohjelmisto koostuu useista erillisistä komponenteista, jolloin päivityksen yhteydessä voidaan ladata vain muuttuneet osat. Tällä tavalla ladataavaa on usein niin pieni määrä, että nykyaikaisella keskimääräisellä latausnopeudella käyttäjän ei tarvitse kokea joutuvansa odottamaan.

Työssä on tarkoitus selvittää, miten kirjallisuudessa esiintyvät hyvät periaatteet ja käytänteet soveltuvat ison ohjelmiston automaattisiin päivityksiin, millaisilla toteutustavoilla päästään hyviin käytänteisiin ja mitä hyviä käytänteitä havaitaan päivityksiä toteutettaessa. Perimmäisenä tarkoituksena itsepäivitystoiminnallisuudelle on ohjelmistotuotantoprosessissa tuotetun arvon nopeampi toimitus käyttäjälle. Tämä näkökulma pyritään ottamaan huomioon kaikkialla työssä ja toteutuksessa.

Työssä käsiteltävä ongelma on rajattu ohjelmistoon toteutettavan toiminnallisuuden suunnitteluun ja toteutukseen. Näihin välittömästi vaikuttavat ohjelmistotuotantoprosessin vaiheet käydään läpi toiminnallisuuden tarpeen ja sen toteutusteknisten ratkaisujen perustelemiseksi. Tarkemmin käsitellään järjestelmät, joihin toiminnallisuus on käyttöympäristössään suoraan yhteydessä. Myös yhteys käyttäjiin huomioidaan käyttäjäkokemusta arvioimalla, koska se vaikuttaa toiminnallisuuden tuomaan kokonaisarvoon.

Luvussa 2 esitellään toteutetun toiminnallisuuden käyttökonteksti. Käydään läpi kehityksen kohteena oleva ohjelmisto sekä teknisiltä että käyttäjiin ja käyttöympäristöön liittyviltä ominaisuuksiltaan. Lisäksi kerrotaan toiminnallisuuden tarpeiden arvioinnin taustoista ja ohjelmistotuotantoprosessista, jonka osaksi ominaisuutta toteutetaan. Lopuksi esitetään tarpeiden pohjalta asetetut tavoitteet.

Suunnittelun pohjana ollut teoria esitellään luvussa 3. Siinä perustellaan toiminnallisuuden tarvetta ohjelmistotuotannon näkökulmasta. Esille tuodaan myös toiminnallisuuden suunnitteluratkaisujen taustalla olevia ohjelmistotuotannon ja käyttäjäkokemuksen periaatteita. Luvussa esitellään itsepäivityksessä käytettävät termit ja kuvaillaan teoriatasolla ohjelmiston päivityksen vaiheet. Suunnittelun eteneminen käsitellään työkaluvalintojen kautta luvussa 4. Tiettyjä ongelmia ratkaisevien työkalujen tarve perustellaan, niiden valinnan kriteerit selvitetään ja kaikista arvioiduista työkaluista arvioidaan niiden soveltuvuus käytettäväksi toiminnallisuuden toteuttamisessa.

Luku 5 on työn sisällön pääosa. Luvussa esitellään toteutettu päivitystoiminnallisuus ja kuvaillaan, miten toteutukseen valittuja työkaluja käytetään. Toiminnallisuuden tekniset ratkaisut perustellaan. Ratkaisut arvioidaan luvussa 6. Siellä tarkastellaan asetettujen tavoitteiden toteutumista, keskustellaan mahdollisista vaihtoehtoisista toteutustavoista ja esitetään havaittuja jatkokehitysmahdollisuuksia. Lopuksi luvussa 7 kerrataan lyhyesti työn kulku ja tulokset.

2. TYÖN TAUSTA

Työn tavoitteena on suunnitella ja toteuttaa itsepäivitystoiminnallisuus suureen ohjelmistoon ja osaksi jatkuvan toimituksen mahdollistavaa ohjelmistokehitysprosessia. Kohdassa 2.1 ja 2.2 kuvaillaan kohteena oleva ohjelmisto ja tuotantoprosessi, joihin toiminnallisuus suunnitellaan ja toteutetaan. Niiden jälkeen kohdassa 2.3 kerrataan ohjelmistotuotantoprosessin vaiheet keskittyen laatuun. Lopuksi suunnittelun ja toteutuksen pohjana ollut taustatutkimus ja tavoitteet käydään läpi kohdassa 2.4.

2.1 Kehityksen kohteena oleva ohjelmisto

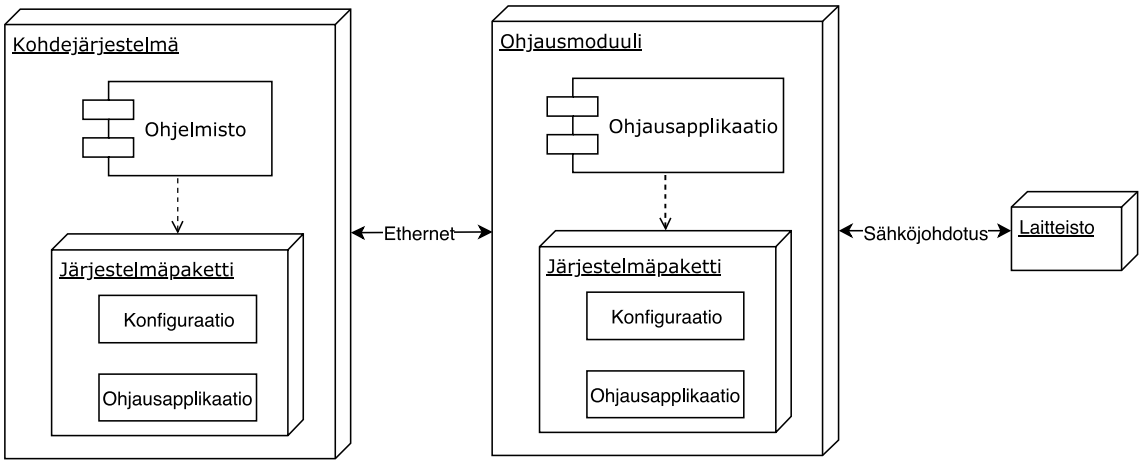
Ohjelmisto on Windows-käyttöjärjestelmässä ajettava työpöytäsovellus, joka on toteutettu C++:lla ja Qt-ohjelmistokehyksellä [40]. Tässä työssä kehityksen kohteena olevasta ohjelmistosta, johon itsepäivitystoiminnallisuus tehtiin, käytetään nimeä ohjelmisto ja ohjelmistoa ajavasta käyttöjärjestelmästä käytetään nimeä kohdejärjestelmä.

Ohjelmistoa kehitetään asiakkaalle tämän tarpeita varten. Ohjelmisto on ammattityökalu, jolla on ainoastaan ammattikäyttäjiä. Sitä käyttävät asiakkaan työntekijät. Toiminnallisuudet ovat laajoja ja niiden muodostama kokonaisuus on monimutkainen.

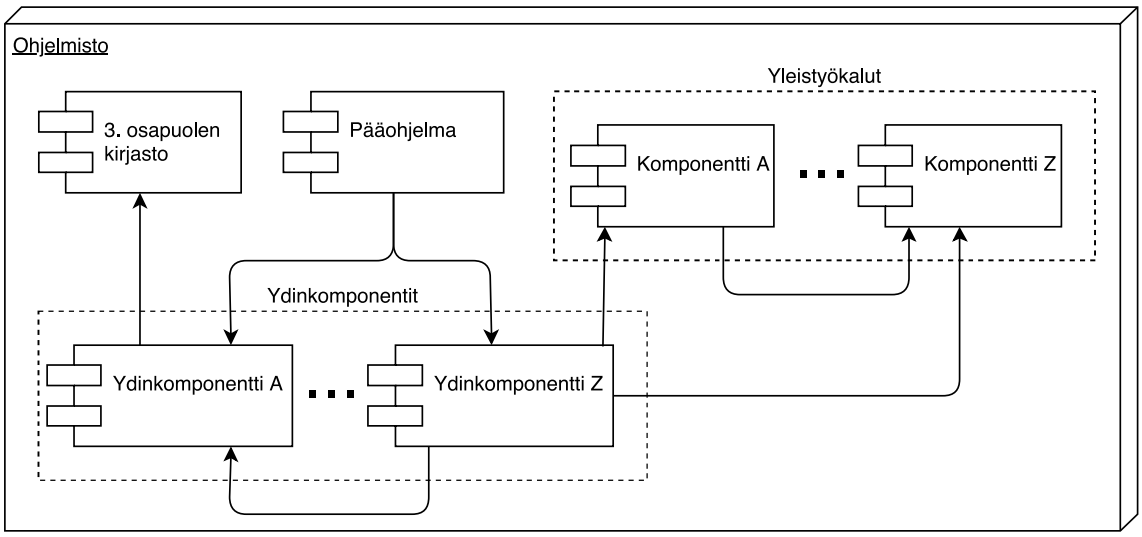
Käyttötarkoituksena ohjelmistolla on automaatiojärjestelmän konfiguraatioiden muokkaaminen ja automaatiojärjestelmän toiminnan tarkkailu. Kuvassa 1 on esitetty ohjelmiston kytkeytyminen automaatiojärjestelmään. Ohjelmisto on siinä kohdejärjestelmässä ajossa. Kohdejärjestelmä on ethernet-liitännällä kiinni automaatiojärjestelmän jossakin ohjausmoduulissa. Ohjelmiston avulla siirretään ohjausmoduulille sille laadittuja järjestelmäpaketteja ja tarkkaillaan sen toimintaa. Järjestelmäpaketin sisältämä ohjausapplikaatio on ohjausmoduulilla ajossa ja ohjaa tämän toimintaa. Sähköjohdotuksen kautta ohjausmoduuli hallitsee siihen kytkettyä laitteistoa ja kerää sensoreilta tietoa.

Ohjelmisto muodostuu liitännäisinä dynaamisesti ladattavista komponenteista, jotka on jaoteltu hierarkisiin ryhmiin niiden tarjoaman toiminnallisuuden luonteen mukaan. Kuvassa 2 on havainnollistettu ohjelmiston jakautumista komponentteihin ja komponenttien välisiä riippuvuuksia. Esimerkkejä ryhmistä ovat muun muassa käynnistyksen yhteydessä ladatavat komponentit, ydintoimintoihin liittyvät komponentit ja yleistyökaluina toimivat komponentit. Komponenttien välillä on monimutkaisia riippuvuuksia.

Ohjelmisto on kehityselinkaaressaan saavuttanut kypsyyden. Kaikki sen perustoiminnallisuudet ovat olleet jo pitkään käytössä. Ohjelmisto on osa suurempaa automaatioalustaa ja se on konfiguraatioiden ja ohjausmoduulien suhteen taaksepäin yhteensopiva pitkän ajan



Kuva 1. *Ohjelmisto käyttökontekstissaan osana automaatiojärjestelmää. [43]*



Kuva 2. *Havainnollistus ohjelmiston koostumisesta komponenteista ja komponenttien keskinäisistä riippuvuuksista.*

päähän. Tästä ja yleisestä toiminnallisuuksien laajuudesta johtuen ohjelmiston asennuspaketti on tiedostokooltaan suuri. Samasta syystä ohjelmiston versionumero on monimutkainen muodostuen käytetyn automaatioalustan versionumerosta ja sen perässä olevasta ohjelmiston omasta versiosta.

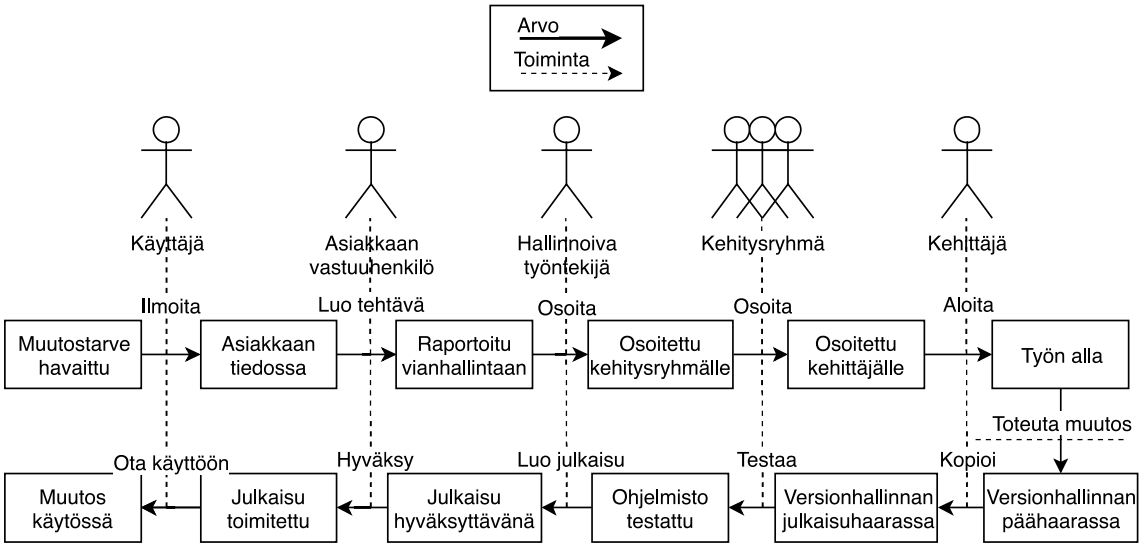
2.2 Ohjelmistotuotantoprosessi

Ohjelmistoa kehitetään ketterillä menetelmillä. Ohjelmistokehitys on jaksotettu säännöllisiin julkaisuihin, jotka koostuvat useammista ohjelmistoon tehdyistä muutoksista. Muutoksille on aina perusteltu ja asiakkaan kanssa läpi käyty tarve. Säännöllisestä julkaisusta huolimatta tuotantoprosesissa on käytössä nopea jatkuvan toimituksen mahdollistava työkalukokoelma, jonka avulla kehitys versionhallintajärjestelmän päähaaraan on ohjelmiston suuri koko huomioon ottaen nopeaa. Kehittämiseen osallistuu useita noin kymmenestä ohjelmistokehittäjästä koostuvia ryhmiä, joilla on kullakin oma erityisvastuunsa ohjelmistokehityksen erilaisista piirteistä ja ohjelmiston eri osista. Käydään läpi ohjelmistokehitysprosessin arvoketju muutostarpeesta ohjelmiston julkaisuun.

Kuvassa 3 on esitetty vuokaavio tilanteesta, jossa asiakkaan havaitsema muutostarve ja sen pohjalta laadittu muutos kulkevat ohjelmistokehitysprosessin läpi. Kuvan vuokaavio soveltuu esimerkiksi tilanteeseen, kun käyttäjä on ohjelmiston käytön yhteydessä havainnut siitä vian, jolloin muutostarve on havaitun vian korjaaminen. Aluksi ohjelmiston käyttäjä havaitsee muutostarpeen, joka voi olla vika, parannus tai tarve uudelle ominaisuudelle, ja ilmoittaa siitä oman organisaationsa sisällä. Asiakkaan tuotteesta vastaava työntekijä raportoi muutostarpeesta vianhallintaohjelmaan luoden sinne työtehtävän muutoksen toteuttamiseksi. Tästä alkaa muutoksen *lead timen* eli raportoinnin ja muutoksen toimitamisen välisen ajan laskenta.

Asiakkaan työntekijän luoma työtehtävä odottaa ensin, että projektia hallinoiva työntekijä osoittaa sille jonkin kehitysryhmän. Ryhmillä on käytössään kanban-taulut, joiden avulla tehtävää työtä suunnitellaan ja seurataan. Kun työtehtävä on osoitettu ryhmälle, se ilmestyy tämän kanban-taulun *backlogiin* eli työjonoon. Ryhmän päivittäisissä kokouksissa osoitetaan ryhmän jäsenille työtehtäviä. Työtehtävä osoitetaan jollekin sen suorittamiseen pätevälle työntekijälle. Kun hän on saanut mahdollisesti keskeneräiset muut työnsä valmiiksi, hän ottaa tehtävän työn alle.

Tehtävän ollessa työn alla käydään läpi useita vaiheita kehitysryhmän sisällä. Näiden vaiheiden läpikäymisen jälkeen palataan käsittelemään seuraavia kuvan 3 vaiheita. Työn alla olevan tehtävän kehitysryhmän sisäiset vaiheet on havainnollistettu kuvassa 4. Kuva on rajattu käsittämään ainoastaan henkilöstön siihen vaikuttavat toimet. Erilaiset jatkuvan toimituksen järjestelmät antavat kuitenkin palautetta samalla tavalla kuin kuvassa 6. Työtehtävä alkaa toteutuksesta. Kun kehittäjä on toteuttanut muutostarpeen tyydyttävän muutoksen, hän lähettää työnsä katselmoitavaksi Gerrit-katselmointityökaluun ja pyytää kehitysryhmänsä muilta kehittäjiltä palautetta. Palautetta voivat antaa muutkin ohjelmistokehittäjät. Lisäksi Jenkins-integraatiopalvelin [11] kääntää koodin ja ajaa sille testejä.

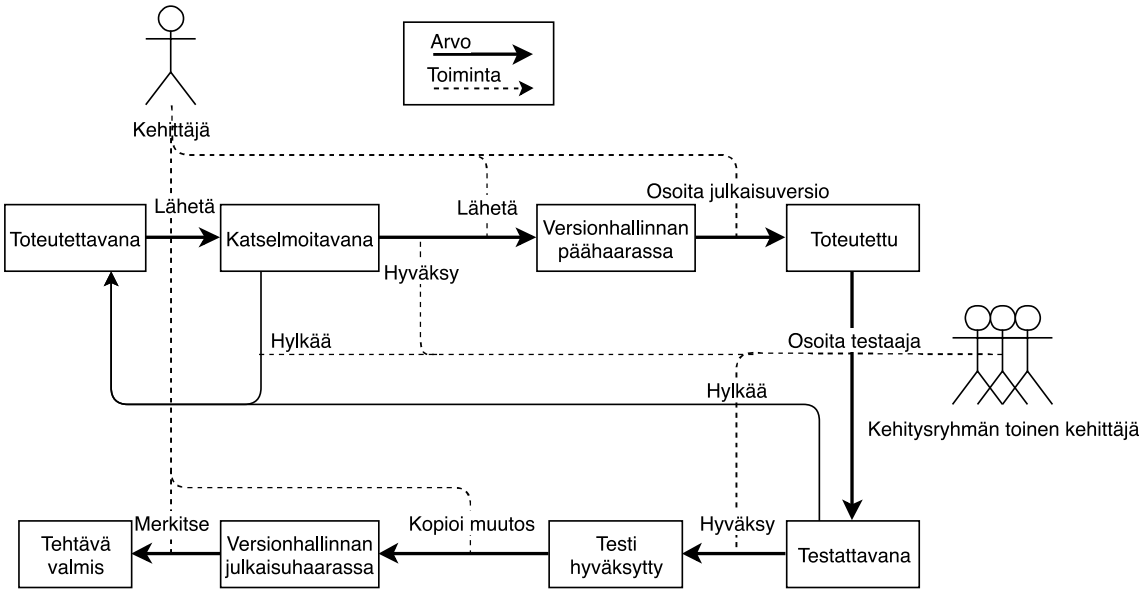


Kuva 3. *Ohjelmiston käyttäjän havaitseman muutostarpeen, siitä luodun työtehtävän, toteutetun muutoksen ja lopulta muutoksen sisältävän julkaisun kulku ohjelmistokehitysprosessin läpi.*

Katselmoinnissa arvioidaan muutetun ohjelmakoodin laatua, johon vaikuttavat esimerkiksi sen helppolukuisuus, ohjelmistoprojektin tyylioppaan mukaisuus ja virheiden esiintyminen. Kehittäjä parantelee muutosta palautteen perusteella, kunnes se läpäisee katselmoinnin ja Jenkinsin testit, jolloin kehittäjä voi lähettää sen eteenpäin versionhallinnan päähaaraan.

Päähaarassa olevat muutokset käännetään ja testataan kattavasti TeamCity-integraatiopalvelimella [12]. Viimeistään tässä vaiheessa kehittäjä selvittää ja merkitsee työtehtävään, mihin ohjelmiston tulevaan julkaisuun muutos on tarkoitettu. Kehittäjä voi tarpeen mukaan myös aloittaa uudestaan kuvan 4 alkuvaiheesta, jos hän jakaa tarvittavan muutoksen useampaan osaan. Kun kaikki tarvittavat osat ovat versionhallinnan päähaarassa, työtehtävään voidaan merkitä, että siinä oleva muutos on toteutettu. Jos muutoksen toiminnallisuudelle ei vielä ollut sen oikean toiminnan varmistavaa manuaalista testitapausta, kehittäjä luo sellaisen. Seuraavassa kehitysryhmän päivittäisessä kokouksessa työtehtävän muutokselle osoitetaan testaaja, ellei joku ole jo ehtinyt testaamaan sitä. Suoritettuaan testin testaaja joko merkitsee työtehtävän testin hyväksytyksi tai kertoo kehittäjälle, että muutos ei toteuta testin vaatimuksia. Tällöin voidaan palata jälleen alkuun laatimaan uutta muutosta. Hyväksytty testi oikeuttaa kehittäjän kopioimaan muutoksen versionhallinnan päähaarasta julkaisuhaaraan ja merkitsemään tehtävän valmiiksi.

Kuvan 4 mukaisesti ohjelmistokehityksen tuloksena versionhallinnan päähaaraan ja julkaisuhaaraan laitettu muutos jatkaa etenemistä kuvan 3 mukaisesti. Ohjelmistosta julkaistaan uusia virallisia versioita joitakin kappaleita vuodessa. Ohjelmistoprojektin johtaja ja muut asiakkaan kanssa yhteyksissä olevat työntekijät sopivat asiakkaan kanssa, mitä ominaisuuksia uuteen versioon otetaan ja mitä ominaisuuksia kehitetään tulevia versioita varten. Julkaisujen aikataulut sovitaan ennalta, ja julkaisupäivää ennen on usean viikon jakso,



Kuva 4. *Työtehtävän kulku ohjelmistokehitysryhmän toimien seurauksena.*

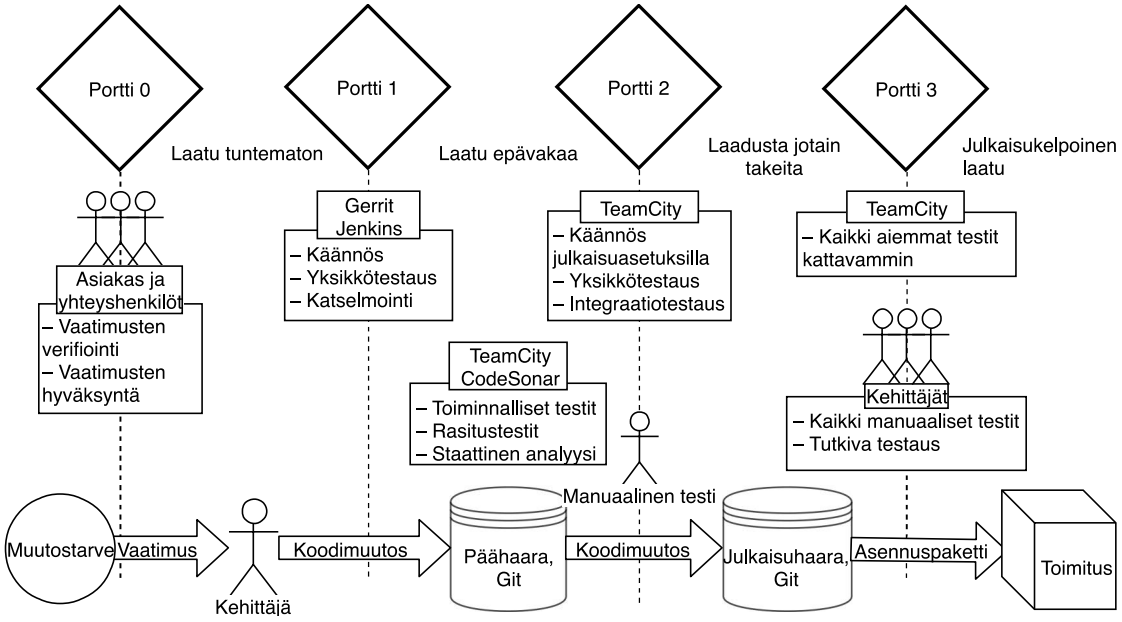
jolloin julkaisua varten suoritetaan valmistelevia toimenpiteitä. Näitä ovat muun muassa ominaisuuksien poimiminen julkaisuhaaraan, ohjelmiston julkaisutestaus ja virallisen ohjelmiston julkaisuversion asennuspaketin luominen.

Julkaisun toimenpiteisiin kuluu paljon työaikaa ja resursseja. Ohjelmiston julkaisuversiolla ajetaan kaikki testijärjestelmän automaattiset testit ja hyvin pitkiä automaattisia eitoiminnallisia rasitustestejä. Lisäksi suoritetaan kaikki manuaaliset testit. Testaamiseen kuluu paljon aikaa ja siihen osallistuu suuri joukko kehittäjiä. Havaitut viat ilmoitetaan ja korjataan nopealla aikataululla. Versionhallinnan päähaara rauhoitetaan tietyksi ajaksi ennen julkaisua, jolloin sinne ei saa lisätä muutoksia normaalin ohjelmistokehityksen puitteissa. Tällä tavalla varmistetaan julkaisuhaaraan menevien korjausten vakaus, kun muut muut muutokset eivät voi aiheuttaa regressiivisiä ongelmia julkaisuhaaraan menevien korjausten testaamiselle.

Virallisen asiakkaalle toimitettavan ohjelmiston julkaisun luominen on käsityötä ja voi kestää sitä suorittavalta työntekijältä täyden työpäivän. Ohjeessa julkaisun luomiseen on noin 30 vaihetta, joihin kuuluu muun muassa ohjelmiston julkaisuun liittyvien konfiguraatiotiedostojen muokkaaminen, kääntäminen, asennuspaketin luominen ohjelmiston mukana käännetyllä sitä varten luodulla ohjelmalla ja asennuspaketin asettaminen saataville.

2.3 Laatu tuotantoprosessissa

Kohdassa 2.2 kuvatussa kehitysprosessissa kehittyvä arvo ja siitä varmistuva laatu on havainnollistettu myös kuvassa 5, jossa muutostarpeesta alkanut työ kulkee eri laatuporttien läpi. Kunkin laatuportin läpäistyään muutokselle voidaan taata tietty laatu. Porttia 0 vartioivat asiakkaan edustajat, projektin toimihenkilöt ja muut mahdolliset vaatimuksiin vaikuttavat osakkaat. Ilmoitettujen ja havaittujen muutostarpeiden pohjalta he verifioivat



Kuva 5. *Ohjelmistokehitysprosessissa olevat laatuportit ja niiden takaama laatu.*

vaatimusten tarpeen ja hyväksyvät vaatimukset projektiin. Tässä vaiheessa työtä vaatimusten täyttämiseksi ei ole vielä tehty, joten työn laadusta puhuminen ei ole mielekästä.

Kun kehittäjä alkaa vaatimusten perusteella laatia ohjelmistoon muutosta, tehdyn työn laatu on vielä prosessille tuntematon. Ensimmäinen varmistus laadusta saadaan laatuportilla 1, jonka vartioimiseen käytetään Gerrit-katselmointityökalua ja Jenkins-integraatiopalvelinta. Jenkins kääntää ohjelmiston uuden muutoksen kanssa ja ajaa sille yksikkötestejä. Käännöksen tulee tapahtua ilman virhetä tai varoituksia ja testien tulee mennä läpi. Muut ohjelmistokehittäjät voivat Gerritin kautta antaa palautetta ja hyväksyä tai hylätä muutoksen. Portin 1 läpäistyään muutos päätyy versionhallinnan päähaaraan, jossa laatu on epävakaa. Tällä laatutasolla toimintojen vaatimuksienmukaisesta toiminnasta ei ole takeita, ja ohjelman kaatavia vikoja voi ilmetä peruskäytössä.

Päähaarassa olevat muutokset otetaan TeamCity-integraatiopalvelimelle testattavaksi. Laatuportissa 2 niiden on läpäistävä TeamCityn ajamat testit ja ylempänä kuvan 4 yhteydessä kuvailtu manuaalinen testi. TeamCityssä ohjelmisto käännetään julkaisuasetuksilla ja siitä luodaan asennuspaketti. Käännettyä ohjelmistoa testataan ajamalla sille yksikkötestit ja toiminnallisia integraatiotestejä. Useampi muutos saatetaan kääntää ja testata kerralla, jos muutoksia tulee nopeammin kuin testejä ehditään ajaa. Epäonnistuneista testeistä lähetetään viesti kehittäjille, joiden lähettämät muutokset olivat mukana epäonnistuneessa testiajossa.

Laatuporteista erillään joka yö ja viikonloppu ajetaan pitkäkestoisia toiminnallisia testejä niin paljon kuin ehditään. Näiden testien tulokset antavat lisää varmuutta laatuportin 2 ohittaneiden muutosten laadulle. Viikoittain ajetaan myös staattinen analyysi CodeSonar-analyysityökalulla [8], joka paljastaa mahdollisia vikoja ohjelmakoodia tutkimalla.

Laatuportin 2 jälkeen voidaan taata, että ohjelmistossa ei ole enää helposti ilmi tulevia vikoja ja että ydintoiminnallisuudet toimivat suurella varmuudella. Versionhallinnan julkaisuhaarassa saattaa olla vielä muutoksia, joita ei olla testattu toiminnallisilla testeillä. Tämän vuoksi ennen julkaisua ajetaan viimeisessä laatuportissa kaikki aiemmat testit. Pitkään kestävät rasitustestit ajetaan pidempikestoisina ja perusteellisempina. Kehittäjät suorittavat myös kaikki manuaaliset testit julkaisua varten ja testaavat ohjelmistoa .

2.4 Taustatutkimus ja asetetut tavoitteet

Ennen tämän työn aloittamista ohjelmiston nopeamman toimituksen mahdollistamisesta oli suoritettu taustatutkimus. Ohjelmistoon oli aiemmin yritetty kehittää itsepäivitystoiminnallisuutta, mutta se ei ollut koskaan päätynyt käyttöön asti.

Ennen työn aloittamista oli myös luotu kohteena olevasta ohjelmistosta pienoismalli, joka on arkkitehtuuriltaan ja tiedostorakenteeltaan kuin ohjelmisto eli kuvan 2 mukainen. Tätä ohjelmistoa käytettiin luvussa 4 kuvailluissa soveltuvuuskokeissa.

Projektissa työlle asetettiin kolme tavoitetta. Pääasiallinen tavoite on lyhentää ohjelmiston toimitusaikaa. Muut tavoitteet olivat laadun takaaminen tuotantokäytössä, palautteen käyttäjiltä keräämisen mahdollistaminen ja toiminnallisuuden pitäminen jatkokehitykselle avoimena. Tavoitteiden pohjalta toiminnallisuudelle johdettiin toiminnallisia ja ei-toiminnallisia vaatimuksia, joiden pohjalta se suunniteltiin ja toteutettiin.

Pääasia on minimoida ohjelmistoon tehdyn muutoksen *lead time* eli aika muutostarpeen havaitsemisesta muutoksen toimittamiseen. Nopean toimittamisen tavoite synnytti vaatimukset, että toiminnallisuudella tulee voida ladata päivitys nopeasti heti kun se tulee saataville ja että saataville asettaminen voidaan tehdä ohjelmistokehitysprosessissa pienellä vaivalla. Ei-toiminnallisia tavoitteesta seuraavia vaatimuksia ovat erityisen sujuva käyttökokemus ja käyttäjien huolettomuus virheistä. Jotta mahdollisimman moni käyttäjä ottaisi päivitykset käyttöön ja siten keskimääräinen *lead time* lyhenisi, käyttöönoton tulee olla heille helppoa ja riskitöntä. Virheistä toipumiseksi ja riskien alentamiseksi vaatimukseksi tuli myös mahdollisuus palata nopeasti edelliseen käytössä olleeseen versioon.

Toinen tavoite toimitettavien päivitysten laadun takaamiseksi on ristiriidassa pääasiallisen nopean toimittamisen tavoitteen kanssa niin, että kumpikin tavoite ei voi täydellisesti täyttyä samaan aikaan ja että paras lopputulos on kompromissi näiden kahden välillä. Muutostarpeen havaitsemisen ja käyttöönoton välistä aikaa on mahdollista lyhentää tai pidentää julkaisemalla päivitykset aiemmin tai myöhemmin testausprosessissa. Tavoitteen pohjalta asetettiin vaatimukseksi vaiheistettu päivitysten julkaiseminen niin, että tuotantoon menee vain julkaisukelpoista laatua, mutta rajatulle joukolle voidaan julkaista aiemmin vähemmän testattuja muutoksia.

Kolmantena tavoitteena on mahdollistaa palautteen kerääminen käyttäjiltä ohjelmiston kautta, jotta tulevasta kehityksestä päätettäessä olisi käytettävissä tietoa päätösten perus-

teiksi. Tämän toteuttavaksi vaatimukseksi asetettiin palautteen kerääminen käyttäjiltä, jotka palaavat päivittämästään versiosta takaisin aiempaan.

Viimeinen tavoite jatkokehityksen mahdollistamisesta tarkoittaa sitä, että vaikka toiminnallisuutta ei aluksi toteuteta jatkuvan toimituksen tai käyttöönoton tilanteeseen, se on laajennettavissa kattamaan nekin tilanteet. Tämä asettaa vaatimuksia toteutukseen valittaville työkaluille. Niiden tulee olla yleisesti ongelmaan sovellettavissa niin, että niiden käyttöä voidaan laajentaa jatkuvan käyttöönoton tilanteessakin. Päivityksen käyttäjäkokemuksen on myös oltava muokattavissa kuhunkin käyttötarkoitukseen sopivaksi.

Asetettuja tavoitteita käytettiin toteutuksen suunnittelussa ja käytettävien työkalujen valinnassa, joista kerrotaan luvussa 4. Tavoitteiden toteutumista toteutetussa itsepäivitystoiminnallisuudessa arvioidaan luvussa 6.

3. OHJELMISTON ITSEPÄIVITYS JATKUVASSA TOIMITUKSESSA

Tässä luvussa käydään läpi yleiset periaatteet ohjelmistojen itsepäivityksistä ja siitä, miten itsepäivitystoiminnallisuus on jatkuvan toimituksen ja käyttöönoton kannalta tärkeä ominaisuus ohjelmistossa, jota käyttäjät ajavat kohdejärjestelmissään paikallisesti. Lisäksi esitetään muita jatkuvan toimituksen tuomia etuja. Käsiteltävänä on muutosten toimittaminen jo toimitettuun ja käyttöönotettuun ohjelmistoon, joka päivittää itseään.

Ensin kohdassa 3.1 käsitellään arvon nopeampaa toimittamista. Nopean tuotantoprosessin laadunvarmistuksesta puhutaan kohdassa 3.2. Päivitysten tarjoamisen eri vaihtoehdoista kerrotaan kohdassa 3.3. Lopuksi kohdassa 3.4 käsitellään erityisesti päivitystoiminnallisuuteen liittyviä käyttäjäkokemus- ja virhetilannetekijöitä.

3.1 Arvon nopeampi toimittaminen

Arvon nopeaa toimittamista voidaan pitää itseisarvoisesti tavoiteltavana, mutta se tuo myös etuja tehokkuudessa. Lehtonen ja kumppanit raportoivat tutkimuksessaan [15] ohjelmistoon tehtyjen muutosten toimitusajan lyhentämisen vähentävän ohjelmistotuotantoprosessissa tehtävää turhaa työtä. Tämä turha työ johtui esimerkiksi tehtävästä toiseen vaihtamisessa syntyvistä kontekstinvaihdoista, keskeneräisiin työtehtäviin palaamisista ja suuremmasta viiveestä palautteen saamisessa. Vähennettäessä muutosten toimitusaikaa tehdään siis tuotantoprosessista tehokkaampi, kun siinä käytetään vähemmän aikaa turhaan työhön.

Ilman suoraa yhteyttä loppukäyttäjiin on vaikeaa saada tietoa uusien ominaisuuksien suunnittelupäätöksen tueksi. Kehitysorganisaatiolla on ongelmana päättää, mihin suuntaan kehitystä priorisoidaan ja millä tavalla jokin ominaisuus kannattaisi toteuttaa, jotta saadaan toimitettua asiakkaalle mahdollisimman paljon arvoa kuten loppukäyttäjiä eniten hyödyttäviä parannuksia. Holmström ja Bosch [24] ovat luoneet jatkuvan toimituksen prosesseille sopivan HYPEX-kehitysmallin, jota he esittävät ratkaisuksi ongelmalle. Mallissa luodaan muutokselle asetetun korkeamman tason tavoitteen perusteella työjonoon tehtäviä ja sitten valitaan niistä toteutettavaksi jokin. Tämän tehtävän tuloksena syntyvälle muutokselle arvioidaan sen aiheuttama odotettu käyttäjän reaktio. Kun muutos toimitetaan käyttäjille, vertaillaan käyttäjien toimista havaittua varsinaista käyttäytymistä ennalta tehtyyn arvioon. Jos arvio oli väärä, laaditaan hypoteesi, miksi se oli väärä. Hypoteesin pohjalta toimitetaan käyttäjille uusi muutos ja toistetaan sykliä, kunnes arvio osui oikeaan. Tällä tavalla saadaan nopeasti varmistettua, että on tehty arvoa lisääviä muutoksia.

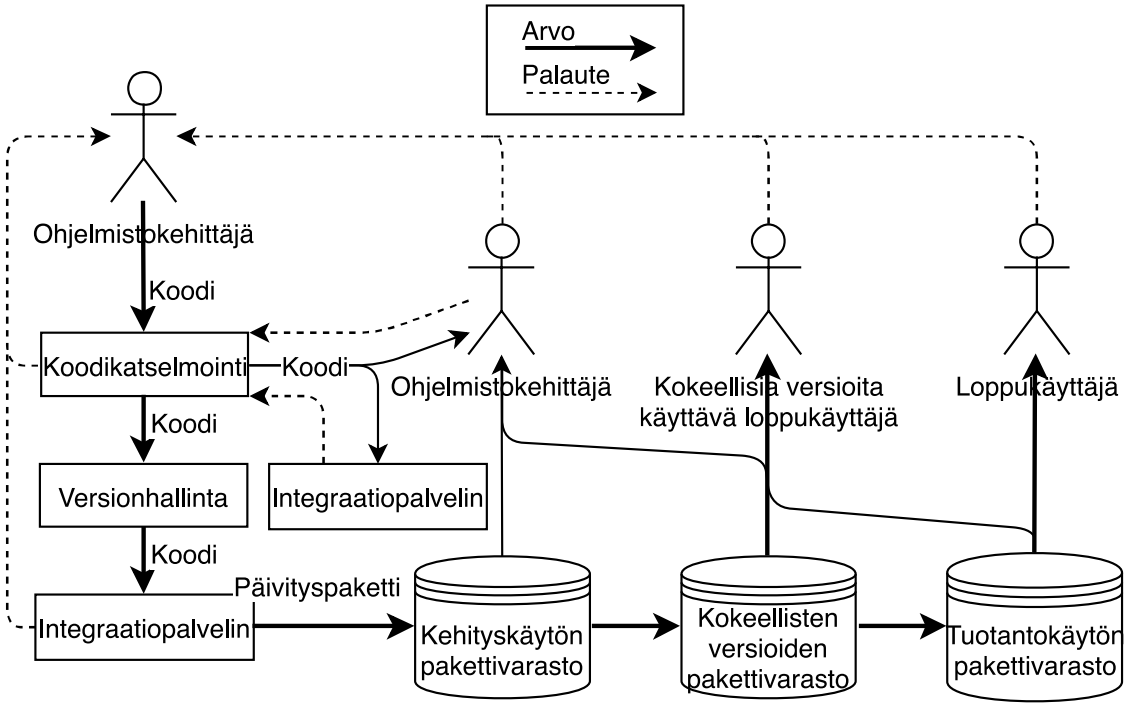
Holmströmin ja Boschin malli[24] on tarkoitettu tilanteeseen, jossa jatkuva toimitus suoraan asiakkaille on käytössä, eikä toimittajan ja asiakkaiden välillä ole tiivistä keskustelukanavaa palautteen saamiseksi ja palautteen takaisinkytkennän mahdollistamiseksi. Sen sijaan mallissa on tarkoitus käyttää ohjelmiston keräämää metriikkaa ja muuta tietoa selvittämään käyttäjien käyttäytymistä, kun muutos johonkin ohjelman toimintoon on tehty. Kuten kohdassa 2.1 on kuvailtu, tässä työssä käsiteltävällä ohjelmistolla on toimittajan näkökulmasta yksi asiakas ja siihen on toimiva keskusteluyhteys. Tilanteeseen voidaan kuitenkin soveltaa Holmströmin ja Boschin mallia, kun heidän mallinsa asiakkaana pidetään ohjelmiston loppukäyttäjiä, joihin ei ole suoraa keskusteluyhteyttä palautteen saamiseksi.

3.2 Laadunvarmistus

Perinteisellä manuaalisella julkaisuprosessilla ohjelmistosta julkaistava versio käy yleensä läpi mittavan manuaalisen testauksen ja laadunvarmistuksen automaattisen testauksen lisäksi. Jos päivitysten julkaisutahti kuitenkin on jatkuvan toimituksen prosessissa merkittävästi suurempi kuin perinteisessä mallissa, samoja perusteellisia laadunvarmistustoimia ei voida suorittaa ilman merkittävää resurssipanostusta. Nopeamman, useimmin tapahtuvan ja hitaamman harvemmin tapahtuvan julkaisun välisistä laatueroista on tutkimusta. Khomh ja kumppanit [14] suorittivat tapaustutkimuksen vertaamalla Mozilla Firefox -projektin laatumittareita ennen ja jälkeen nopeamman toimituksen kehitysprosessin käyttöönottoa. He havaitsivat, että käyttäjät eivät kokeneet merkittävästi enempää vikoja Firefoxissa, kun sen automaattisten päivitysten julkaisutahtia nopeutettiin. Sen sijaan he havaitsivat, että viat korjattiin nopeammin ja että viat tulivat nopeammin esille käytön aikana.

Khomin ja kumppaneiden tutkimuksessa [14] huomionarvoista oli se, että uudet päivitykset saivat heti paljon käyttöä, kun ne toimitettiin nopeasti laajalle käyttäjäjoukolle. Tämä tuo osin samanlaista etua kuin edellisessä kappaleessa kuvattu manuaalinen testaus ja laadunvarmistus. Ongelmana voidaan kuitenkin pitää sitä, että vikojen toimittaminen – varsinkin nopeasti ohjelman käytössä ilmenevien – ei ole lainkaan toivottavaa, eikä käyttäjiä tule kohdella testaajina. Heidän tulee voida luottaa ohjelmiston täyttävän samat laatukriteerit kuin manuaalinen julkaisuprosessi takasi. Ratkaisuksi tähän voidaan soveltaa vaiheistettua julkaisemista.

Kuvassa 6 on havainnollistettu esimerkki kolmivaiheisesta päivityksen toimittamisesta jatkuvan toimituksen ohjelmistotuotantoprosessissa ja siitä, miten arvo ja palaute liikkuvat tuotantoprosessissa. Kuvassa ohjelmistokehittäjä on laatinut ohjelmiston koodiin muutoksen ja lähettää sen otettavaksi käyttöön. Ensin koodimuutos päätyy katselmoitavaksi. Sitä kautta koodi tulee muiden ohjelmistokehittäjien näkyville ja he voivat antaa siitä palautetta katselmointisovelluksen kautta. Katselmointisovellukseen on myös liitetty integraatiopalvelin, joka kääntää koodin ja ajaa sille testejä. Jos käännös ja testit suoriutuvat ongelmitta, integraatiopalvelin antaa palautteeksi hyväksyntänsä muutokselle. Kun sekä kanssaohjelmistokehittäjien palaute että integraatiopalvelimen hyväksyntä on saatu,

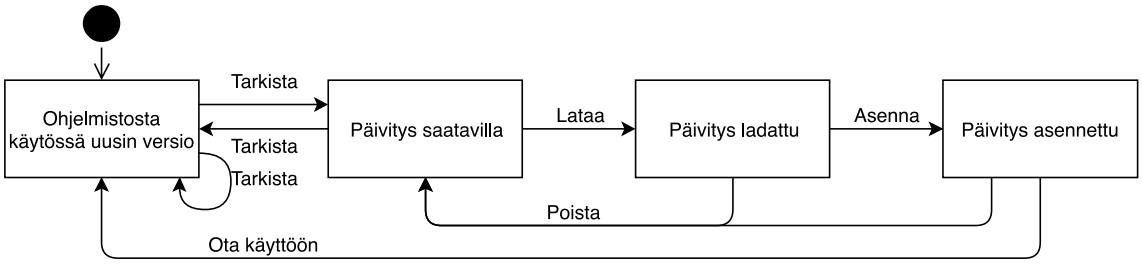


Kuva 6. *Palautteen takaisinkytkentä ohjelmistokehitysprosessissa ja päivitysten vaiheistus eri pakettivarastoissa.*

ohjelmistokehittäjä voi lähettää muutoksensa versionhallintaan. Toinen integraatiopalvelin lataa uuden versionhallintaan tulleen muutoksen, luo sen pohjalta ohjelmiston julkaisuversion ja ajaa tälle kattavasti testejä. Jos testit suoriutuivat ongelmitta, integraatiopalvelin luo myös päivityspaketin ja julkaisee sen sisäiseen kehityskäyttöön tarkoitettuun pakettivarastoon.

Kun kuvan 6 päivityspaketti on päätynyt kehityskäytön pakettivarastoon, ohjelmistokehittäjät voivat ottaa sen käyttöön ohjelmiston itsepäivitystoiminnon avulla. Käytössä he voivat arvioida sen toimivuutta ja antaa palautetta, jos päivityksen mukana tuli epäsuotuisia muutoksia. Päivityspaketti voidaan samalla julkaista osan loppukäyttäjistä saataville kokeellisten versioiden pakettivarastoon, mutta julkaisussa voidaan myös odottaa, ettei päivityksestä kehityskäytössä jonkin ajan kuluessa ilmene vikoja. Odottaessaan julkaisua kokeellisten versioiden pakettivarastoon päivitystä voidaan myös automaattitestata erittäin kattavasti. Lisäksi Khomh ja kumppanit [14] mainitsevat muissa tutkimuksissa [28, 10] havaitun, että liian usein julkaistavat päivitykset vähentävät käyttäjien jaksamista ottaa päivitykset käyttöön.

Ohjelmistokehitysprosessissa taatusta laadusta monivaiheisesta päivitysten julkaisusta huolimatta julkaistussa ohjelmiston uudessa versiossa voi olla vikoja. Jos verrataan jatkuvan toimituksen julkaisutahtia aiemmin kohdassa 2.2 kuvattuun, huomataan ettei jatkuvalla toimituksella ole mahdollista suorittaa niin kattavia julkaisutestejä kuten perinteisellä manuaalisella julkaisuprosessilla. Samoja manuaalisia laadunvarmistustoimia ei voida suorittaa niin lyhyessä ajassa ilman kohtuuttomia resurssipanostuksia. Kuten Khom ja



Kuva 7. *Ohjelmistopäivityksen asentumisen tasot ja siirtymät niiden välillä.*

kumppanit totesivat [14], nopeampi julkaisu ei poista virheitä kokonaan vaan mahdollistaa niiden aiemman havaitsemisen. Tuotannossa käyttöönotetut viat voivat kokonaan estää ohjelmiston käyttämisen työtehtäviin. Tällöin käyttäjän työrutiini keskeytyy ja hänen täytyy palata edelliseen hyväksi tiedettyyn versioon.

3.3 Päivityksen käyttöönotto

Ohjelmistopäivityksen käyttöönottoa itsepäivitystoiminnolla edeltää usea vaihe. Ensin täytyy tarkistaa uusien päivitysten olemassaolo, ladata päivityspaketti ja asentaa se. Itsepäivitystoiminto voi lisäksi automatisoida toimenpiteittä halutun käyttökokemuksen mukaan.

Kun itseään päivittävä ohjelma pitää kirjaa päivitysten tilasta, kaavio sen tiloista voi olla esimerkiksi kuvan 7 mukainen. Ohjelmisto pitää oletusarvoisesti itseään uusimpana saatavilla olevana versiona. Kun päivitysten saatavuustarkistus tehdään, ohjelmisto vaihtaa tilaansa, jos uudempi versio on saatavilla. Jos ohjelmisto on päivitystarkistusta tehdessään jo tietoinen uudesta versiosta, mutta tämä on vedetty pois saatavilta, ohjelmisto siirtyy takaisin pitämään itseään uusimpana versiona. Kun päivitysten olemassaolosta on tieto, ne voidaan ladata ja lataamisen jälkeen asentaa ja ottaa käyttöön. Lataaminen ja asentaminen voidaan tosin kumota poistamalla päivityspaketti ja asennus. Uuden päivityksen käyttöönoton jälkeen ohjelmisto pitää jälleen itseään uusimpana saatavilla olevana versiona.

Asentamisella ja käyttöönotolla on eroa itsepäivittyvän ohjelmiston tapauksessa. käyttöönotto on eriytetty omaksi vaiheekseen, koska itsepäivityksen tapauksessa käynnissä oleva päivityksen suorittava ohjelmisto voi estää käyttöönoton tai ajonaikainen käyttöönotto voi aiheuttaa ei-toivottua käyttäytymistä ajonaikaisessa ohjelmistossa. Käyttöönotto voi estyä esimerkiksi siksi, että operaatio muokkaisi käynnissä olevan ohjelmiston auki pitämiä tiedostoja. Haittaa ajossa olevalle ohjelmalle voi aiheutua esimerkiksi muokkaamalla asetustiedostoja, joiden ajossa oleva ohjelmisto olettaa pysyvän koskemattomina. Asentaminen voi olla esimerkiksi ohjelmiston purkaminen pakatusta tiedostosta ja sijoittaminen haluttuun paikkaan. Käyttöönotto voi olla asennuksen integroiminen käytettäväksi kohdejärjestelmässä luomalla tarvittavia rekisteriavaimia ja pikakuvakkeita.

Aiemmassa työssä [13] kuvatun mukaisesti ohjelmiston päivitysten käyttöönotto on mahdollista suorittaa kokonaan automaattisesti käyttäjän huomaamatta. Työssä kuvataan, kuinka päivitystarkistus, päivityksen lataaminen ja asentaminen voidaan suorittaa taustalla ohjelman normaalin toiminnan aikana. Lisäksi siinä esitetään esimerkki, kuinka liitännäisarkkitehtuurin mukaisen ohjelmiston tapauksessa käyttöönotto voidaan suorittaa ajonaikaisesti vapauttamalla käytössä oleva dynaamisesti linkitetty kirjasto ja lataamalla sen tilalle uusi versio. Kirjaston vapauttamisen ja lataamisen välissä voidaan vaihtaa liitännäistiedosto uuteen tai ladata uusi versio eri sijainnista ja siirtää tiedostot paikoilleen myöhemmin.

Aiemman työn [13] mukaisesti automaattisesti suoritetut operaatiot saattavat kuluttaa kohdejärjestelmän resursseja käyttäjän siitä tietämättä ja näin haitata käyttökokemusta aiheuttamalla muiden toimintojen hitautta. Työssä esitetään suuren päivityspaketin automaattisen lataamisen mahdollisiksi haitoiksi latauskaistan käyttämistä niin, että muu liikenne hidastuu tai että datansiirtorajoitettua yhteyttä käytettäessä rajoitus tulee vastaan. Siinä myös mainitaan päivityspaketin purkamisen käyttämä prosessointikapaisteetti ja tiedostojen siirtelyn käyttämä siirräntäkapasiteetti. Lisäksi työssä mainitaan, että jos päivitysvaiheiden tuloksia jäädään ohjelmistossa odottamaan, aikaa voi kulua ennalta tietämätön määrä. Tämä voi haitata käyttäjäkokemusta tai ohjelmiston toimintaa, jos sillä on reaaliaikavaatimuksia.

3.4 Käyttäjäkokemus päivittämisessä

Sujuva ja turvallinen käyttäjäkokemus on merkittävä tekijä päivitysten käyttöönottokynnykselle. Tätä työtä edeltäneessä työssä [13] käsiteltiin tutkimuksia ohjelmistopäivitysten käyttäjäkokemuksesta [18, 17]. Tutkimuksessaan käyttäjien ohjelmistopäivityksiin liittyvistä uskomuksista Mathur ja kumppanit havaitsivat, että erityisesti menneet kokemukset ohjelmistopäivitysten asentamisen virhetilanteista ja epäonnistumisista ovat syynä siihen, että käyttäjät jättävät päivitykset asentamatta. Toisessa Android-käyttöjärjestelmän päivittämistä koskevassa kyselytutkimuksessa Mathur ja Chetty havaitsivat saman. Edeltäneessä työssä todettiin, että tämän seikan vuoksi päivitykset tulee tehdä vikasietoisesti. Samalla ongelmaksi nostettiin kuitenkin se, että virhetapauksilta ei voida päivitystoiminnallisuuteen toteutetuilla keinoilla kokonaan välttyä, sillä päivitykseen liittyvät olennaisesti ohjelmistosta ulkoiset tekijät. Tällaisia ovat esimerkiksi päivityspalvelimen toiminta, verkkoyhteyden tila matkalla palvelimelle, verkkoyhteyden tila kohdejärjestelmässä ja kohdejärjestelmän tiedostojärjestelmä. Ne voivat ohjelmistosta riippumattomien tekijöiden vuoksi lakata toimimasta tai toimia virheellisesti milloin hyvänsä.

Edeltäneessä työssä esitettiin myös pohdintaa näennäisen satunnaisten virheiden aiheuttamasta käyttökokemuksesta ja sen seurauksista. Työssä esitettiin, että jos virheet ovat käyttäjälle näennäisen satunnaisia, hän ei saa muodostettua toiminnallisuuksista johdonmukaista käsitteellistä mallia, jolloin ohjelmisto toimisi vastoin hänen käsitteellistä malliansa. Tällöin hän kokisi toiminnon käytön epävarmemmaksi ja voisi seurauksena välttää

sen käyttöä.

Ohjelmistopäivitysten käyttäjäkokemusta koskevassa tutkimuksessa Vaniea ja Rashidi [46] selvittivät luomansa kyselyn avulla käyttäjien kokemuksia erilaisten ohjelmistojen päivitysten asentamisesta. Eräänä johtopäätöksenään he mainitsivat päivitysten asentamiseen liittyvän riskin haittaavan päivitysten käyttöönottoa. He havaitsivat käyttäjien turvautuneen järjestelmän palautustoimintoihin, joilla saadaan asennuksen epäonnistuttua palautettua järjestelmä takaisin asennusta edeltäneeseen tilaan. Vaniea ja Rashidi esittävät riskin pienentämiseksi ohjelmistoon toteutettavaa toimintoa, jolla voidaan palata edelliseen toimivaan tilaan virheen sattuessa päivitettäessä.

Ohjelmistopäivitysten vaikutusta päivittyvän ohjelmiston käyttäjälojaaliuteen (*continuance intention*) eli käyttäjien haluun jatkaa ohjelman käyttöä ja palata sen pariin ovat tutkineet Fleischmann ja kumppanit [4]. He havaitsivat, että lojaalius kasvaa, kun ohjelmistopäivitykset parantavat käyttäjille näkyvästi ohjelman toimintoja.

Fleischmannin ja kumppaneiden tutkimuksessa mittarina oli käyttäjien halu jatkaa tutkimuksessa käytetyn ohjelmiston käyttöä, mikä ei suoraan tee heidän havaintojaan sovellettavaksi ammattikäyttäjien ohjelmistoon, jolle käyttäjillä ei ole vaihtoehtoa. Voidaan kuitenkin ajatella ohjelmiston itsepäivitystoiminnallisuuden olevan tässä tapauksessa kohteena käytön jatkamisaikeille, jolloin positiiviset jatkamisaikeet kohdistuvat ohjelmiston päivittämiseen. Tämä lisää käyttäjän halukkuutta käyttää toimintoa ja siten se nopeuttaa päivitysten käyttöönottoa. Samalla tavalla voidaan ajatella, että käyttäjillä on vaihtoehto päivittämiselle eli toiminnallisuuden käyttämättömyys ja vanhassa versiossa pysyminen.

4. OHJELMISTON PÄIVITTÄMISEN TYÖKALUT

Projektin alussa ennen itsepäivitystoiminnon suunnittelemista ja ohjelmistoon toteuttamista suoritettiin taustatutkimus, jossa kartoitettiin sopivia valmiita työkaluja toteutuksen hyödynnettäviksi. Tässä kohdassa arvioidaan itsepäivitystoiminnallisuuden toteuttamiseen sopivia työkaluja ja tekniikoita. Käydään läpi, miten ne soveltuvat käytettäväksi työn kontekstissa ja kohdassa 2.4 asetettujen tavoitteiden täyttämiseen.

Ensin esitellään ohjelmiston ja itsepäivityksen toteutukseen käytetty Qt-ohjelmistokehys kohdassa 4.2 ja sen jälkeen kohdassa 4.1 käydään läpi ohjelmiston toiminnallisuuden käyttämän NuGet-paketinhallintatyökalun soveltuvuusarviointi ja ominaisuudet. Seuraavaksi käsitellään ohjelmistokehitysprosessin palvelinpuolella toteutusta olevat TeamCity ja Nexus Repository manager kohdissa 4.3 ja 4.4. Lopuksi esitellään muut arvioidut työkalut ja niiden soveltuvuusarviot kohdassa 4.5.

4.1 NuGet

NuGet [20] on yleisesti Microsoftin Visual Studio -sovelluskehityksessä käytetty työkalu, jota käytetään avoimesti saatavilla olevien, uudelleenkäytettävien ohjelmistokomponenttien projektiin lisäämiseen. Se osaa ladata ohjelmapaketteja, selvittää ja ladata niiden riippuvuudet ja purkaa nämä paketit projektin tiedostoihin niin, että ne ovat heti käytettävissä projektissa. Sen pääasiallinen käyttötarkoitus on hoitaa Visual Studion paketinhallintaa ja tarjota ohjelmointiyhteisön luomia ohjelmistokomponentteja keskitetystä internetissä olevasta pakettivarastosta.

Paketit noudattavat hyvin dokumentoitua rakennetta. Ne ovat ZIP-pakattuja tiedostoja, jotka on nimetty paketin nimen ja *Semantic Versioning 2.0.0* -muotoisen [29] versionumeron yhdistelmällä. NuGetin yleisimmässä käyttötarkoituksessa paketit sisältävät käännettyjä .NET-kirjastoja yhdelle tai useammalle .NET-ohjelmistokehyksen versiolle ja mahdollisesti näiden lähdekoodit ja symbolitiedostot debuggausta varten. Lisäksi paketeissa on metatietoja varten tiedostoja, jotka kertovat paketin sisällöstä ja sen rakenteesta. Tiedostoissa kerrotaan, mitä toimintoja paketti sisältää ja mitkä tiedostot tarjoavat niistä mitäkin. Näiden tietojen avulla NuGet osaa pakettia purkaessaan asettaa tiedostot oikeille paikoilleen ja muuttaa tarvittavia konfiguraatioita Visual Studion projektissa. Tuki on myös Visual Studion C++-projekteille.

Paketteja voidaan luoda ja muokata millä tahansa ZIP-työkalulla, mutta NuGet tarjoaa siihen oman helppokäyttöisemmän ratkaisunsa. Komentoriviltä on mahdollista luoda haluttu paketti käyttämällä erillistä XML-tiedostoa määrittelemään sen rakenne ja sen sisältämät tiedostot. Pakettivarastona voidaan myös käyttää mitä tahansa paikallista kansiota.

Sen voi luoda manuaalisesti, mutta NuGet tarjoaa senkin luomiseen ja käsittelemiseen komennot.

NuGet on laajalti käytetty ja sillä on hyvin dokumentoitu rajapinta. Komentorivikäytössä NuGet toimii itsenäisesti ajettuna, kunhan kohdejärjestelmässä on asennettuna Microsoftin .NET-kirjaston versio 4.5 tai uudempi. Tämä riippuvuus on hyvin usein valmiiksi tyydytetty, sillä uusissa Windows-käyttöjärjestelmissä kyseinen kirjasto on usein valmiiksi asennettuna. NuGetia voidaan käyttää komentoriviltä kaikkiin sen tarjoamiin toimintoihin. Sillä on mahdollista ottaa yhteys HTTPS:n yli NuGet-pakettivarastoon ja muun muassa listata tarjolla olevat paketit tai asentaa haluttu paketti.

Kohdassa 2.4 asetetun tavoitteen päivitysten nopeasta lataamisesta tyydyttämiseksi suunniteltiin, että komponenteista muodostuva ohjelmisto voidaan jakaa pienempiin paketteihin ja ladata vain tarvittavat osat. Tätä lähestymistapaa käyttäen NuGet oli muihin arvioituihin työkaluihin nähden ylivoimainen, sillä sen ydintoiminnallisuuksiin kuuluu riippuvuuksien selvittäminen. NuGet-paketeille voidaan merkitä luomisen yhteydessä riippuvuuksia toisiin paketteihin, jolloin nämä ladataan ja asennetaan samalla. [21]

NuGetin muita kehittyneitä toiminnallisuuksia ovat siihen sisäänrakennettu salatun yhteyden oletusarvoinen käyttäminen ja autentikaatio. Vaikka molemmat olisivat toteutettavissa Qt:n verkkotyökaluilla, se olisi aikaavievää ja virhealtista verrattuna valmiiseen toteutukseen. Jo pelkkä virhetilanteiden asianmukainen käsittely paisuttaisi toteutuksen työmäärää huomattavasti.

NuGetin merkittävimmäksi heikkoudeksi todettiin, ettei sitä voi C++:ssa käyttää suoraan ohjelmallisesti, vaan sen ympärille tarvitsisi luoda kehysfunktiot kutsumista ja tulosten tarkastelua varten. NuGetin käytöstä päivitystoiminnallisuudessa kerrotaan enemmän kohdassa 5.7 ja virheentarkastelua tulosteista kohdassa 5.8.

Kun NuGet oli todettu kohdassa 4.5 listattuja itsepäivityksen mahdollistavista työkaluja paremmaksi, toteutettiin käytännön soveltuvuusselvitys luomalla prototyyppi itsepäivitystoiminnallisuudesta kohdassa 2.4 mainittuun sitä varta vasten luotuun pieneen ohjelmaan, joka oli arkkitehtuuriltaan varsinaisen ohjelmiston kaltainen. Pieni ohjelma jaettiin komponenteittain NuGet-paketteihin, joiden välille muodostettiin riippuvuussuhteet kuten on kuvassa 11. Luodut paketit lisättiin saataville paikalliseen pakettivarastoon, josta ne oli tarkoitus ladata käyttöönotettavaksi.

Ohjelmallisen kutsumisen toteuttamisen raskauden vuoksi prototyyppi toteutettiin Windowsin BAT-komentojonotiedostoja käyttäen. Komentojonotiedosto kutsui NuGetin komentoja päivitysten lataamiseksi, sen tulosteista luettiin toimintojen vaikutukset. Sitten vanha ohjelma siirrettiin sivuun ja tulosteiden pohjalta siirrettiin ladatut komponentit paikoilleen, jolloin ne olivat käyttöönotetut. Ohjelman uuden version luomisessa muutettujen pakettien ja kaikkien niistä riippuvien pakettien versionumeroa kasvatettiin, jolloin NuGet latasi uudet tarvittavat paketit. Versionumeronsa säilyttäneitä paketteja ei tarvinnut

ladata, koska ne olivat jo ladattuina väliaikaistiedostoissa. Prototyyppi toimi odotetulla tavalla, joten NuGet valittiin käytettäväksi päivitystyökaluksi.

4.2 Qt-ohjelmistokehys

Qt [40] on C++- ja Python-ohjelmistokehys, joka mahdollistaa graafisen käyttöliittymän alustariippumattoman toteutuksen. Kohdan 2.1 mukaisesti kehityksen kohteena oleva ohjelmisto on toteutettu C++:lla ja Qt:lla ja itsepäivitystoiminnallisuus on osa sitä, joten Qt-ohjelmistokehyksen käyttö seuraa luonnostaan. Muita vaihtoehtoja ei harkittu. Jo vaihtoehtojen selvittämisen aloittaminen vaatisi erittäin tärkeän syyn.

Ensisijaisesti Qt näyttäytyy käyttöliittymäkirjastona tarjoten valmiit luokat käyttöliittymäikkunoille ja niiden sisältämille elementeille. Käyttöliittymiä voi suunnitella Qt:n omalla Qt Creator -kehitysympäristöllä [39] visuaalisessa käyttöliittymässä, muokata käsin XML-dokumentista tai asettaa ohjelmallisesti muun koodin seasta. Käyttöliittymäelementtien välille voi myös kaikilla näistä tavoista kytkeä signaaleja, joilla saadaan käyttäjäinteraktiivista toiminnallisuutta aikaiseksi. Kehittyneemmät toiminnot vaativat ohjelmallista signaalien kytkemistä.

Qt:n signaaleihin ja niiden käsittelijöihin perustuva suunnittelumalli [42] mahdollistaa käyttöliittymäelementtien sulavien toimintojen lisäksi helpon asynkronisen suorituksen aikaa vieville operaatioille. Itsepäivitystoimintojen yleensä tarvitsemille operaatioille tiedostojärjestelmän käsittelyssä, verkkoyhteyden muodostamisessa ja aliprosessien luomisessa Qt tarjoaa työkaluiksi luokat QNetworkAccessManager, QDir, QFile ja QProcess [38]. Muiden kuin tiedostojärjestelmäoperaatioiden avulla on mahdollista toteuttaa toimenpiteet asynkronisesti, jolloin käyttöliittymä tai ohjelmiston muut toiminnot eivät häiriinny pitkäkestoisista verkko- tai prosessinsuoritusoperaatioista.

Qt on kypsä avoimen lähdekoodin ohjelmistokehitys ja sitä kehitetään edelleen aktiivisesti The Qt Companyn [41] toimesta. Toteutettavan toiminnallisuuden jatkokehityksen mahdollisia tarpeita varten Qt:n avulla on mahdollista luoda myös kevyt ohjelmiston asennustyökalu, joka lataa asennettavan ohjelman verkosta (*online installer*).

4.3 TeamCity

TeamCity on ohjelmistotuotantotyökalu, joka soveltuu jatkuvan integraation ja jatkuvan toimituksen toteuttamiseen ohjelmistokehitysprosessissa. TeamCity integroi uudet versiohallintajärjestelmään tulleet muutokset ohjelmistoon. Se kääntää ohjelmiston uusillla muutoksilla, ajaa sille testejä ja luo asennuspaketin, jota käytetään sisäisesti uusia toiminnallisuuksia testatessa. TeamCity on kypsä ohjelma ja edelleen aktiivisessa kehityksessä.

Ohjelmistokehitysprosessin nopean palautteen takaisinkytkennän mahdollistamiseksi TeamCity voi ajaa useita käännöksiä samanaikaisesti useilla eri käännöspalvelimilla, joita voidaan lisätä käytön tarpeen mukaan. Eri projekteille ja yhden projektin versionhallinnan

haaroille voidaan luoda omia konfiguraatioita, joilla mahdollistetaan erilaisten käännösasetusten lisäksi muutkin operaatiot kuin käännökset.

TeamCity mahdollistaa muidenkin kuin tavallisten ohjelmiston kääntämiseen ja testaamiseen liittyvien tehtävien luomisen. Sen avulla voidaan automatisoida mitä vain tapahtumasarjoja uudelle integroitavaksi tulleelle muutokselle. Tällä tavalla voidaan esimerkiksi luoda automaattisesti päivityspaketti ja asettaa se saataville.

Kohdan 2.2 mukaisesti TeamCity on jo käytössä ohjelmistotuotantoprosessissa toteuttamassa jatkuvaa toimitusta eli luomassa ohjelmistosta asennuspaketteja. TeamCityn säilyttämistä nykyasemassaan ja sen käyttämistä myös päivityspakettien luomiseen tuki se, että se soveltuu hyvin päivityspakettien luomiseen ja että organisaatiossa on paljon kokemusta sen käyttämisestä.

4.4 Nexus repository manager

Nexus repository manager [31] on pakettivarastoja tarjoava palvelimella ajettava palvelu. Samalla palvelininstanssilla voidaan tarjoilla useita eri pakettivarastoja eri tarpeisiin. Nexus tarjoaa seittikäyttöliittymän pakettivarastojen hallintaa varten ja pitää kirjaa tarjoilemistaan paketeista. Pakettivaraston hallintapaneelista voi esimerkiksi seurata, mitä paketteja pakettivarastoissa on tarjolla ja tarkkailla niiden lataustilastoja.

Nexuksesta pyydettiin sitä kehittävältä yhtiöltä koeversio käyttöön, jotta sen soveltuvuutta voitiin arvioida. Pakettivaraston koeversio asennettiin käyttöön kehityspalvelimelle ja se asetettiin tarjoamaan NuGet-päivityspakettivarastoa. Päivityspakettien lisäämistä pakettivarastoon testattiin komentoriviltä. Hallintapaneelia kokeiltiin pakettien latausmäärien tarkkailuun. Kullekin paketille latauskohtainen loki aikaleimoineen olisi ollut toivottava ominaisuus, mutta sitä ei ollut tarjolla. Muuten hallintapaneeli latausmäärineen tarjosi riittävät tiedot.

Nexus todettiin käyttöön soveltuvaksi ja sitä varten hankittiin lisenssi. Se tukee useiden yleisten pakettivarastojen tarjoamista, joten samalla lisenssillä sen käyttöä voi laajentaa muihinkin tarkoituksiin.

4.5 Muita arvioituja itsepäivitystyökaluja

Projektin alkuvaiheessa etsittiin ja arvioitiin useita itsepäivitystoiminnallisuuden mahdollistavia työkaluja. Osa niistä oli tarkoitettu toteuttamaan itsepäivitystoiminnallisuuden kattavasti ja osa oli muita työkaluja, joita voisi käyttää itse tehdyssä toteutuksessa hyödyksi. Nimenomaisista itsepäivitystyökaluista osa oli omaan ohjelmistoon mukaan liitettäviä ohjelmakirjastoja ja osa kytkeytyi käyttäjän ja ohjelmiston väliin niin, että käyttäjä ohjelmiston sijaan käynnisti päivitysohjelman, joka päivitystarkistustensa jälkeen käynnisti itse ohjelmiston.

Arviointikriteereinä olivat työkalun yleinen soveltuvuus kyseisen ongelman ratkaisemiseen, ohjelmistoon integroimiseen kuluva työmäärä, muokattavuus, ylläpidettävyys ja lisenssiehdot. Soveltuvuudessa ongelman ratkaisemiseen otettiin huomioon kohdassa 2.4 asetetut tavoitteet. Arvioitujen työkalujen vaatimukset, lisenssit ja projektien aktiivisuudet on koostettu taulukkoon 1.

Taulukko 1. *Arvioidut itsepäivitystyökalut ja niiden ei-toiminnalliset arviointikriteerit.*

Työkalu	**Vaatimukset**	**Projektin aktiivisuus**	**Lisenssi**
Chocolatey [2]	–	2011–	Apache 2.0 [37]
Fervor [45]	Qt	2012–2013, ed. 2017	MIT [27]
Fervor autoupdate [44]	Qt	2012–2013	MIT
ClickOnce [19]	WinForms / WPF	2003–	Käyttöoikeus
Omaha [7]	–	2009–	Apache 2.0
QSimpleUpdater [32]	Qt	2015–2016, ed. 2017	DBADPL [35]
QtAutoUpdater [1]	Qt Installer Framework	2016–	BSD3 [26]
Squirrel [33]	–	2014–	MIT
Update-Installer [16]	–	2011, ed. 2014	BSD2 [25]
Winsparkle [30]	–	2010–	MIT

Vaatimuksissa on listattu työkalun vaatimukset käytetystä tekniikasta ohjelmistossa, johon työkalua ollaan soveltamassa.Osalla työkaluista ei ollut lainkaan väliä, millä tekniikalla ohjelmisto oli toteutettu, ja osa oli suunnattu nimenomaan Qt- ja C++-käyttöön. Microsoftin ClickOnce vaatii ohjelmiston toteutukseksi Windows Forms- tai Windows Platform Foundation -tekniikoita, joten se hylättiin yhteensopimattomana. Sillä on myös huono maine [5, 6, 34, 47]. QtAutoUpdater vaatii, että ohjelmisto on alunperin asennettu Qt Installer Framework -työkaluilla. Ohjelmistossa tapahtuva muutos kyseisiin työkaluihin siirryttäessä todettiin liian suureksi, joten QtAutoUpdater hylättiin.

Kirjatut aktiivisuusajat ovat tammikuulta 2018, kun arviointi suoritettiin. Projektit ovat voineet sen jälkeen aktivoitua ja niihin on voinut sen jälkeen tulla uusia päivityksiä. Projektien aktiivisuutta tarkkailtiin projektien kypsyyden ja elinkaaren vaiheen selvittämiseksi. Hyvin uuden projektin työkalu ei välttämättä olisi riittävän suurella varmuudella virheetön. Lisäksi uusi projekti saattaa muuttua merkittävästi, jolloin ylläpito ja korjausten käyttöönotto voivat olla työläitä. Ylläpidettävyysongelmaksi voi myös muodostua vanha projekti, jota ei enää aktiivisesti ylläpidetä.

Avoin lähdekoodi ja salliva lisenssi mahdollistavat työkalun ylläpitämisen ja muokkaamisen itse. ClickOncea lukuunottamatta työkalut ovat avoimen lähdekoodin sallivasti lisensoituja projekteja. ClickOnce on suljettu ja sille on vain käytön salliva lisenssi. Minkään työkalun lisenssin ei katsottu olevan esteenä käytölle.

Vaatimusten lisäksi työkaluilla oli erilaisia riippuvuuksia ohjelmakirjastoihin tai muihin työkaluihin. NuGet-paketteja käsittelevä Squirrel vaatii saman kuin NuGet eli että kohdejärjestelmässä on asennettuna .NET-kirjaston versio 4.5 tai uudempi. Fervor autoupdate vaatii toimiakseen QuaZIP-kirjaston [36], joka on LGPL-lisensoitu. Googlen päivitystyökalusta julkaistu avoimen lähdekoodin versio Omaha puolestaan vaatii oman kehitysympäristön pystyttämisen ja muutoksia lähdekoodiinsa, jotta se voidaan ottaa projektissa

käyttöön. Kehitysympäristöllä on lisäksi lukuisia riippuvuuksia eri työkaluihin. Omaha hylättiin integroimiseen kuluvan suuren työmääränsä vuoksi. riippuvuudet muihin työkaluihin tai kirjastoihin ja

Jäljelle jääneiden työkalujen soveltuvuus toiminnallisiin vaatimuksiin arvioitiin. Taulukossa 2 on listattu arvioidut työkalut, arviointitapa ja hylkäämiseen johtanut syy. Suuri osa työkaluista hylättiin niiden dokumentaatiossa kuvattujen vaatimusten vastaisten toimintatapojen vuoksi. Chocolatey on työkaluna tarkoitettu käyttöjärjestelmätason paketinhallintaan eikä sovellu itsepäivitystyökaluksi. Työkaluista Fervor, Fervor autoupdater, QSimpleUpdater ja Update-Installer yksikään ei sisältänyt kaikkia haluttuja toiminnallisuuksia.

Taulukko 2. *Toiminnallisilta ominaisuuksiltaan arvioidut itsepäivitystyökalut.*

Työkalu	**Hylkäämisen syy**
Chocolatey	Ei ole tarkoitettu itsepäivitystyökaluksi
Fervor	Ei asenna lataamaansa päivitystä automaattisesti
Fervor autoupdate	Ei tue ladattavan päivityspaketin koon minimointia
QSimpleUpdater	Haluaa päivityspakettina asennusohjelman
Update-Installer	Ei lataa päivityspakettia verkosta
Squirrel	Jatkokehitysnäkymät huonommat kuin NuGetilla
Winsparkle	Ei tue ladattavan päivityspaketin koon minimointia

Työkaluja Squirrel ja Winsparkle arvioitiin testaamalla niiden integroimista testiprojektiin. Squirrel lataa ja asentaa NuGet-paketteja ja tukee muutokseen perustuvia (*delta*) päivityspaketteja latauskoon minimoimiseksi. Sen toimintaperiaate on olla ohjelmistoa käynnistäessä käyttäjän ja käynnistettävän ohjelmiston välissä, mikä parantaa yhteensopivuutta eri ohjelmien kanssa mutta vähentää muokattavuutta. Squirrelin ollessa hankalasti muokattavissa ja kykenevä päivittämään ohjelmisto ainoastaan uusimpaan versioon todettiin, ettei se tarjoa NuGetin komentorivikäyttöä enempää hyötyä. Sillä oli sen sijaan enemmän vaatimuksia NuGet-pakettien rakenteelle kuin NuGetin komentorivikäytössä.

Winsparklen integroimista testattiin, vaikka dokumentaation perusteella jo todettiin, ettei se tue päivityspaketin latauskoon minimointia muutokseen perustuvilla päivityspaketeilla tai muulla tavalla. Testaus suoritettiin osin koska Winsparklen C-kielinen dynaamisesti ladattavan kirjaston tuottava toteutus oli vaivaton integroida testiprojektiin ja osin koska tarkoituksena oli selvittää sen käyttö NuGetin kanssa. Winsparklella olisi voitu toteuttaa päivitysten saatavuustarkistus ja käyttöliittymän toiminnot. Testin perusteella havaittiin, että suunniteltu käyttö NuGetin kanssa ei olisi mahdollista eikä käyttöliittymän muokkaaminen noudattamaan ohjelmiston yleistä ilmettä ollut riittävän vaivatonta.

5. TOTEUTETTU ITSEPÄIVITYSTOIMINNALLISUUS

Suunniteltu itsepäivitystoiminnallisuus toteutettiin ohjelmistoon omaksi ohjelmakomponentikseen. Tässä luvussa esitellään sen toimintaperiaatteet ja perustelut toteutustavoille. Ensin käsitellään yleinen arkkitehtuurikuvaus kaikista päivitykseen liittyvistä komponenteista kohdassa 5.1. Sitten esitellään ohjelmiston paketteihin jakamisen ja niiden palvelimelle toimittamisen ratkaisu kohdassa 5.2 ja palvelimen valinta kohdassa 5.3. Ohjelmistossa olevan toteutuksen tekniset yksityiskohdat ja käyttöliittymä päivitysten saatavuuden tarkistamisessa, asentamisessa ja käyttöönotossa käsitellään kohdissa 5.4, 5.5, 5.6 ja 5.7. Lopuksi kuvaillaan virhetilanteiden käsittelyä kohdassa 5.8 ja niistä toipumista kohdassa 5.9.

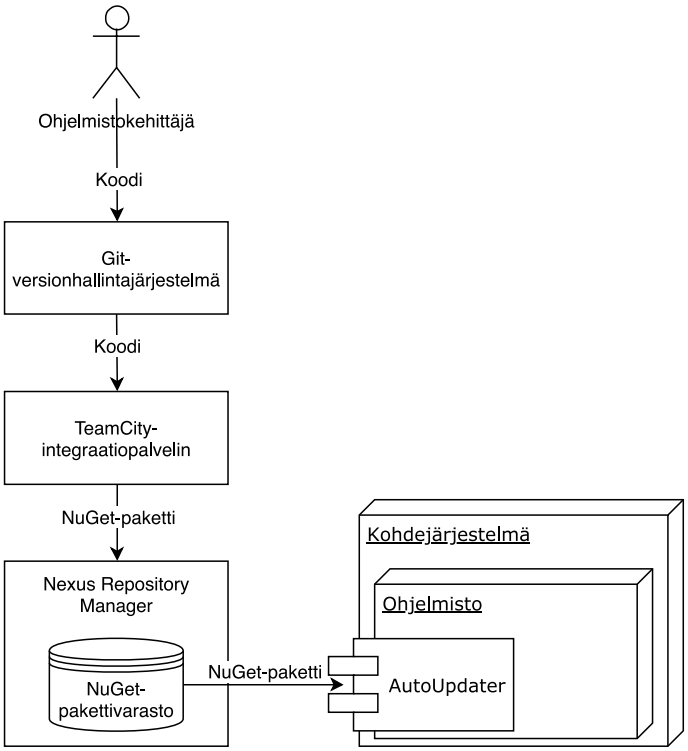
5.1 Arkkitehtuurikuvaus

Koko päivitystoiminnallisuuden toteutus koostuu kahdesta osasta, ohjelmiston itsepäivitystoiminnallisuuden toteuttavasta komponenttista ja ohjelmistotuotantoprosessiin integroidusta päivityspakettipalvelimesta. Näiden välillä kulkee päivityspaketteja palvelimelta ohjelmistoon. Kuvassa 8 on esitetty osat ja niiden kytkeytyminen toisiinsa ja ohjelmistokehitysprosessiin.

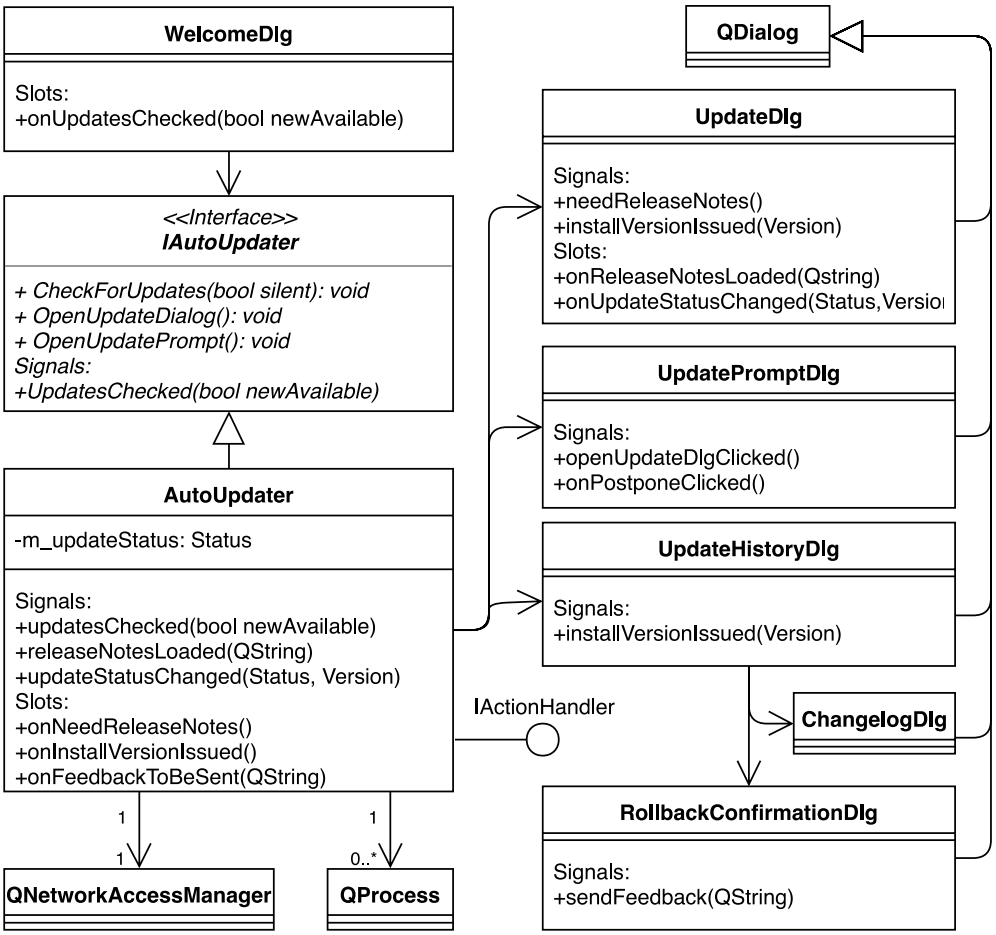
Itsepäivityskomponentin luokkarakenne on havainnollistettu kuvassa 9. Komponentti tarjoaa julkisen rajapintansa kautta toiminnot päivitysten saatavuuden tarkistamiseen ja päivitysikkunan avaamiseen. Ohjelmiston käynnistyksen yhteydessä oleva tervetulonäkymä (kuva 13) kutsuu sitä tarkistamaan päivitykset. Komponentti kommunikoi päivitysten saatavuudesta ja muista tiedoista lähetämällä ja vastaanottamalla Qt:n signaaleja [42]. Valtaosan toiminnoista hoitaa suuri pääluokka `AutoUpdater`, joka toteuttaa toiminnallisuuden julkisen rajapinnan. Muut toteutuksen luokat ovat Qt:n QDialogista [38] periytettyjä käyttöliittymätoimenpiteitä toteuttavia modaalisia ikkunoita. Pääluokka vastaa niiden luomisesta ja tiedon hakemisesta niiden käyttöön.

Olioiden välinen kommunikaatio päivitystoiminnallisuuden sisällä on myös toteutettu Qt:n signaaleilla. Ikkunoiden luomisen yhteydessä yhdistetään signaalit pääluokan instanssin ja ikkunaolion välillä. Lisäksi käyttöliittymäikkunoiden luomisen yhteydessä annetaan rakentajien kautta ikkunoille muuttumattomia tietoja.

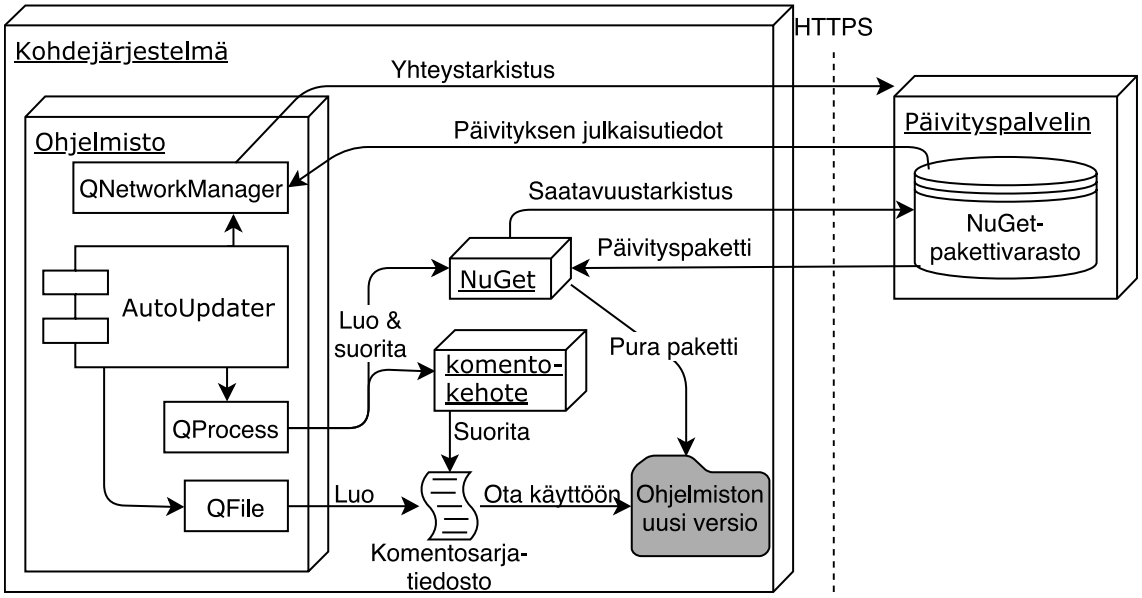
Komponentin kytkeytyminen muuhun ohjelmistoon on luokkaakaaviossa havainnollistettu sillä, että WelcomeDlg käyttää `AutoUpdater`-luokkaa tämän `IAutoUpdater`-rajapinnan kautta. Toimintoja voi käyttää myös ohjelmiston päävalikon kautta. Se on mahdollistettu sillä, että `AutoUpdater` toteuttaa `IActionHandler`-rajapinnan, joka tuo valikkotapahtumat sille käsiteltäväksi.



Kuva 8. *Uuden muutoksen kulku ohjelmistokehittäjiltä käytössä olevaan ohjelmistoon.*



Kuva 9. *Toteutetun AutoUpdater-liitännäisen yksinkertaistettu luokkakaavio.*



Kuva 10. *Toteutetun AutoUpdater-liitännäisen käyttämät Qt-työkalut ja niiden käyttö ulkoisten komponenttien kanssa.*

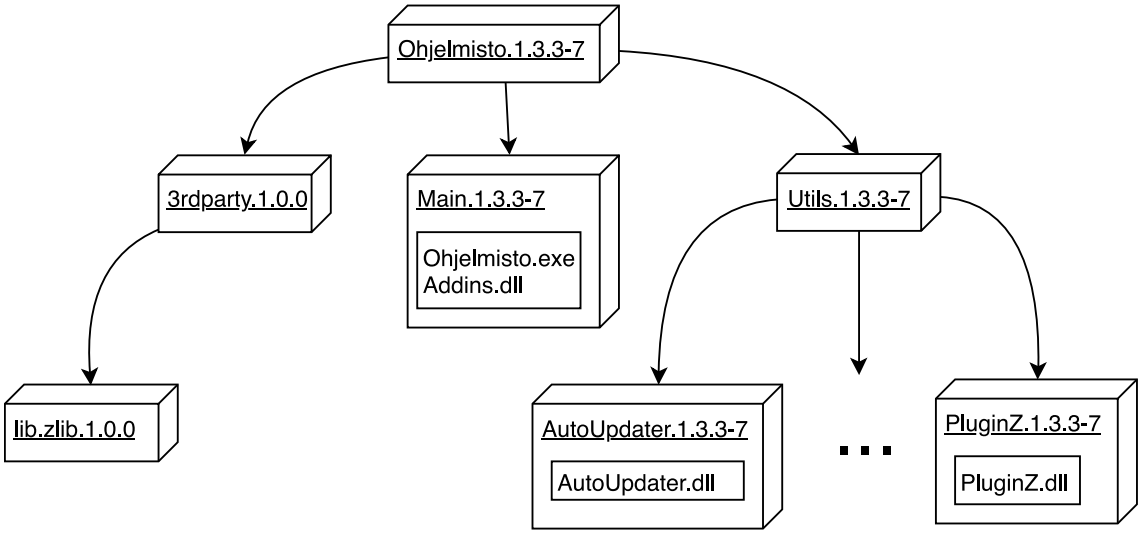
Kuvassa 10 on havainnollistettu kohdissa 5.3, 5.4 ja 5.6 kuvailtu päivityspalvelimen yhteystarkistus, päivityksen saatavuuden tarkistaminen ja niiden lataaminen NuGetin avulla QProcess-luokan kautta sekä lopulta käyttöönotto luomalla uuden version käyttöönottava komentosarjatiedosto ja ajamalla se.

5.2 Päivityspakettivarasto ja päivityspaketit

Päivityspakettivarastona käytetään kohdassa 4.4 tarkemmin kuvattua Nexus Repository Manageria. Järjestely on kuvan 6 mukainen. Ohjelmistokehityspalvelimiin kuuluvalla palvelimella on kehityskäytön Nexus-palvelin, joka tarjoilee NuGet-pakettivarastoa. Sinne saadaan laitettua paketteja saataville suoraan TeamCity-integraatiopalvelimelta. Lisäksi asiakkaan hallinnassa olevalla palvelimella on toinen Nexus-instanssi, joka tarjoilee kahta NuGet-pakettivarastoa. Toinen pakettivarasto on kokeellisille päivityspaketeille ja toinen tuotantokäyttöön tuleville.

Ohjelmiston päivityspaketteja syntyy joko paikallisesti luotuna kohdassa 2.2 kuvatulla julkaisun luomiseen tarkoitetulla ohjelmalla ohjelmiston tavanomaisen asennuspaketin luomisen yhteydessä tai TeamCityn luomana. Kuvan 8 mukaisesti TeamCity luo lähes jokaisesta koodimuutoksesta päivityspaketit.

Paketteja julkaistaan asiakkaan pakettivarastoihin ainoastaan erikseen sovitusti ja suunnitellusti. Kehityskäytön pakettivarastoon paketteja voi kuka tahansa kehittäjä julkaista ajamalla TeamCity-integraatiopalvelimella olevan julkaisua varten luodun tehtävän. Tehtävä ajetaan halutulle TeamCityn luomalle versiolle ohjelmistosta, minkä jälkeen kyseinen versio on kehitysorganisaation saatavilla itsepäivitystoiminnallisuuden kautta.



Kuva 11. *Yksinkertaistettu esimerkki ohjelmiston jakautumisesta toisistaan riippuviin NuGet-paketteihin.*

Ohjelmisto on jaettu kuvan 11 mukaisesti useampaan NuGet-pakettiin niin, että kussakin paketissa on yksi ohjelmiston komponentti. NuGet-paketit on jaettu riippuvuussuhteita kuvaaviin metapaketteihin ja ohjelmakomponentteja sisältäviin sisältöpaketteihin. Metapaketeissa ei ole sisältöä eikä sisältöpaketeissa riippuvuuksia. Paketit on jaettu hierarkisesti samalla periaatteella kuin komponentit kuvassa 2. Saman komponenttiryhmän komponenteista luodut NuGet-paketit ovat riippuvuuksina koko komponenttiryhmän abstrahoivalle metapaketille.

Koska ohjelmiston versionumero on monimutkainen ja sisältää myös automaatioalustan versionumeron, se ei ole yhteensopiva NuGet-pakettien *Semantic Versioning 2.0.0* -versionumeroinnin kanssa. Tämän vuoksi ohjelmistoon toteuttiin päivitystoiminnallisuuden käyttöön oma erillinen versionumero, joka sisältää alkuperäisen monimutkaisen versionumeron *Semantic Versioning 2.0.0* -yhteensopivalla tavalla ja lisää siihen perään edelleen yhteensopivalla tavalla juoksevan numeroinnin.

Kun jotakin komponenttia päivitetään, sen juoksevaa versionumeroa kasvatetaan. Komponentin versionumeroa kasvatettaessa sen vanhemman riippuvuus päivitetään koskemaan tätä uutta kasvatettua versiota. Samalla myös kasvatetaan vanhemman versionumeroa ja toistetaan tämä vaihe kaikille riippuvuuspuun komponenteille. Tällä tavalla versionumeropuun lehdestä lähtenyt muutos kulkee koko matkan aina päätason pakettiin asti ja kasvattaa sen versionumeroa. Metapakettien avulla saadaan minimoitua latauskoko tapauksissa, joissa vain osa ohjelmistoa on päivittynyt. Kun edellä kuvatulla tavalla korotetaan kaikkien esivanhempien versionumeroa, nämä paketit tulevat ladattaviksi päivityksiä ladatessa.

Komponenttiin kuuluvat tiedostot ovat paketin sisällä hakemistopuussa samalla tavalla kuin ne sijaitsevat toimivassa ohjelmistossa. Paketin juuressa on kansio `lib`, jossa kaikki ohjelmiston tiedostot sijaitsevat. Metatietoa sisältävät tiedostot ovat myös paketin juures-

sa. Kun kaikki ohjelmiston eri paketit puretaan ja niiden sisällöt siirretään samaan juurihakemistoon, muodostuu toimiva versio ohjelmistosta.

5.3 Päivityspakettivaraston valinta

Päivitystoiminnallisuudella voidaan ladata päivityksiä useasta eri NuGet-päivityspakettivarastosta, jotka voivat sijaita usealla eri palvelimella. NuGetista on kerrottu tarkemmin kohdassa 4.1 ja NuGet-pakettivarastoista kohdassa 5.2. Tässä kohdassa kuvaillaan, miten valinta tehdään prioriteettijärjestyksen, verkkoyhteystestauksen ja käyttäjien tunnuksiin liittyvien käyttöoikeusryhmien mukaan. Lisäksi kerrotaan, miten päivitysten julkaisemisen vaiheistus on toteutettu eri pakettivarastojen avulla.

Ohjelmistossa on rajattu pääsy eri pakettivarastoihin tietyille käyttäjäryhmille. Käyttäjät ovat ohjelmistoa käyttäessään kirjautuneena siihen omilla käyttäjätunnuksillaan. Tunnuksiin on useita erilaisia käyttöoikeusryhmiä, joihin käyttäjät voivat kuulua. Ohjelmistossa voidaan tarkastella, mihin kaikkiin ryhmiin sitä käyttävä käyttäjä kuuluu. Päivitystoiminnallisuutta varten tunnuksiin on luotu oma erillinen käyttöoikeusryhmänsä. Tällä tavoin mahdollistetaan päivitysten julkaisun vaiheistaminen niin, että uusi päivitys ei tule kerralla kaikkien käyttäjien saataville vaan voi olla ensin pienemmän käyttäjäryhmän käytössä.

Päivitysten julkaisemisen vaiheistamista varten käytössä on kolme pakettivarastoa kuvan 6 mukaisesti. Ohjelmiston kehittäjille on oma kehitysorganisaation sisäisellä palvelimella sijaitseva pakettivarastonsa, jonka käyttöön oikeuttaa käyttäjätunnuksen kuuluminen ohjelmistokehittäjien käyttäjäryhmään. Sisäiseen pakettivarastoon voidaan laittaa uusia päivityksiä suoraan asennuspaketteja automaattisesti luovalta palvelimelta. Erikseen päivitysten testausryhmään valituilla asiakkaan työntekijöillä on edellisessä kappaleessa kuvattu erillinen käyttöoikeusryhmänsä, joka oikeuttaa käyttämään asiakkaalle suunnattua pakettivarastoa. Se tarjoilee etukäteen päivityspaketteja, jotka on tarkoitus tarjota myöhemmin yleisesti kaikille käyttäjille. Lisäksi on kaikille käyttäjille avoin pakettivarasto, joka tarjoilee virallisesti hyväksyttyjä päivityksiä.

Ohjelmisto valitsee käytettävän päivityspakettivaraston ennen kuin se tarkistaa päivitysten saatavuuden. Se käy kaikki ohjelmistoon määritetyt päivityspakettivarastot läpi prioriteettijärjestyksessä. Järjestys noudattaa edellä kuvailtua eli ensin on sisäisen käytön pakettivarasto, sitten rajatun käyttäjäjoukon ja lopuksi kaikille avoin. Jos käyttäjän käyttäjätunnuksessa on oikeudet vuorossa olevaan päivityspakettivarastoon, tämä lisätään oikeutettujen listalle. Päivityspakettivarastoja käytetään aina prioriteettijärjestyksessä ensinnä olevaa, jos se on mahdollista.

Kun ohjelmisto tietää, mihin kaikkiin päivityspakettivarastoihin käyttäjällä on oikeudet olla yhteydessä, se tarkistaa verkkoyhteyden toimivuuden kyseisiä päivityspakettivarastoja tarjoaville palvelimille. Yhteystarkistuksella selvitetään, että palvelimeen saadaan muodostettua yhteys ja että palvelin tarjoaa haetun URL-osoitteena takana NuGet-pakettivarastoa. Tarkistus tehdään ennen päivitysten saatavuustarkistusta ja erillään siitä, koska

saatavuustarkistus suoritetaan NuGet-ohjelmalla, jota ajetaan ohjelmistosta erillisenä prosessina kohdassa 5.7 kuvaillulla tavalla. NuGetilla suoritettavaan päivitysten saatavuustarkistukseen ei voida määrittää omaa haluttua rajaa yhteyden aikakatkaisuun. NuGetia käytettäessä aikakatkaisun raja on useita minuutteja, mikä on käyttäjän odotuttamisen kannalta liikaa. Lisäksi NuGetin tulosteista on vaikeaa selvittää mahdollisten yhteysongelmien syitä.

Yhteystarkistus päivityspakettivarastoihin toteutetaan Qt:n verkkotyökaluilla. Ohjelmistossa on tiedossa kaikkien päivityspakettivarastojen URL-osoitteet. Kaikkien yhteystestattavana olevien päivityspakettivarastoihin lähetetään QNetWorkManager-luokan avulla GET-pyyntö yhdellä kertaa ja vastaukset käsitellään asynkronisesti. Saadun vastauksen tai yhteyden aikakatkaisun perusteella merkitään testin tulokseksi päivityspakettivaraston saavutettavuus. Jos vastaukseksi saatiin HTTP-tilakoodi ja se ei ollut `404 not found`, niin päivityspakettivarasto tulkitetaan saavutettavaksi. Muut tilakoodit virhekoodit mukaanlukien lasketaan yhteystarkituksen onnistumiseksi. Jos puolestaan vastauksena on HTTP-tilakoodi `404 not found` tai yhteys aikakatkaistiin, päivityspakettivarasto merkitään saavuttamattomaksi.

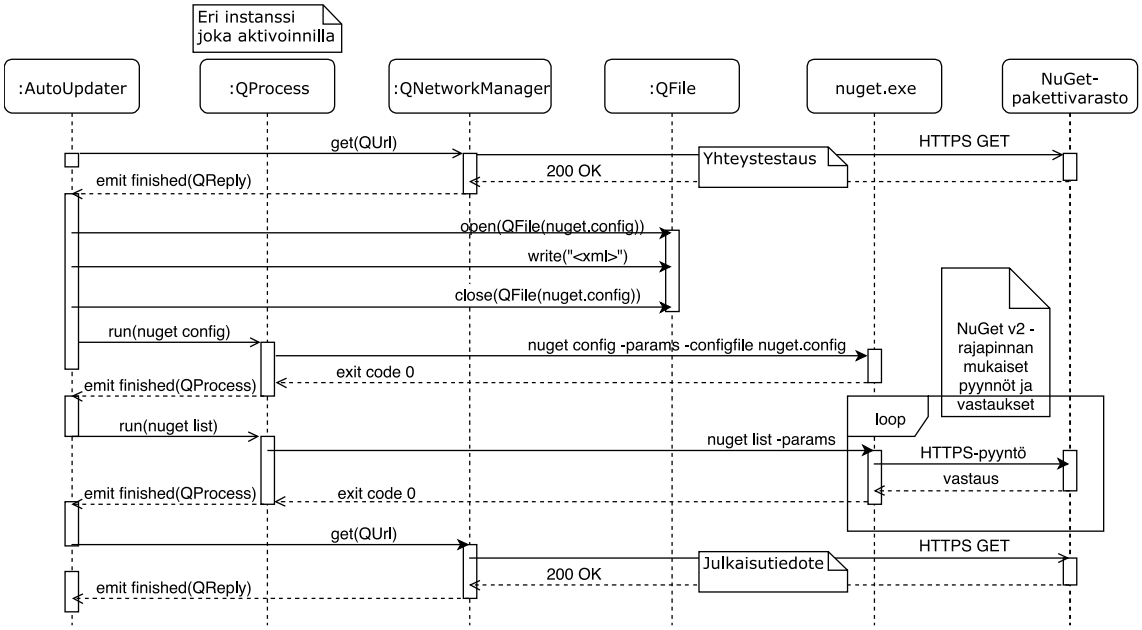
Yhteystarkistuksen edetessä ja saatuja vastauksia käsiteltäessä otetaan huomioon myös päivityspakettivarastojen keskinäinen prioriteetti. Jos yhteystesti tuotti positiivisen tuloksen ja päivityspakettivarasto on prioriteettilistalla ensimmäinen saavutettavissa oleva päivityspakettivarasto eikä sitä ennen ole tarkistamattomia, se valitaan käytössä olevaksi. Tällöin yhteystestaus muihin päivityspakettivarastoihin keskeytetään ja suoritetaan kohdassa 5.4 kuvattu päivitysten saatavuustarkistus käyttäen juuri valittua päivityspakettivarastoa kohteena.

Jos yhteystarkistuksessa ei saada yhteyttä yhteenkään päivityspakettivarastoon, itsepäivitystoiminnallisuus ei voi tehdä mitään. Virheilmoitusta ei näytetä, mikäli yhteystarkistus suoritettiin automaattisesti ohjelmiston käynnistyksen yhteydessä. Ohjelmiston normaalin käytön kannalta ei ole oleellista, saadaanko päivityspalvelimiin yhteyttä. Jos yhteystarkistus suoritettiin käyttäjän aloitteesta eli päivitysten saatavuustarkistuksen yhteydessä, näytetään virheilmoitus. Ilmoituksen lisätiedoissa listataan jokainen testattu palvelin verkko-osoitteineen ja kerrotaan kullekin syy testin epäonnistumiseen. Tällä tavalla käyttäjän on mahdollista selvittää, saako kohdejärjestelmästä ylipäätään yhteyden palvelimille vai onko vika ohjelmistossa.

5.4 Päivitysten saatavuuden tarkistaminen

Uusien päivitysten saatavuus tarkistetaan ohjelmiston käynnistyksen yhteydessä automaattisesti ja kun käyttäjä valikosta erikseen valitsee. Tässä kohdassa käsitellään päivitystarkistuksen vaiheet yhteystarkistuksen jälkeen. Käyttöliittymän osuus päivitystarkistuksesta on selitetty kohdassa 5.5.

Päivitysten saatavuustarkistuksen kulku on esitetty tapahtumasekvenssinä kuvassa 12.



Kuva 12. *Yhteystestauksen, päivitysten saatavuustarkistuksen ja julkaisutietojen lataamisen tapahtumasekvenssikaavio.*

Kuvassa on yksinkertaistettu yhden onnistuneen päivitystarkistuksen kulku, jossa ei oteta huomioon virheistä johtuvia sivuhaaroja. Päivitystarkistus alkaa, kun yhteystarkistus on suoritettu. Ensin luodaan QFile-luokan avulla tyhjä NuGet-konfiguraatiotiedosto, johon sen jälkeen luodaan NuGetin komennolla `config` valittua pakettivarastoa käyttävä konfiguraatio. Tämä vaihe ja muut NuGetin käytön yksityiskohdat on kerrottu tarkemmin kohdassa 5.7.

Uusien päivitysten olemassaolon tarkistus suoritetaan vertailemalla saatavilla olevien päivitysten versionumeroita käytössä olevan ohjelman versionumeroon. Päivitystarkistus suoritetaan ajamalla NuGet-komento `list` ja käyttämällä juuri luotua konfiguraatiotiedostoa. Komennolle annetaan parametriksi ohjelmiston pääpaketin nimi, jolloin se listaa kaikki palvelimella olevat pääpaketin versiot. Ennen versionumeroiden vertaamista kaikista saaduista versioista suodatetaan pois ne, joiden versionumero eroaa perusosaltaan käytössä olevan ohjelmiston versiosta. Tällä tavalla sallitaan ainoastaan ohjelmiston päivittäminen saman version sisällä, kunnes itsepäivitykset laajennetaan joskus koskemaan kaikkia päivityksiä.

Kun uusien päivitysten saatavuudesta on tieto, päivitetään komponentin sisäinen tila vastaamaan juuri tarkastettua ja lähetetään signaali `UpdatesChecked(bool)`, missä totuusarvo bool kertoo, oliko uusia päivityksiä saatavilla. Ennen kuin tarkistuksia on saatu tehtyä komponentti olettaa, että käytössä on ohjelmiston uusin versio. Jos uusia päivityksiä havaitaan olevan saatavilla tilaa muutetaan.

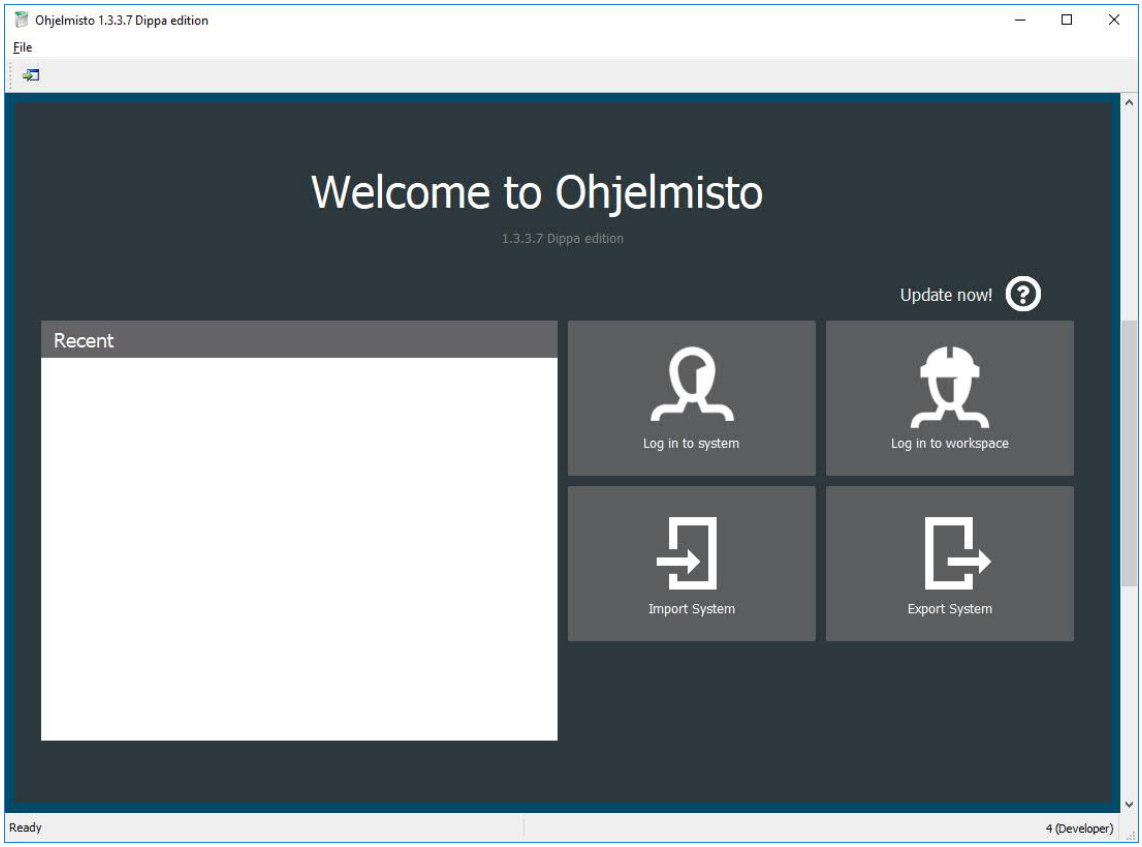
5.5 Päivitysten tarjoaminen käyttäjälle

Kuten aiemmin kohdassa 2.1 on kerrottu, ohjelmistoa käyttää suljettu käyttäjäkunta, jolloin sillä ei ole erityisen suurta levinneisyyttä eikä sen päivitysten mahdollisilla tietoturvakorjauksilla ole siten samanlaista tärkeyttä kuin esimerkiksi suosituimpien käyttöjärjestelmien tietoturvakorjauksilla. Tästä syystä käyttäjiä ei erityisesti painosteta päivitysten asentamiseen. Lisäksi ohjelmisto on työkaluna ainoastaan ammattikäyttäjillä, joiden työn tuottavuus on erityisen tärkeää. Päivityksistä ei saa koitua yllättävää odottelua tilanteeseen, jossa se ei ole toivottavaa. Sen vuoksi päivityksiä ei ladata eikä asenneta automaattisesti. Jos käyttäjä valitsee päivityksen asennettavaksi, ohjelmisto hoitaa sen käyttöönottoon tarvittavat vaiheet automaattisesti ja lopuksi ohjelmisto uudelleenkäynnistetään uuteen versioon.

Kun ohjelmisto käynnistymisensä yhteydessä havaitsee uuden päivityksen olevan tarjolla, se tarjoaa käyttäjälle päivitystä asennettavaksi. Ohjelmistoa käynnistettäessä käyttäjälle avautuu ensimmäisenä kuvassa 13 oleva tervetulonäkymä. Tässä näkymässä käyttäjä kirjautuu johonkin järjestelmään sisään voidakseen käsitellä sitä. Vasemmalla on lista viimeisimmistä käytetyistä järjestelmistä ja oikealla painikkeet useimmin käytetyille toiminnoille. Näiden painikkeiden yläpuolella ovat napit *Update now!* ja ympyröity kysymysmerkki. *Update now!* ilmestyy näkyviin päivitystarkistuksen jälkeen, jos uusi päivitys on saatavilla. Sen klikkaaminen avaa kuvassa 14 näkyvän päivitysdialogin. Ympyröidyn kysymysmerkin klikkaaminen avaa kuvassa 15 näkyvän ohjelmiston uusista ominaisuuksista tarkemmin kertovan *What's new* -dialogin.

Kuvan 14 päivitysdialogissa kerrotaan päivitysten saatavuuden tila. Päivitysdialogin saa auki myös ohjelmiston päävalikosta valikkopolun *Help* → *Check for Updates* kautta. Jos nykyistä käytössä olevaa versiota uudempi versio on valitulla päivityspalvelimella saatavilla, siitä kerrotaan dialogin yläosan tekstissä. Tämä tilanne näkyy kuvassa 14a. Jos uusia päivityksiä ei ole saatavilla, dialogi on kuvan 14b mukainen. Dialogissa tarjotaan asennettavaksi vain uusinta saatavilla olevaa päivitystä. Käytössä olevan version ja tarjottavan päivityksen välissä saattaa siis olla useampi versio, jotka eivät ole tämän dialogin kautta enää asennettavissa. Uusimmasta versiosta näytetään HTML-muotoinen julkaisutiedote, jossa kerrotaan uusimman version tärkeimmät muutokset lyhyesti. Tämän on tarkoitus toimia käyttäjän apuna, jotta käyttäjä voi nopeasti arvioida, haluaako hän perehtyä päivitykseen tarkemmin vai jättää päivityksen välistä.

Päivitysdialogissa on tietosisällön lisäksi kaksi toimintoa. Siinä voi suorittaa päivitysten saatavuustarkistuksen uudestaan ja aloittaa päivitysten lataamisen ja asentamisen. Kuvassa 14a oikealla alalaidassa oleva nappi *Install update* näkyy ainoastaan silloin, kun uusi päivitys on saatavilla. Sitä painettaessa aloitetaan päivitysten lataaminen ja asentaminen, joka on tarkemmin kuvailtu kohdassa 5.7. Aloitettaessa lataaminen ja asentaminen päivitysdialogin päälle avautuu latausdialogi. Painamalla kuvassa 14 vasemmassa alalaidassa näkyvää nappia *Recheck* suoritetaan pävitysten saatavuustarkistus. Tällöin päivitysdialogin päälle avautuu latausdialogi saatavuustarkistuksen ajaksi. Kun tarkistus on suoritettu,

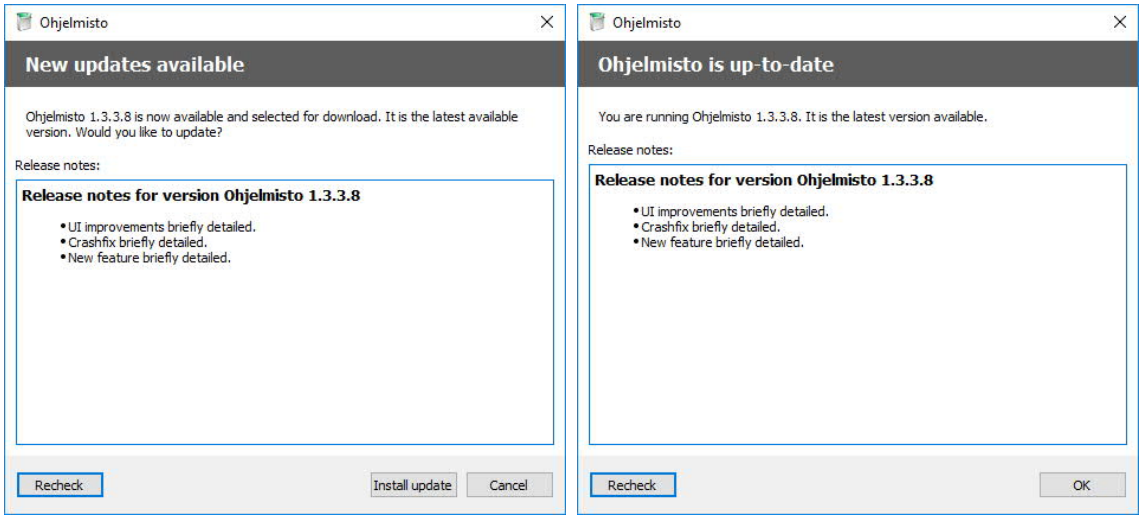


Kuva 13. *Ohjelmiston käynnistymisen yhteydessä avautuva tervetulonäkymä.*

jos päivitysten saatavuuden tila muuttui, päivitysdialogin saatavuustiedot ja julkaisutiedote päivitetään vastaamaan uutta tilannetta. Tässä yhteydessä ladataan päivityspalvelimelta uusi julkaisutiedote.

Uudelleentarkistustoiminto on olemassa kahdesta syystä. Tekninen syy on se, että uuden päivityksen julkaisutietojen lataaminen saattaa epäonnistua ohjelmistosta riippumattomista syistä tai hetkellisen häiriön takia. Tällöin julkaisutietojen tilalla lukee, ettei niitä saatu ladattua ja että päivitysten saatavuuden uudelleentarkistaminen saattaa auttaa.

Toinen syy päivitysten saatavuuden uudelleentarkistustoiminnolle on vähemmän tekninen ja koostuu useammasta tekijästä. Päätekijä on, että käyttäjän tulee voida varmistua siitä, että päivitysdialogin tiedot ovat ajan tasalla. Ohjelmisto voi olla yhtäjaksoisesti käynnissä useita päiviä, ja tällä välillä voi tulla uusia päivityksiä saataville. Ohjelmisto suorittaa päivitysten saatavuustarkistuksen käynnistymisensä yhteydessä ja säilyttää tässä tarkistuksessa saatua tietoa. Tästä on kerrottu enemmän kohdassa 5.4. Tämän vuoksi päivitysdialogi voi näyttää vanhaa tietoa. Käyttäjä voi myös tietää uuden päivityksen saatavuudesta, jos hän on kuullut siitä muuta kautta. Tällöin päivitysdialogi näyttäisi väärää tietoa antamatta mahdollisuutta tehdä uudelleentarkistusta antamiensa tietojen paikkaansapitävyyteen. Jos ohjelmistossa ei olisi mahdollisuutta ajon aikana tarkistaa päivitysten saatavuutta uudelleen, uusi tarkistus vaatisi ohjelmiston uudelleenkäynnistämisen. Se on raskas ja käyttäjän toimet keskeyttävä toiminto, ja lisäisi kynnystä tarkistaa päivitys-



(a) *Uusia päivityksiä on saatavilla.* ***(b)*** *Käytössä on uusin versio.*

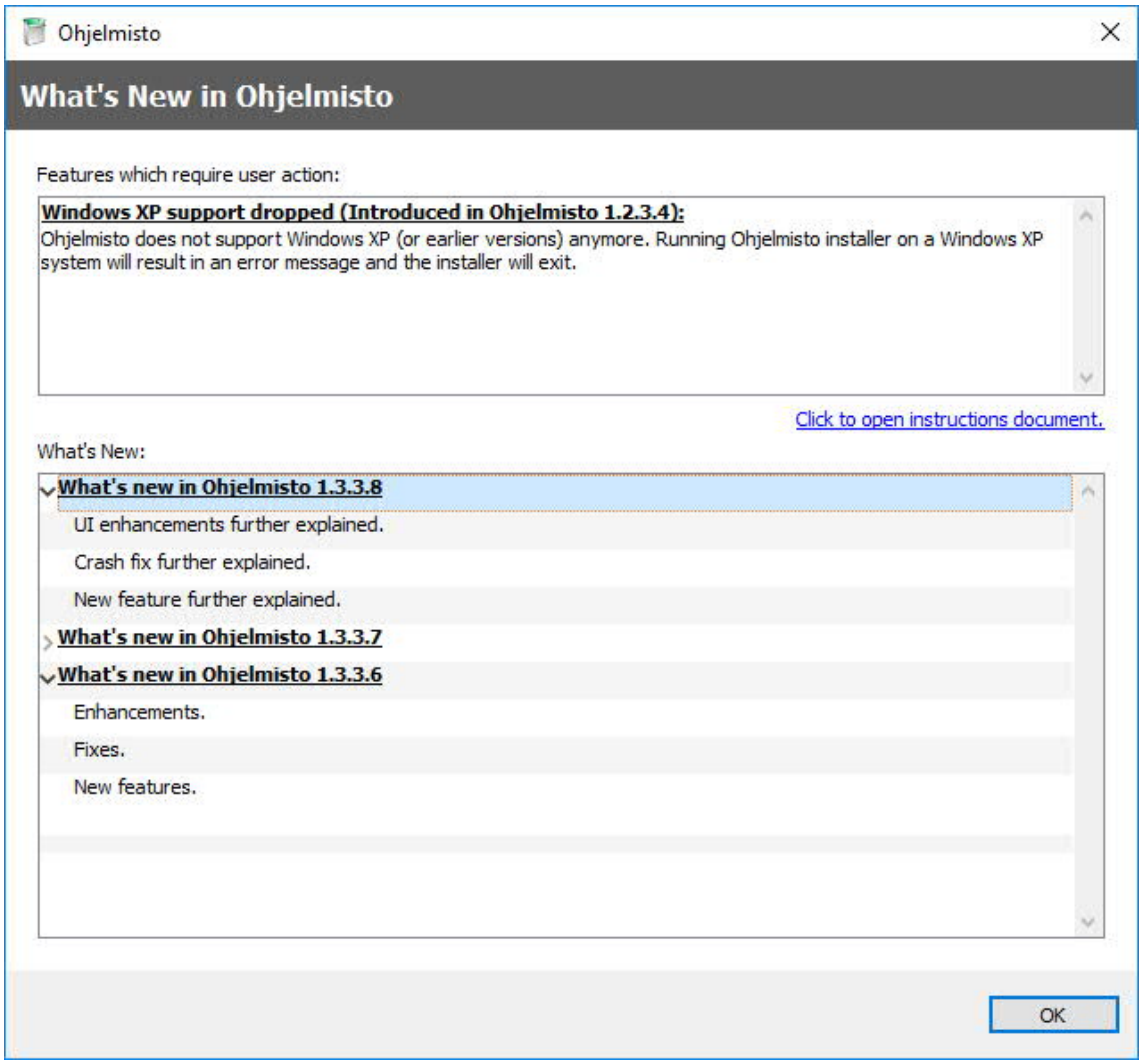
Kuva 14. *Päivitysdialogi.*

ten saatavuus. Viimeinen tekijä on käyttöliittymälooginen. Koska päivitysdialogiin pääsee myös painamalla valikosta nappia *Check for Updates*, sen tulee tarjota napissa kuvailtu toiminto, eli sen kautta tulee olla mahdollista suorittaa päivitysten saatavuustarkistus.

Kuvassa 15 näytetään kaksi listaa, joissa on tietoja ohjelmistoon tulleista muutoksista. Ylempi listaa käyttäjiltä vaadittavia toimia, jotka johtuvat ohjelmiston muutoksista. Nämä ovat harvemmin päivittyviä tietoja eivätkä ne oleellisesti liity aiheena olevaan päivitystoimintoon. Alemmassa listassa kerrotaan ohjelmistoon päivitysten mukana tulleista muutoksista tarkemmin. Lista on puumallinen niin, että kukin ohjelmistoon tullut päivitys on oma suljettava ja laajennettava kohtansa, jonka alla on useita yksittäisiä alikohtia. Nämä alikohdat kertovat kukin yhdestä muutoksesta ja sen vaikutuksesta ohjelmiston käyttöön. Niissä voidaan kertoa esimerkiksi korjatuista vioista, parannuksista toimintoihin ja uusista ominaisuuksista. Yksityiskohdat ovat tarkempia kuin kuvassa 14 olevat julkaisutiedot, mutta ne eivät ole teknisiä vaan keskittyvät käyttäjän kokemiin parannuksiin ja muutoksiin.

5.6 Päivitysten asentaminen ja käyttöönotto

Päivityksen lataaminen ja asentaminen tapahtuu NuGet-ohjelmalla, jonka käytöstä on kerrottu tarkemmin kohdassa 5.7. Kohdan 5.5 mukaisesti aloitettu asennus aiheuttaa signaalin lähtemisen käyttäjän painamasta napista. Signaali on kytkeytynyt auki olleen käyttöliittymäikkunan kautta `AutoUpdater`-luokan käsittelijäfunktioon, joka aloittaa asentamisen. NuGet-ohjelmaa ajetaan komennolla `install` ja argumentiksi annetaan signaalin mukana kulkenut tieto asennettavasta ohjelmiston versiosta. Tämän version saatavuus on tarkistettu jo aiemmin kohdassa 5.4. Lataamisen ajaksi avataan modaalinen latausikkuna, josta käy ilmi ohjelman lataavan päivityksiä ja jossa olevalla napilla *Cancel* asentamisen voi peruuttaa.



Kuva 15. *Ohjelmiston uusista ominaisuuksista tarkemmin kertova What's new -dialogi.*

```
1  connect(QCoreApplication::instance(), &QCoreApplication::aboutToQuit, [=]{
2     QFile scriptFile("autoupdater.bat");
3     QFileInfo fileInfo(scriptFile);
4     scriptFile.open(QFile::WriteOnly);
5
6     QTextStream file(&scriptFile);
7     file << "script␣content␣here";
8     scriptFile.close();
9
10    QStringList args = QStringList() << "/C" << fileInfo.absoluteFilePath();
11    QProcess::startDetached("cmd.exe", args);
12 });
```

Listaus 1. *Päivityksen käyttöönottavan komentosarjatiedoston luominen ja ajaminen ohjelmiston sulkemisen yhteydessä.*

Kun NuGet on ladannut ja asentanut ohjelmiston uuden version, luetaan NuGetin tulosteet läpi ja luodaan niiden perusteella taulukko kaikista asennetuista paketeista. Tämän jälkeen uusi päivitys on valmiina käyttöönotettavaksi. Käyttäjältä kysytään, haluaako hän käynnistää ohjelmiston uudestaan ottaen samalla uuden version käyttöön. Jos käyttäjä haluaa, ohjelmisto suljetaan ja aloitetaan seuraavaksi kuvailtu käyttöönotto. Jos käyttäjä ei halua, hän voi jatkaa ohjelmiston käyttöä normaalisti. Päivitystoimnnallisuuden avulla ei kuitenkaan enää voi asentaa päivityksiä, koska asennettu päivitys odottaa jo käyttöönottoa. Kun käyttäjä sulkee ohjelmiston, uusi versio otetaan käyttöön.

Käyttöönotto sisältää päivityksen siirtämisen ohjelmiston nykyisen version tilalle, nykyisen version siirtämisen pois alta talteen ja uuden version käynnistämisen. Koska ohjelmisto itse hoitaa päivityksen, se on käynnissä samaan aikaan, kun käyttöönoton tulisi tapahtua. Tämä aiheuttaa ongelman, koska ohjelmiston ohjelmatiedostoa ei voida siirtää tai korvata sen ollessa ajossa. Ongelma on ratkaistu listauksen 1 mukaisella tavalla. Ratkaisussa käynnistetään ohjelmiston sulkemisen yhteydessä ohjelmistosta irrallinen prosessi, joka suorittaa tiedostojen siirtämisen ja käynnistää uuden version. Eli kun ajossa olevan ohjelmiston itsepäivitystoiminnallisuus on saanut uuden version asennettua, kytketään oma käsittelijäfunktio Qt:n signaaliin, jonka Qt lähettää ohjelmiston sulkeutumisen yhteydessä. Funktio luo päivityksen yhteydessä asennettujen pakettien perusteella listauksen 2 mukaisen skriptitiedoston ja Qt:n QProcess-luokan avulla erillisen prosessin, joka ajaa juuri luodun skriptitiedoston.

Kuvassa 16 on esitetty itsepäivitystoiminnallisuuden ohjelmiston asennuskansioon luoma kansio väliaikaistiedostoille ja varmuuskopioille. Kansiossa on kaikki NuGet-ohjelmalla ladatut ja asennetut paketit sekä alkuperäisinä NuGet-paketteina että purettuina kansioina. `lib`-kansioiden alla on ohjelmiston kansiohierarkian mukaisesti alikansioissaan kaikki kunkin pakettiin kuuluvan komponentin tiedostot. Lisäksi edellisille käytössä olleille versioille on kansio `rollback`, johon ne säilötään versionumeron mukaisesti nimettyi-

```
1  :loop
2  jos prosessia ei ole olemassa, goto :ok
3  echo odotetaan ohjelmiston sulkeutumista...
4  sleep 1
5  goto :loop
6
7  :ok
8  kopioi asennetut paketit
9  siivoa turhat tiedostot
10 kaynnista kayttoonotettu versio
11 sulje komentorivi-ikkuna hiljaisesti
```

Listaus 2. *Päivityksen käyttöönottava ohjelmiston sulkemisen yhteydessä ajettava komentosarjatiedosto pseudokoodilla ilmaistuna.*

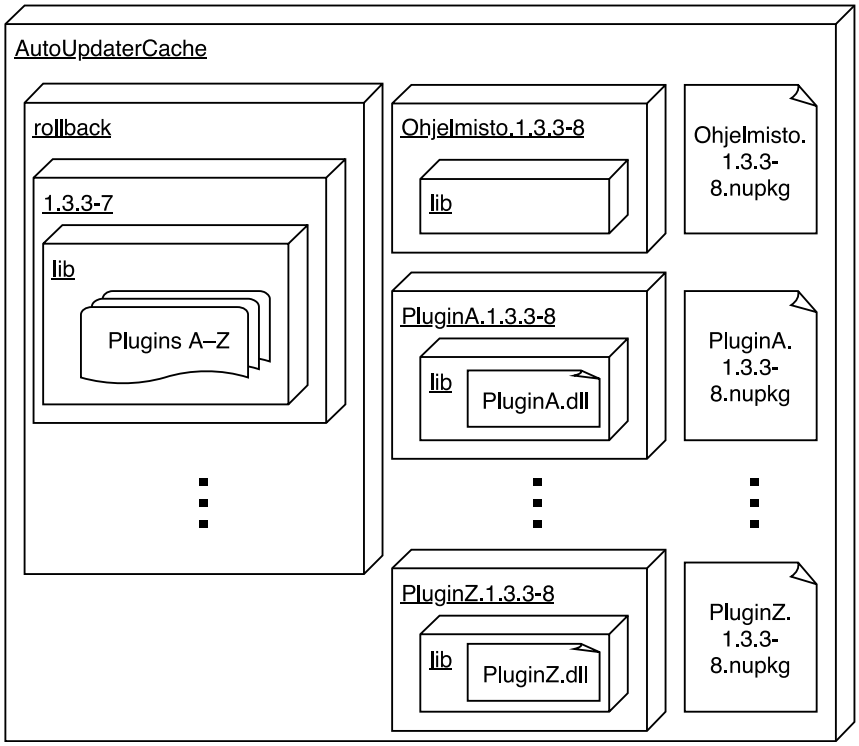
hin alikansioihin.

Käyttöönoton suorittavan listauksen 2 komentosarjatiedoston luomisessa käytetään hyväksi asentamisen yhteydessä taulukoituja tietoja kaikista ladatuista ja asennetuista paketeista. Tämän taulukon perusteella kullekin sen alkiolle luodaan komentosarjaan oma kohtansa, joka siirtää paketin asennustiedostot paikoilleen väliaikaistiedostokansiosta. Jos käyttöönotettavana on ohjelmiston vanha versio, johon ollaan palaamassa kohdassa 5.9 kuvaillulla tavalla, luodaan sitä varten erilainen komentosarjatiedosto.

Listauksessa 2 olevaa käyttöönoton suorittavaa komentosarjatiedostoa ajettaessa ohjelmisto on vielä aluksi käynnissä. Siksi komentosarjassa odotetaan ensin, että ohjelmiston prosessi katoaa kohdejärjestelmän ajossa olevien prosessien listalta. Sitten se siirtää kaikki nykyisen version tiedostot erikseen kuvassa 16 luomaansa alikansioon talteen aliluvussa 5.9 kuvailtua vanhaan versioon palaamisen mahdollistava toiminnallisuutta varten ja siirtää uuden version tiedostot sinne missä nykyisen version tiedostot olivat. Lopuksi komentosarja käynnistää juuri käyttöönotetun version ohjelmistosta ja sulkee komentoriviikkunan.

Tiedostojen siirtämisen jälkeen käyttöönottokomentosarjatiedosto siivoaa mahdolliset ylimääräiset vanhat versiot `rollback`-kansiosta ja poistaa muut turhat tiedostot kansiosta `AutoUpdaterCache`. Juuri ajossa ollutta ohjelmiston versiota vanhempia versioita ei enää säilytetä. Tästä kerrotaan lisää kohdassa 5.9. Palvelimelta ladatut NuGet-paketit säilytetään, jotta niitä ei tarvitse seuraavan päivityksen yhteydessä ladata, jos ne eivät ole päivittyneet.

Käyttäjälle käyttöönotto näkyy ohjelman sulkemisen yhteydessä vilahtavana komentokehoiteikkunana, kun komentosarjatiedosto on ajossa, ja uuden version automaattisena käynnistymisenä. Uuden version käytössä oleminen näkyy käyttäjälle juuri käynnistetyn ohjelmiston tervetulonäkymässä olevasta versionumerosta.



Kuva 16. *Asennustiedostot ja vanhan version varmuuskopio itsepäivitystoiminnallisuuden väliaikaistiedostokansiossa.*

5.7 NuGetin käyttö

Kohdassa 4.1 kuvailtua erillistä NuGet-ohjelmaa käytetään päivitysten saatavuuden tarkistamiseen sekä päivitysten lataamiseen ja asentamiseen. Tässä kohdassa käsitellään taustalla oleva tekninen toteutus. Saatavuustarkistuksesta kerrotaan yleisellä tasolla kohdassa 5.4. Lataamisesta ja asentamisesta kerrotaan kohdassa 5.6.

NuGet ei tarjoa C++:lle suoraa ohjelmallista rajapintaa, jonka kautta sitä voisi kutsua ja jonka kautta siltä saisi toimintojen tulokset ulos. Tämän takia NuGetia varten luotiin päivityskomponenttiin omat käsittelyfunktiot, jotka ajavat NuGetia omana prosessinaan, käsittelevät sen tulosteet ja käsittelevät mahdolliset virhetilanteet. Virheistä kerrotaan tarkemmin kohdassa 5.8. Erillisen NuGet-prosessin käsittelyssä hyödynnetään Qt:n tarjoamaa QProcess-luokkaa.

NuGetista toimitetaan ohjelmiston mukana ainoastaan sen komentorivikäytön mahdollistava ohjelmatiedosto nuget.exe. NuGetin ajoa varten sille luodaan konfiguraatiotiedosto käytettäväksi valitun päivityspalvelimen perusteella. Tämän tiedoston avulla NuGetille saadaan välitettyä päivityspakettivaraston sijainnin kertova URL-osoite ja käyttäjätunnukset, joilla se voi tunnistautua palvelimelle. Konfiguraatiotiedosto sisältää salasanan salatussa muodossa. Tunnistautumistiedot lähetetään purettuna palvelimelle salatun yhteyden yli.

Konfiguraatiotiedosto on XML-muotoinen [22]. Ensin luodaan Qt:n tiedostonkäsittely-

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3    <packageSources>
4      <add key="Internal␣test␣repository" value="https://server.tld/repository/" />
5    </packageSources>
6    <packageSourceCredentials>
7      <Internal_x0020_test_x0020_repository>
8        <add key="Username" value="username" />
9        <add key="Password" value="encryptedpassword" />
10     </Internal_x0020_test_x0020_repository>
11   </packageSourceCredentials>
12 </configuration>
```

Listaus 3. *NuGetin käyttämä konfiguraatiotiedosto asetettuna käyttämään sisäistä päivityspakettivarastoa.*

toimintojen avulla mahdollisimman yksinkertainen konfiguraatiotiedoston rungon sisältävä pohjatiedosto. Sitten ajetaan NuGet-komento `config`, jolle annetuilla parametreilla osoitetaan muokattavana olevan juuri äsken luotu konfiguraatiotiedosto ja siihen syötettävät tiedot. Tällä tavalla konfiguraatiotiedosto on varmasti syntaksiltaan oikein. Päivityspalvelimelle tarvittava salasana on pakattuna ohjelmiston mukana toimitettuun salattuun tiedostoon. Konfiguraatiota luodessa salasana puretaan tiedostosta ja annetaan parametrina NuGetille, joka jälleen salaa sen konfiguraatiotiedostoon.

Tietoturvallisuuden kannalta päivityspalvelimen salasana on ohjelmiston käyttökontekstissa riittävän hyvin suojassa, mutta se voidaan saada ajonaikaisesti selville, jos osataan tutkia ohjelmaa ja prosesseja oikeassa välissä. Salasana on selkokielisenä ohjelman muistissa sillä välillä, kun se on purettu salatusta tiedostosta ja sitä ollaan antamassa NuGetin konfiguraatiotiedostoon. Salasana annetaan myös selkokielisenä suoritettavalle NuGet-prosessille, jolloin se voidaan havaita käyttöjärjestelmätasolla tutkimalla ajettavien prosessien argumentteja.

5.8 NuGetin virhetilanteiden käsittely

Merkittävän osan päivitystoiminnallisuuden raskaista töistä tekee omassa aliprosessissaan Qt:n QProcess-luokan avulla ajettu NuGet. Kun sitä ajetaan, sen virheistä saadaan tietoa ainoastaan sen tulosteiden ja prosessin paluukoodin kautta. Lisäksi on mahdollista, että prosessi tuhoutuu muusta syystä, jolloin ohjelmalta ei välttämättä saada tulosteita ja paluukoodia. Näihin kaikkiin virhetyyppeihin on varauduttu.

Iso osa virhetapauksista on havaittavissa vain NuGetin tulosteista. NuGet tulostaa toiminnoistaan runsaasti viestejä standarditulostevirtaan ja virheiden sattuessa virheviestit standardivirhevirtaan. Virheen sattuessa standardivirhevirtaan tulee kuitenkin hyvin rajallisesti tulosteita, ja standarditulostevirtaan tulee virhettä tarkemmin kuvaavia tietoja.

Prosessin tulosteita ei lueta kuin vasta sen loputtua, eivätkä tulosteet ole aikaleimattuja, joten kahdesta erillisestä virrasta ei saataisi koottua jälkikäteen yhtenäistä kuvausta siitä, mitä tapahtui ja missä järjestyksessä. Tästä syystä NuGet-prosessi ajetaan niin, että standardituloste- ja standardivirhevirta on yhdistetty yhteen ulostuloon.

Virheiden havaitsemisen toteutus tulosteista ei ole yksinkertainen eikä mahdollisten NuGetissa tapahtuvien muutosten takia varma ratkaisu. Tämän vuoksi virhetarkastuksiin tehtiin helposti muutoksiin mukautuva toteutus. Varmuus kaikkien NuGetin virhetilanteiden kattamiseen on suuri, koska NuGet on avoimen lähdekoodin ohjelma. Sen lokalisaatiotiedostoista kirjattiin ylös kaikki virhekoodit ja niiden sanamuodot. Virheet kategorisoitiin muutamaan yleistapaukseen ja yksittäisiä virheviestejä varten luotiin lisätietoja säännöllisillä lausekkeilla keräävät lisätietueet. NuGetia ajetaan pakottaen se käyttämään tutkittua lokalisaatiota, jonka kaikki mahdolliset tulosteet ovat tiedossa. Täten voidaan olla varmoja siitä, että säännöllisillä lausekkeilla ei vahingossa tunnisteta vääriä virheilmoituksia, joihin ei olla osattu varautua.

Virheitä NuGetin tulosteista keräävästä tietueesta on havainnollistus listauksessa 4. Päivitystoiminnallisuudessa tällaiset tietueet luodaan erillisessä tehdasfunktiossa, joka valmistelee kutakin erilaista virhetyyppiä varten sopivan tietueen. Tällä tavalla kaikki virhetekstit on koottu tiiviisti yhden funktion sisälle. Tietueen kenttä `detailMatchers` sisältää taulukollisen Qt:n QRegularExpression-olioita, jotka toteuttavat säännöllisen lausekkeen toimintoja. Nämä QRegularExpression-oliot luodaan tehdasfunktiossa ja niille annetaan siellä nimettyjä kaappauksia (*capture*), joihin täsmäävän tekstin ne kaappaavat talteen NuGetin tulosteista. Virhetietueen kenttä `detailUiFormat` sisältää Qt:n QString-merkkijonoluokan mukaisen virheviestin, jossa on valmiiksi paikat säännöllisten lausekkeiden nimetyille kaappauksille. Tällä tavalla saadaan muotoiltua NuGetin teknisellä kielellä esitetystä tulosteesta käyttäjille näytettäväksi sopiva virheilmoitus ja käännettyä NuGetin käyttämät ilmaisut normaalista NuGetin käyttökontekstista ohjelmiston omaan kontekstiin.

Listauksen 4 `FindError`-funktiossa virhetietueita käytetään NuGetin tulosteiden lukemiseen yhtä kerrallaan. Ensin tietueet asetetaan prioriteettijärjestykseen sen mukaan, mikä virhe raportoitaisiin ensisijaisesti, jos virheitä olisi useita tai jos useamman tietueen virheentunnistukseen käytettävät säännölliset lausekkeet tuottaisivat osuman samalle tulosteelle. Sitten koko tuloste käydään läpi kullakin tietueella vuorostaan tarkistaen, tuottaako kyseisen tietueen virheentunnistukseen käytetty säännöllinen lauseke `firstErrorLine` osuman tulosteesta. Jos se tuottaa, kyseisen tietueen virhe raportoidaan myöhemmin. Riippumatta osumasta ensin käydään kuitenkin koko tuloste läpi, jolloin teknisiä yksityiskohtia etsivä `detailMatchers` kerää kaikki mahdolliset yksityiskohdat virheilmoitusta varten.

Nykytoteutuksessa virheitä ei odoteta tulevan useita kerrallaan, ja prioriteettilistalla on aina tuntematonta virhetilannetta varten yksi tietue, jonka tunnistuslauseke tuottaa osuman kaikkiin tulosteisiin. Eli jos mikään tunnettuun virhetilanteeseen reagoiva virhetietue ei

täsmää tulosteeseen, viimeisenä keinona näytetään yleinen virheilmoitus. Tämän virheen teknisiin lisätietoihin kerätään NuGetin koko tuloste.

5.9 Paluu aiempaan versioon

Itsepäivitystoiminnallisuus tarjoaa mahdollisuuden ottaa nopeasti käyttöön edellinen käytössä ollut ohjelmiston versio. Tätä kutsutaan *rollback*-ominaisuudeksi. Se tarjotaan, jotta käyttäjien tuottavuus ei kärsisi mahdollisen kelvottoman version käyttöönoton myötä ja jotta voitaisiin toipua päivitystilanteessa sattuvasta virhetilanteesta, joka pahimmillaan tuhoaa käyttöönotetun asennuksen.

Rollback vähentää vanhan toimivan version käyttöönottoon kuluvaa aikaa ja vaivaa merkittävästi. Toimivaa versiota ei tarvitse erikseen ladata verkosta eikä sitä tarvitse erikseen asentaa. Tällöin vanha versio on otettavissa käyttöön käytännössä ilman että käyttäjän tarvitsee erikseen odottaa lainkaan. Ilman paikallisesti säilöttyä vanhaa versiota lataamisessa voi kestää kauan, jos kohdejärjestelmässä on hidas verkkoyhteys. Manuaaliseen asennukseen vaadittaisiin käyttäjältä toimiksi vanhan version asennuspaketin etsiminen, lataaminen ja asentaminen. Tällä toteutuksella käyttöönoton voi tehdä suoraan ohjelmistosta yhdellä toimenpiteellä.

Jos käyttäjä on ottamassa vanhaa versiota käyttöön, sille on yleensä selkeä syy. Käyttäjän ei odoteta aloittavan erikseen toimenpidettä, jolla kumotaan päivitys uuteen, oletusarvoisesti parempaan versioon. Jos uusi versio ei ole parempi esimerkiksi vikojen ilmenemisen vuoksi, siitä on hyvä tietää mahdollisimman nopeasti. Silloin ongelmat voidaan korjata nopeammin. Nopeasti toimien on mahdollista myös rajoittaa viallisen päivityksen aiheuttamia haittoja ottamalla päivitys pois saatavilta.

Rollback on toteutettu säilömällä edellisen käytössä olleen version tiedostot ohjelmiston asennuskansiossa olevaan itsepäivitystoiminnallisuuden väliaikaistiedostokansioon ja ottamalla ne käyttöön samoilla mekanismeilla kuten kohdassa 5.6 kuvailtu yleinen päivitysten käyttöönotto. Rollback-tiedostojen sijainti on kuvan 16 mukainen. `Rollback`-kansiossa voi olla tallessa useampi versio samanaikaisesti, mutta turhan levytilan käytön vähentämiseksi ohjelmistosta ei säilötä kahta useampaa versiota kerralla. Ajatuksena on, että jos käyttäjä on asentanut yhden päivityksen ja myöhemmin päivittänyt ohjelmiston vielä toisen kerran, hänellä ei ole todennäköisesti enää syytä palata alkuperäiseen versioon. Taustalla on oletus, että käyttäjä ehtii havaita versiopäivitysten julkaisun välillä, onko päivitys kelvollinen vai ei. Tällöin edellisen asennetun päivityksen voidaan tulkita olevan kelvollinen vanha versio, johon palata.

Rollbackin yhteyteen on osin toteutettu palautteen kerääminen käyttäjiltä. Palautteen lähettämiseen käytetään QNetworkAccessManager-luokkaa, jolla otetaan palautetta keräävään palveluun yhteyttä HTTPS:n yli. Palaute on tarkoitus kerätä Jira-tehtävienhallintaoh-

```
struct NugetError
{
   enum class ErrorType
   {
      UNABLE_TO_CONNECT,
      FEED_NOT_FOUND,
      UNAUTHORIZED,
      OTHER_UNEXPECTED_CODE,
      PACKAGE_NOT_FOUND,
      OTHER
   };
   ErrorType errorType;
   // What to tell the users about what went wrong.
   QString briefDescription;
   // Match the first line indicating this error occurred.
   QRegularExpression firstErrorLine;
   // What the user should do to resolve the problem.
   QString instructions;
   // Capture relevant info. Only named captures are used.
   QVector<QRegularExpression> detailMatchers;
   // Map from capture names used in detailMatchers to descriptive UI text.
   QHash<QString, QString> detailUiFormat;
   // Store formatted detailUiFormat strings here.
   QVector<QString> storedDetails;
};
NugetError AutoUpdater::FindError(const QStringList& nugetOutputLog) const
{
QVector<NugetError> errors;
CreateErrorMatchers(errors); // Create every error type in priority order.
for (NugetError& err : errors)
{
   bool hasErrorMatched = false;
   for (const QString& line : nugetOutputLog)
   {
      if (line.contains(err.firstErrorLine))
      {
         hasErrorMatched = true;
      }
      // Store details for the error even if it hasn't matched.
      for (const QRegularExpression& detail : err.detailMatchers)
      {
         QRegularExpressionMatch match;
         if (line.contains(detail, &match))
         {
            for (const QString& name : detail.namedCaptureGroups())
            {
               ...
               // Format the UI string with the named capture.
               QString f = err.detailUiFormat[name].arg(match.captured(name));
               err.storedDetails.push_back(f);
            }
         }
      }
   }

   if (hasErrorMatched)
   {
      return err;
   }
}
return errors.back(); // If none matched, return error of type OTHER.
}
```

Listaus 4. *Tietojen kerääminen virheilmoitukseen NuGetin tulosteiden pohjalta.*

jelmistoon tämän REST-rajapintaa käyttäen. Jira-instanssia palautetta varten ei ole vielä otettu käyttöön, joten palautteen lähettämistoiminto on toistaiseksi poissa käytöstä.

6. ARVIOINTI

Työssä toteutettiin teorialuvun periaatteiden mukaisesti toimiva itsepäivitystoiminnallisuus ohjelmistoon. Tässä luvussa arvioidaan sille asetettujen vaatimusten toteutumista ja verrataan sitä muiden ohjelmistojen vastaaviin päivitystoiminnallisuuksiin. Lopuksi pohditaan arvioinnin ja vertailun pohjalta mahdollista jatkokehitystä.

6.1 Tulokset

Kohdassa 2.4 työlle asetettiin tavoitteita. Myöhemmin tavoitteiden täyttämiseksi toteutettiin toiminnallisuuksia. Käydään tavoitteet ja niiden toteutumiset läpi.

Nopeamman toimituksen tavoitteen täyttämiseksi toiminnallisuuteen toteutettiin päivitysten saatavuudesta ilmoittaminen käyttäjille ja mahdollisuus ottaa päivitykset käyttöön ohjelmiston käyttöliittymän kautta. Työn tuloksena toteutettu itsepäivitystoiminnallisuus on ohjelmistossa käytössä ja saavuttanut kypsyyden ominaisuutena. Toiminnallisuudella on toimitettu asiakkaan käyttäjille kokeellinen päivitysjulkaisu.

Lisäksi toteutettiin vaiheistettu päivitysten julkaiseminen, jolloin osa käyttäjistä saa päivityksen nopeammin käyttöönsä. Vaiheistettu julkaisu toteutettiin myös täyttämään laadun takaamisen tavoite. Palautteen takaisinkytkennän nopeuttaminen keräämällä käyttäjiltä palautetta ei ole toteutunut. Palautetoiminnallisuus on keskeneräinen ja edelleen työn alla.

Jatkokehitys mahdollistettiin valitsemalla löyhästi itsepäivitystoteutukseen kytkeytyvä työkalu ja toteuttamalla itsepäivityksen käyttöliittymä ja tiedostonkäsittelytoiminnot itse. Ylläpidettävyyteen kiinnitettiin huomiota tekemällä uudelleenkäytettäviä ja dokumentoituja toimintoja.

6.2 Keskustelu

Vaikka itsepäivitystoiminnallisuus on käytössä, siitä ei ole vielä saatu kerättyä järjestelmällisesti käyttäjäpalautetta eikä sen toimintaa olla vielä tarkasteltu useammassa käyttäjille suunnatun päivityksen julkaisussa. Kun toiminnallisuudesta saadaan käyttäjäpalautetta, sen ratkaisuja voidaan arvioida luotettavammin.

Toteutettu päivitystoiminnallisuus ei lyhennä käyttöönottoon kuluvaa aikaa niin paljoa kuin on mahdollista, koska se ei automaattisesti suorita aikaa vieviä päivityksen vaiheita taustalla. Sen sijaan se ryhtyy päivittämiseen vasta käyttäjän kehoitteesta, jolloin asennukseen tulee odotusaikaa lataamisesta, purkamisesta, tiedostojen siirtelystä ja ohjelmis-

ton uudelleenkäynnistyksestä. Kaikki odotusaika lisää päivityskynnystä. Automatisoimalla aikaavievät vaiheet saataisiin kynnys pienemmäksi, joskin siinäkin on ongelmansa taustalla tapahtuvien käyttäjälle näkymättömien toimenpiteiden kuluttaessa resursseja.

Itse toteutettuna päivitysmekanismi on muun ohjelmiston näkökulmasta monimutkaisempi ja sen toteutukseen on kulunut huomattavasti enemmän työtä kuin valmiin päivitystoiminnallisuuden käyttöönottoon olisi mennyt. Sitä on kuitenkin mahdollista jatkokehittää ja mukauttaa muuttuviin tarpeisiin, mihin muut eivät pysty. Jatkokehityksen kannalta ongelmallisia tekniikoita toteutuksessa ovat BAT-komentojonotiedostojen käyttö,

NuGetin heikko yhteys ohjelmistoon aiheuttaa ongelmia virheiden havaitsemisessa ja päivityksen etenemisen seuraamisessa. Koska NuGetin ajonaikaista toimintaa ei voida tarkkailla ulkopuolelta, ei tiedetä, tekeekö se haluttuja asioita vai odottaako se jotakin tapahtumaa aikakatkaisuun asti. Pelkästä ohjelman tulosteiden mukana tulevasta tiedosta ei voida kaikissa tilanteissa päätellä, onko ohjelman suoritus etenemässä. Ongelmana on myös NuGetin pitkät aikakatkaisut verkkoyhteysongelmien kanssa. Oman lyhyemmän aikakatkaisun tekeminen prosessin ajoon voi aiheuttaa ongelmia, koska prosessi saattaa olla tekemässä haluttuja asioita mutta hyvin hitaasti. Tällöin ei jälkikäteen pystytä välttämättä päättelemään, olisiko operaatio onnistunut.

6.3 Jatkokehitys

Työssä toteutetun itsepäivitystoiminnallisuuden kehitys jatkuu. Tässä kohdassa esitetään erilaisia parannusehdotuksia ja jatkokehitysvaihtoehtoja, joilla toiminnallisuus voisi täyttää vaatimukset paremmin.

Työssä aiemmin kuvaillut tavoitteet jatkuvasta toimituksesta ja komponenttikohtaisesta päivityksestä on hyvä arvioida ottamalla niitä ohjelmistotuotantoprosessissa vaiheittain käyttöön. Kohdassa 3.1 mainittu Holmströmin ja Boschin [24] työssään kuvaama malli mahdollistuisi loppuunsaatetun jatkuvan toimituksen avulla. Sitä tai jotain sen kaltaista mallia voisi soveltaa uusien toiminnallisuuksien toteutuksessa.

Palautteen keräämisen lisäksi käyttökokemusta ja muutosten vaikutusta ohjelmiston käyttöön saataisiin arvioitua keräämällä metriikkaa ohjelmiston käytöstä. Tilastoja päivitysten käyttöönotosta on helppo luoda ja käyttäjien reagoimista päivityksiin olisi yksinkertaista raportoida. Ensin on selvitettävä, millaisten ehtojen nojalla dataa voidaan kerätä, lähettää ja säilöä.

Käyttäjäkokemusta parantaisi päivitystoimenpiteiden edistymisen seuranta lataamista odotellessa. Käyttöliittymän lisätty toimenpiteisiin jäljellä olevan ajan arviointi ja edistymisen visualisointi antaisivat käyttäjille nopeasti käsityksen ohjelmiston päivittämiseen kuluvasta ajasta. Tämän toteuttaminen tarkoittaa nykytoteutuksessa NuGetin käyttötavan arviointia tai sille vaihtoehtojen uudelleenkartoittamista. Monet työkalut tukevat NuGet-paketteja.

7. YHTEENVETO

Työssä esiteltiin itsepäivitystoinnallisuuden suunnittelu ja toteutus nopeuttamaan ohjelmistopäivitysten käyttöönottoa. Työssä käytiin läpi suunnitteluun vaikuttavat taustat työn kohteena olevasta ohjelmistosta ja sen kehitysprosessista. Niiden pohjalta asetettiin tavoitteet toteutukselle. Alan kirjallisuuden teoriaa käytiin läpi ja sen pohjalta suunniteltiin itsepäivitystoiminnallisuuden toteutusta ja perusteltiin olemassaolevia suunnitelmia.

Osana suunnittelua suoritettiin käytettävien työkalujen arviointi, jonka tulokset dokumentoitiin ja esiteltiin. Työkalujen tarkemmassa arvioinnissa toteutettiin prototyyppejä, joissa käytettyjä ratkaisuja sovellettiin itsepäivitystoiminnallisuuteen. Toiminnallisuuden toteutus esiteltiin ja sen tärkeimpiä yksityiskohtia tuotiin esiin. Lopuksi toteutusta arvioitiin ja sen ratkaisuista keskusteltiin. Keskustelun pohjalta esitettiin ehdotuksia jatkokehitykselle.

LÄHTEET

[1] F. Barz, QtAutoUpdater. Saatavissa (viitattu 20.11.2018): https://github.com/Skycoder42/QtAutoUpdater/

[2] Chocolatey Software, Inc., Chocolatey – The package manager for Windows. Saatavissa (viitattu 20.11.2018): https://chocolatey.org/

[3] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano, DevOps, IEEE Software, vsk. 33, nro 3, 2016, s. 94–100.

[4] M. Fleischmann, M. Amirpur, T. Grupp, A. Benlian, T. Hess, The role of software updates in information systems continuance—An experimental study from a user perspective, Decision Support Systems, vsk. 83, 2016, s. 83–96.

[5] R. Gandrud, All You Need Is One – A ClickOnce Love Story. Saatavissa (viitattu 20.11.2018): https://blog.netspi.com/all-you-need-is-one-a-clickonce-love-story/

[6] K. Górski, Installation and Update Systems for Windows (Part 2): ClickOnce - As simple as It Gets? Saatavissa (viitattu 20.11.2018): https://developers.livechatinc.com/blog/clickonce/

[7] Google Update for Windows. Saatavissa (viitattu 20.11.2018): https://github.com/google/omaha

[8] GrammaTech, Inc., CodeSonar - Static Analysis SAST Software for Secure SDLC. Saatavissa (viitattu 20.11.2018): https://www.grammatech.com/products/codesonar

[9] M. Hüttermann, DevOps for Developers, Springer Verlag, Berkeley, Kalifornia, Yhdysvallat, 2012, 176 s.

[10] S. Jansen, S. Brinkkemper, Ten Misconceptions about Product Software Release Management explained using Update Cost/Value Functions, teoksessa: 2006 International Workshop on Software Product Management (IWSPM'06 - RE'06 Workshop), Minneapolis, Minnesota, Yhdysvallat, 2006, IEEE Press, s. 44–50.

[11] Jenkins. Saatavissa (viitattu 20.11.2018): https://jenkins.io/

[12] JetBrains s.r.o., TeamCity: the Hassle-Free CI and CD Server by JetBrains. Saatavissa (viitattu 20.11.2018): https://www.jetbrains.com/teamcity/

[13] M. Kajander, Ohjelmistokehityksen arvontoimituksen nopeuttaminen automaattisilla päivityksillä, kandidaatintyö, Tampereen teknillinen yliopisto, Tietotekniikan laboratorio, Tampere, 2018, 17 s.

[14] F. Khomh, T. Dhaliwal, Y. Zou, B. Adams, Do faster releases improve software quality?: an empirical case study of Mozilla Firefox, teoksessa: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, Zürich, Sveitsi, 2012, IEEE Press, s. 179–188.

[15] T. Lehtonen, T. Kilamo, S. Suonsyrjä, T. Mikkonen, Continuous, Lean, and Wasteless: Minimizing Lead Time from Development Done to Production Use, teoksessa: 42th Euromicro Conference on Software Engineering and Advanced Applications, Limassol, Kypros, 2016, IEEE, s. 73–77.

[16] M. Ltd., Update-Installer. Saatavissa (viitattu 20.11.2018): https://github.com/mendeley/Update-Installer/

[17] A. Mathur, M. Chetty, Impact of user characteristics on attitudes towards automatic mobile application updates, teoksessa: Thirteenth Symposium on Usable Privacy and Security (SOUPS), Santa Clara, California, Yhdysvallat, 2017.

[18] A. Mathur, N. Malkin, M. Harbach, E. Peer, S. Egelman, Quantifying Users' Beliefs about Software Updates, tou. 2018.

[19] Microsoft Corporation, ClickOnce security and deployment. Saatavissa (viitattu 20.11.2018): https://docs.microsoft.com/en-us/visualstudio/deployment/clickonce-security-and-deployment

[20] Microsoft Corporation, NuGet Documentation | Microsoft Docs. Saatavissa (viitattu 20.11.2018): https://docs.microsoft.com/en-us/nuget/

[21] Microsoft Corporation, NuGet Package Dependency Resolution | Microsoft Docs. Saatavissa (viitattu 20.11.2018): https://docs.microsoft.com/en-us/nuget/consume-packages/dependency-resolution

[22] Microsoft Corporation, nuget.config File Reference | Microsoft Docs. Saatavissa (viitattu 20.11.2018): https://docs.microsoft.com/en-us/nuget/reference/nuget-config-file

[23] T. H. Netland, D. J. Powell, The Routledge Companion to Lean Management, Routledge Ltd, Lontoo, jou. 2016, 478 s.

[24] H. H. Olsson, J. Bosch, From Opinions to Data-Driven Software R&D: A Multi-case Study on How to Close the 'Open Loop' Problem, teoksessa: 40th EUROMICRO Conference on Software Engineering and Advanced Applications, Verona, Italia, elo. 2014, IEEE, s. 9–16.

[25] Open Source Initative, The 2-Clause BSD License. Saatavissa (viitattu 20.11.2018): https://opensource.org/licenses/BSD-2-Clause

[26] Open Source Initative, The 3-Clause BSD License. Saatavissa (viitattu 20.11.2018): https://opensource.org/licenses/BSD-3-Clause

[27] Open Source Initative, The MIT License. Saatavissa (viitattu 20.11.2018): https://opensource.org/licenses/MIT

[28] A. Porter, C. Yilmaz, A. M. Memon, A. S. Krishna, D. C. Schmidt, A. Gokhale, Techniques and processes for improving the quality and performance of open-source software, Software Process: Improvement and Practice, vsk. 11, 2006, s. 163–176.

[29] T. Preston-Werner, Semantic Versioning 2.0.0. Saatavissa (viitattu 20.11.2018): https://semver.org/

[30] V. Slavik, WinSparkle. Saatavissa (viitattu 20.11.2018): https://winsparkle.org/

[31] Sonatype Inc., Nexus Repository Manager - Software Component Management. Saatavissa (viitattu 20.11.2018): https://www.sonatype.com/nexus-repository-sonatype

[32] A. Spataru, QSimpleUpdater. Saatavissa (viitattu 20.11.2018): https://github.com/alex-spataru/qsimpleupdater/

[33] Squirrel.Windows: An installation and update framework for Windows desktop apps. Saatavissa (viitattu 20.11.2018): https://github.com/Squirrel/Squirrel.Windows/

[34] S. Streeting, Friends don't let friends use ClickOnce. Saatavissa (viitattu 20.11.2018): https://www.stevestreeting.com/2013/05/12/friends-dont-let-friends-use-clickonce/

[35] P. Sturgeon, The "Dont Be a Dick" Public License. Saatavissa (viitattu 20.11.2018): https://dbad-license.org/

[36] S. Tachenov, QuaZIP. Saatavissa (viitattu 20.11.2018): https://github.com/stachenov/quazip/

[37] The Apache Software Foundation, Apache License Version 2.0. Saatavissa (viitattu 20.11.2018): https://www.apache.org/licenses/LICENSE-2.0

[38] The Qt Company Ltd., All Classes | Qt 5.11. Saatavissa (viitattu 20.11.2018): http://doc.qt.io/qt-5/classes.html

[39] The Qt Company Ltd., Libraries & APIs, Tools and IDE | Qt. Saatavissa (viitattu 20.11.2018): https://www.qt.io/qt-features-libraries-apis-tools-and-ide/

[40] The Qt Company Ltd., Qt | Cross-platform software development for embedded & desktop. Saatavissa (viitattu 20.11.2018): https://www.qt.io/

[41] The Qt Company Ltd., The Qt Company. Saatavissa (viitattu 20.11.2018): https://www.qt.io/company/

[42] The Qt Company Ltd., Signals & Slots | Qt Core 5.11. Saatavissa (viitattu 20.11.2018): https://doc.qt.io/qt-5/signalsandslots.html

[43] J. Vainio, Kontekstisensitiivisen ohjejärjestelmän suunnittelu ja toteutus Qt-sovellukseen, diplomityö, Tampereen teknillinen yliopisto, Tietotekniikan laitos, Tampere, 2015, 47 s. Saatavissa: http://urn.fi/URN:NBN:fi:tty-201504221213

[44] L. Valiukas, Autoupdate-haara projektista Fervor. Saatavissa (viitattu 20.11.2018): https://github.com/pypt/fervor/tree/autoupdate

[45] L. Valiukas, Simple multiplatform (Qt-based) application update tool inspired by Sparkle. Saatavissa (viitattu 20.11.2018): https://github.com/pypt/fervor/

[46] K. Vaniea, Y. Rashidi, Tales of Software Updates, teoksessa: Proceedings of the 2016 CHI Conference on human factors in computing systems, San Jose, California, Yhdysvallat, tou. 2016, ACM, CHI '16, s. 3215–3226.

[47] C. Williams, Eight Evil Things Microsoft Never Showed You in the ClickOnce Demos (and What You Can Do About Some of Them). Saatavissa (viitattu 20.11.2018): https://semver.org/