VAASAN YLIOPISTO

TEKNIIKAN JA INNOVAATIOJOHTAMISEN YKSIKKÖ

TIETOJENKÄSITTELYTIETEET

Ville Tienvieri

TUOTEINFORMAATION HALLINTAJÄRJESTESLMÄN KEHITTÄMINEN

Diplomityö, joka on jätetty tarkastettavaksi diplomi-insinöörin tutkintoa varten
Vaasassa 16.9.2019.

Työn valvoja Jouni Lampinen

Työn ohjaaja Tuomo Heikkilä

SISÄLLYSLUETTELO **sivu**

LYHENNELUETTELO

CFD	Cumulative Flow Diagram.
JBoss	Sovelluspalvelin.
Kanban	Tapa visualisoida työmäärä ja mahdollistaa prosessin jatkuvan kehittämisen
Scrum	Tapa kehittää ohjelmistoja nopeasti ja sopeutuen tilanteisiin niiden tullessa eteen.
SGML	Standard Generalized Markup Language on ISO 8879:1986 standardi, jolla määritellään geneerinen merkkauskieli.
SOAP	Single Object Access Protocol, ohjelmistojen välinen viestipohjainen tietoliikenneprotokolla.
SPD	Standard Product Database, toteutetun ohjelmiston nimi.
Sprint	Tietyn mittainen ajanjakso, jonka aikana kehitetään ohjelmistoa.
Stub	Koodin osa, jolla korvataan hetkellisesti monimutkainen toteutus.
W3C	World Wide Web Consortium kehittää ja ylläpitää internettiin liittyviä tekniikoita ja standardeja.
XML	Extensible Markup Language, kuvauskieli, jolla voidaan esittää ja jäsentää isojakin tietomääriä.
XP	Extreme programming, yksi tapa kehittää ohjelmistoa ketterällä menetelmällä.
Qt	Alustariippumaton ohjelmistojen kehitysympäristö.

KUVALUETTELO

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen yksikkö
Tekijä: Ville Tienvieri
Diplomityön nimi: Tuoteinformaation hallintajärjestelmän kehittäminen
Valvoja: Prof. Jouni Lampinen
Ohjaaja: DI, KTM Tuomo Heikkilä
Tutkinto: Diplomi-insinööri (DI)
Koulutusohjelma: Tietotekniikan koulutusohjelma
Suunta: Ohjelmistotekniikka
Opintojen aloitusvuosi: 2009
Diplomityön valmistumisvuosi: 2019 **Sivumäärä:** 64

TIIVISTELMÄ:

Tämän diplomityön aiheena on tuoteinformaation hallintajärjestelmän kehittäminen, joka tehdään toimeksiannosta kohdeyritykselle. Työn tavoitteena on kuvata ohjelmiston kehitysprosessia sekä toteutettava ohjelmisto. Tuoteinformaation hallintajärjestelmään voidaan tallentaa tuotetietoa ja tuoda se myös muiden ohjelmien käyttöön. Etuina nähtiin ohjelmiston integroimisen olemassa olevaan konfigurointityökaluun ja samalla saatiin myös käyttöliittymä modernisoitua. Kaikki tiedot ovat tällöin hallittavissa samasta paikasta ja käyttäjien työskentely tehostuu. Tuoteinformaation hallintajärjestelmän kehittäminen lähti liikkeelle asiakkaan vaatimusten keräämisestä. Asiakkaan tarpeiden ymmärtäminen on kehityksen lopputuloksen kannalta erittäin olennaista. Asiakkaan kanssa yhdessä kirjoitettujen vaatimusten pohjalta voitiin suunnitella millainen järjestelmästä tuli.

Ohjelmisto päätettiin toteuttaa ketterää ohjelmistokehitystä käyttäen. Ohjelmiston kehityksen kestäessä hyvinkin pitkän aikaa oli mahdollista käyttää kehitysprosessissa ensin Scrumia ja lopulta Kanbania. Kanbaniin siirryttiin, koska Scrum tuntui liian byrokraattiselta ja työläältä. Näiden kahden ketterän kehityksen menetelmän välissä käytettiin myös hyvin kevennettyä Scrum-prosessia, jossa oli mukana oikeastaan vain työtehtävien arvioiminen. Yhteistä näille metodeille ovat pyrkimys tehdä pieni osa kerrallaan valmiiksi. Koska tehtävät osakokonaisuudet ovat pieniä, on tällöin mahdollista reagoida asiakkaalta tuleviin muutospyyntöihin nopeasti.

Lopputulokseksi aikaansaatiin järjestelmä, jossa säilytetään ja hallinnoidaan tuoteinformaatiota. Järjestelmä koostuu tietokannasta, sovelluspalvelimesta sekä asiakasohjelmasta. Asiakasohjelmaan syötetään tuotetietoja ennalta määrätyssä muodossa. Johtuen tuotteiden erilaisuudesta, myös tietojen esitystapa ja muoto voivat olla hyvin erilaisia. Ohjelmistolla voidaan tietojen syöttämisen lisäksi myös ottaa käyttöön aikaisemmin sinne tallennettuja tietoja. Ohjelmisto on otettu tuotantokäyttöön ja sitä käytetään kohdeyrityksessä aktiivisesti.

AVAINSANAT: Ohjelmisto, järjestelmän kehittäminen, ketterä, tuoteinformaatio

UNIVERSITY OF VAASA
School of technology and innovation
Author: Ville Tienvieri
Topic of the Thesis: Development of a product information management system
Supervisor: Prof. Jouni Lampinen
Instructor: M.Sc. (Tech), M.Sc (Econ) Tuomo Heikkilä
Degree: Master of Science in Technology
Degree Programme: Degree Programme in Information Technology
Major of Subject: Software Engineering
Year of Entering the University: 2009
Year of Completing the Thesis: 2019 **Pages:** 64

ABSTRACT:

The subject of this thesis is to develop a product information management system. This software is developed as an assignment for the target company. The goal of the thesis is to describe the software development process and what kind of software was actually developed. Product information can be saved to the product information management system and also the end data can be exported to other software. It was seen beneficial to integrate the software to the existing configuration tool. At the same time user interface could be modernized. All the data was located in one big software. This allows users to work in more efficient way. The software development started by collecting requirements from the customer. Understanding the customer's needs is were crucial that the end result is something that customer thinks it should be. With these requirements written in co-operation with customer it was possible to design how the software was to be.

It was decided to implement the software by applying agile software methodology. Because the software development process took lots of time it was possible to use multiple agile methods. First Scrum was used and later Kanban. The move from Scrum to Kanban was done because Scrum started to feel too bureaucratic and also too laborious. Between these two agile methods very light Scrumbut was used. It basically used only estimation process from Scrum. Common things for these two methods are the goal to make a small fraction of the program ready at a time. Then the developed software is so small it is possible to react quite fast to change requests received from the customer.

As the end result system was developed that allows to store and manage product information. The system consists of a database, an application server and the end user software. Product information is filled in the software in a predefined format. Products that are entered can be really different and how the information is shown can vary a lot. The software can also modify and utilize the existing data. The system has been taken into production use and it is used actively.

KEYWORDS: Software, system development, agile, product information

1 JOHDANTO

1.1 Tutkimuksen taustaa

Kohdeyrityksellä on erillinen ohjelmisto, johon tallennetaan toisen järjestelmän tarvitsemia tietoja. Tätä ohjelmistoa kutsutaan jatkossa nimellä SPD (Standard Product Database). Se on ollut käytössä jo usean vuoden ja siinä on huomattu puutteita ja rajoitteita. SPD-ohjelmiston käyttöoikeudet eivät ole yhtenevät kohdeyrityksen muiden ohjelmien kanssa. Käyttöoikeuksien hakeminen ohjelmistoon on hankalaa, koska se on tehtävä eri prosessin mukaan kuin muiden ohjelmien. Ohjelmisto on kokoelmia eri näkymiä. Eri näkymille on eri käyttöoikeudet. Käyttöoikeudet jokaiseen näkymään pyydetään eri paikasta ja eri henkilöt ovat vastuussa eri näkymien oikeuksien antamisesta. Käyttöoikeuksien hallinnoiminen on myös täten hyvin hankalaa.

SPD-ohjelmiston ulkoasu on myös hieman vanhahtava nykymittapuulla. Ulkoasu ei kuitenkaan ole suurin syy, miksi SPD halutaan uudistaa. Uusien ominaisuuksien rakentaminen ohjelmistoon tulisi olemaan hyvin haasteellista monimutkaisen tietokantarakenteen takia. Lisäksi sen alkuperäiset kehittäjät ovat siirtyneet muiden töiden pariin tai lähteneet kokonaan yhtiöstä. Nämä syyt ovat varmasti olleet syitä siihen miksi ohjelmistosta päätettiin tehdä uusi versio. Lisäksi uuden version kehittämiseen on osaltaan varmasti ollut vaikuttamassa rinnalla rakennettu uusi konfigurointityökalu.

Konfigurointityökalu on Windowsin päällä ajettava ohjelma, jolla voi konfiguroida ison automaatiojärjestelmän toimintaa. Kyseinen konfigurointiotyökalu käyttää SPD-ohjelmiston tarjoamaa tietoa. Kyseisestä konfigurointiotyökalusta on olemassa myös kaksi vanhempaa ohjelmaa. Se käytännössä yhdistää molemmat vanhat työkalut yhdeksi työkaluksi. Toisella vanhalla työkalulla konfiguroitiin automaatiojärjestelmää ja toisella tarkkailtiin kyseistä järjestelmää, kun se on toiminnassa. Vanhempien työkalujen kehitys on lopetettu ja niihin tehdään korjauksia enää vain, kun löydetään kriittisiä ongelmia, jotka estävät tiettyjen toiminnallisuuksien käytön. Uuden konfigurointityökalun kehitys

jatkuu edelleen vahvana. Se mahdollistaa kaikki toiminnot, mitä vanhemmatkin ohjelmat ja lisäksi siihen on tehty paljon parannuksia ja uusia toimintoja.

Uuden konfigurointityökalun kehittämisen perusteena oli saada kahden työkalun toiminnallisuudet yhteen työkaluun. Lisäksi uuden ohjelman kirjoittaminen mahdollistaisi uudemman ja modernimman alustan käyttämisen kehityksessä. Uusi SPD-ohjelmisto haluttiin myös integroida huomattavasti paremmin konfigurointityökaluun kuin aikaisemmin oli tehty. Tietojen välittäminen SPD-ohjelmiston ja konfigurointityökalun välillä tapahtui aikaisemmin XML-tiedoston välityksellä, joka piti ladata erikseen vanhan SPD-ohjelmiston export-toimintoa käyttäen.

Tavoitteena on rakentaa uusi ohjelmisto, joka sisältää vanhan SPD-ohjelmiston toiminnallisuudet ja lisäksi uusia toimintoja uuden konfigurointityökalun tarpeisiin. Lähtökohtaisesti SPD-ohjelmiston sisältämä informaatio on käytössä vain konfigurointiotyökalussa. Näillä perusteilla nähtiin hyödyllisenä sulauttaa SPD-ohjelmisto osaksi konfigurointityökalua, siltä osin kuin se on mahdollista. Uuden SPD-ohjelmiston sulauttaminen osaksi konfigurointiotyökalua tarjoaa suuria synergiaetuja. Järjestelmän aikaisempi tuntemus mahdollistaa varsinkin alussa hyvin nopean kehityksen. Olemassa oleva koodipohja tarjoaa paljon olemassa olevia toimintoja, joita voidaan käyttää suoraan tai hyvin pienin muokkauksin.

Integrointi samaan työkaluun tuo myös ongelmia. Yksi suurimpia näistä on ohjelman julkaiseminen. SPD-työkalu on sidottu konfigurointityökalun julkaisusykliin, joka on huomattavasti hitaampi kuin SPD-työkalulla voisi olla. Samojen komponenttien käyttämisellä on myös varjopuolensa. Komponenttien muuttuessa myös niiden toiminnallisuus muuttuu paikoissa, joissa se ei ole välttämättä haluttua.

1.2 Tavoitteet ja tutkimusongelma

Diplomityön tavoitteena on toteuttaa ohjelmisto, jossa voidaan säilöä ja hallinnoida muiden ohjelmien tarvitsemaa tuoteinformaatiota. Tämä pitää sisällään graafisen

ohjelmiston loppukäyttäjälle, tietokantapalvelimen käyttöönoton sekä konfiguroinnin ja näiden kahden välisen kommunikoinnin mahdollistamisen. Lisäksi diplomityössä kuvataan kuinka ohjelmistoa on kehitetty, millaisia ratkaisuja on toteutettu ja mitä tekniikoita on käytetty. Diplomityö keskittyy enemmän ohjelmiston graafiseen loppukäyttäjälle tarkoitettuun osuuteen ja kehitysprosessiin. Sovelluspalvelin ja muut palvelimella sijaitsevat toiminnot jäävät vähemmälle käsittelylle.

1.3 Tutkimuksen rakenne

Diplomityön teoriaosiossa kuvataan ohjelmistossa käytettyjä keskeisiä tekniikoita, ketteriä kehitysmenetelmiä ja kerrotaan testauksesta. Lisäksi kerrotaan taustaa suuremmasta ohjelmasta, jonka osana diplomityössä toteutettu ohjelma on. Oman työn osuutta kuvaavissa kappaleissa kerrotaan ohjelmiston toiminnallisuuksia ja miten niitä on toteutettu. Olennaisena osana kehitystä on ollut myös ohjelmiston testaus. Sitä kuinka testaus on hoidettu, kuvataan yhden kappaleen verran. Ennen johtopäätöksiä annetaan parannusehdotuksia asioihin, joita on huomattu kehityksen aikana. Niitä annetaan teknisessä mielessä kuin myös kehitysprosessiin liittyen.

2 OHJELMISTON KEHITYS KETTERILLÄ MENETELMILLÄ

Ketterät kehitysmenetelmät ovat saaneet viime vuosina melkoista hypetystä osakseen. Ketterällä kehityksellä tarkoitetaan tapaa, jolla ohjelmistoa kehitetään. Agile alliancen (2015a) mukaan ketterä ohjelmistokehitys on kokoelma menetelmiä ja käytäntöjä, jotka perustuvat ketterän ohjelmistokehityksen julistukseen (Agile manifesto 2001a) ja ketteriin periaatteisiin (Agile manifesto 2001b). Ketterä kehitys siis ei itsessään vielä kerro tarkemmin käytännöistä ja tavoista, kuinka kehitys tapahtuu, sen avulla määritellään vain pääperiaatteet. Agile manifestoon (2001b) kuuluu kolme pääperiaatetta. Ensimmäinen on toimivan ohjelmiston arvostus ennen dokumentaatiota. Toisena painotetaan yksilön ja kanssakäymisen arvostusta ennen menetelmiä ja työkaluja. Kolmantena nähdään muutoksiin sopeutumisen parempana kuin alkuperäisissä suunnitelmissa pitäytymistä. Näitä periaatteita noudattamalla ohjelmien laatu saadaan paremmaksi, ainakin ketterän ohjelmistokehityksen julistuksen mukaan.

Tuovinen (2017) katsoo, että projektit, jotka käyttävät ketterää kehitystä ovat joustavia. Tämän takia toimiva kommunikaatio on hyvinkin tärkeää. Ilman sitä ei saavuteta joustavuutta. Tuovinen (2017) näkee myös, että asiakastyytyväisyys on tärkein laadun mittari ja projektin työntekijöiden tyytyväisyys on tärkein laadun tae. Lisäksi kehitystyö pitäisi olla systemaattista. Nämä tulkinnat on saatu käymällä läpi ketterän kehityksen 12 periaatetta. Ketterät menetelmät ja prosessit nojaavat hyvin vahvasti muutokseen. On nähty, että muutos on vääjäämätön tilanne, projektissa joudutaan tekemään hyvin joustavia ratkaisuja projektin hallinnassa.

Seuraavissa kappaleissa on kerrottu joistakin yleisesti käytössä olevista ketteristä kehityksen menetelmistä.

2.1 Scrum

Scrum on prosessimalli, jonka avulla on tarkoitus kehittää monimutkaisia tuotteita (Agile Alliance 2015b). Scrum ei ole tarkoitettu käytettäväksi jonkin asian valmistukseen.

(Schwaber Ken, Jeff Sutherland 2016). Schwaber ym. (2016) ja Agile Alliance (2015b) molemmat katsovat, että Scrum pohjautuu hyvin vahvasti empiiriseen eli kokemukselliseen prosessiin. Tieto pohjautuu kokemukseen ja päätettävät asiat pohjautuvat siihen mitä tiedetään. Scrumissa käytetään toistuvia ja kasautuva prosesseja, joilla hallitaan riskejä ja ennustettavuutta. Scrumissa on kolme keskeistä asiaa, jotka kaikki alleviivaavat Scrumin kokemuspohjaista luonnetta. Nämä ovat läpinäkyvyys, tarkastelu ja mukautuminen (Schwaber ym. 2016).

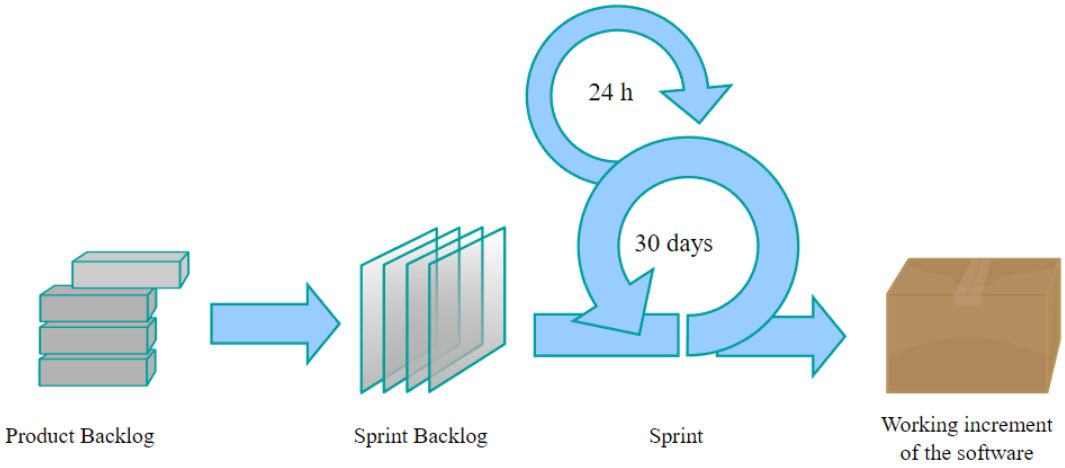
Läpinäkyvyys mahdollistaa sen, että koko tiimi on tietoinen toistensa tehtävistä. Lisäksi yhteisymmärrys siitä, mitä ollaan tekemässä, on myös osa läpinäkyvyyttä. (Agile Alliance 2015b) Tarkastelu on tärkeää, koska sillä tavalla nähdään, ollaanko menossa kohti tavoiteltua lopputulosta vai onko poikettu sivuraiteille. Tarkastelu hoidetaan Scrumissa päivittäisissä palavereissa sekä sprintin katselmointipalaverissa. Mukautuminen on hyvin oleellinen osa Scrumia. Kun tarkastusvaiheessa huomataan, että prosessi tai haluttu lopputulos ei ole tyydyttävä, on ryhdyttävä heti korjaaviin toimenpiteisiin, jotta projekti saadaan taas raiteilleen (Schwaber ym. 2016).

Scrum on hyvin prosessivetoinen tapa kehittää ohjelmistoja. Sillä määritellään kehys, jonka avulla päästään haluttuun lopputulokseen. Scrum ei ota kantaa mitä tekniikoita tulisi käyttää tai millaisia dokumentteja projektin aikana tulisi tehdä. Scrum ottaa kantaa siihen, kuinka projektiryhmän tulisi toimia ja tehdä yhteistyötä. (Tuovinen 2017)

Scrumissa on neljä eri vaihetta, jotka ovat suunnittelu, valmistelu, kehitys ja julkaisu. Suunnitteluvaiheessa luodaan perusta koko projektille. Siinä myös tehdään esitutkimusta ja varmistetaan, että projekti on varmasti toteuttamiskelpoinen. Lisäksi suunnitteluun kuuluu myös perinteinen vaatimusten kerääminen. Ne siirretään projektin backlogiin, jota kutsutaan myös työlistaksi. Valmisteluvaiheessa vaatimuksia tarkennetaan ja niitä priorisoidaan tärkeyden mukaan. Tässä vaiheessa yleensä myös tehdään jonkinlainen prototyyppi tai alustava suunnitelma, kuinka ohjelma tullaan kehittämään. (Tuovinen 2017)

Kolmas vaihe Scrumissa on varsinainen kehitystyö. Kehitysprosessi on kuvattu kuvassa 1. Tämä kehitysvaihe voidaan katsoa olevan koko Scrumin sydän (Schwaber ym. 2016).

Kehittäminen on pilkottu sprintteihin, joiden kesto projektiryhmästä riippuen voi olla kahdesta kuuteen viikkoon. Alun perin kesto oli aina 30 päivää.



Kuva 1 Scrum-prosessi (Wikimedia.org 2009).

Sprinttiin valitaan sopivat tehtävät kehityslistalta. Se on priorisoitu siten, että ideaalissa tilanteessa listan huipulta otetaan sopiva määrä tehtäviä. Tehtävälistaa ylläpitää projektin omistaja. Projektitiimi arvioi itse kuinka paljon töitä he pystyvät ottamaan työn alle yhteen sprinttiin. Projektin omistajalla on yleensä oma näkemys mitä hän haluaa saada tehdyksi. Jokainen kehityspäivä aloitetaan päivittäisellä tapaamisella, jossa käydään läpi mitä henkilöt ovat tehneet, mitä he tulevat seuraavaksi tekemään ja onko ollut ongelmia. Tätä prosessia jatketaan sprintin loppuun asti, jonka jälkeen tiimin aikaansaannokset käydään läpi sprintin katselmointipalaverissa. Tuotoksena voi olla esimerkiksi joku dokumentti tai valmis osa ohjelmistoa, jota esitellään. Tuotoksia peilataan vaatimuksiin, joita suunnittelupalaverissa tehtiin sprintin alussa. Tässä palaverissa on mukana henkilöt, joita varten ohjelmaa tehdään. Näitä ovat yleensä projektin omistaja sekä muita projektin sidosryhmiin kuuluvia henkilöitä. (Schwaber ym 2016, Tuovinen 2017 & Agile Alliance 2015b)

Kaikki sprintin aikana tehdyt toiminnallisuudet tulisi olla siinä kunnossa, että niiden voidaan sanoa olevan valmiita käytettäväksi. Sprintin jälkeen pidetään sprint retrospective -palaveri, jossa käydään tiimin sisällä läpi mitä voitaisiin parantaa. Parannettavat asiat voivat liittyä työtapoihin, prosesseihin tai melkein ihan mihin vain. Kun sprintti on saatu valmiiksi, aloitetaan uusi sprintti aivan samalla tavalla kuin edellinenkin ja tätä jatketaan niin kauan, että ohjelmiston katsotaan olevan valmis. (Schwaber ym. 2016, Tuovinen 2017 & Agile Alliance 2015b)

Neljäs vaihe pitää sisällään ohjelmiston julkaisemisen tuotantokäyttöön. Riippuen ohjelmistosta, näitä julkaisuja voi olla joko vain kerran tai mahdollisesti jopa jokaisen sprintin jälkeen (Tuovinen 2017).

Schwaber ym. (2016) totetaa, että on hyvin mahdollista ottaa käyttöön vain osan Scrumin menetelmistä ja rooleista mutta tällöin ei ole enää kyseessä Scrum. Tälle menetelmälle on oma terminsä ScrumBut. Scrum.org (2017) määrittelee ScrumButin siten, että projektissa käytetään Scrumia mutta jokin asia ei toimi niin se jätetään tekemättä tai tehdään toisella tavalla kuin Scrumissa alun perin. Esimerkiksi käytämme Scrumia mutta emme pidä päivittäisiä statuspalavereja joka päivä, koska siihen menee liian kauan aikaa.

2.2 Kanban

Sana kanban on japania ja se tarkoittaa kirjaimellisesti visuaalista korttia tai taulua (Andersson & Carmichael 2016). Tässä kappaleessa Kanbanilla tarkoitetaan ohjelmistokehitykseen käytettävää prosessia. Kanbania voisi hyvinkin käyttää myös muillakin aloilla. Alun perin autoyhtiö Toyota kehitti sen optimoidakseen autojen rakennusta. Järjestelmän periaatteina oli tee vain se mikä tarvitaan ja silloin kun sitä tarvitaan. (Toyota Motor Corporation 2017). Hukan vähentäminen on myös olennainen osa tätä Toyota Production Systemiä. Agile Alliancen (2015c) mukaan Kanbanilla annetaan tavat, joilla virtaavien kehitysprojektien kehitystä voidaan parantaa. Virtaavalla projektilla tarkoitetaan sitä, että työtehtävien voidaan katsoa virtaavan läpi projektin tiettyjen vaiheiden läpi. Virtaus (flow) onkin yleinen termi, jolla kuvataan tehtyä työtä,

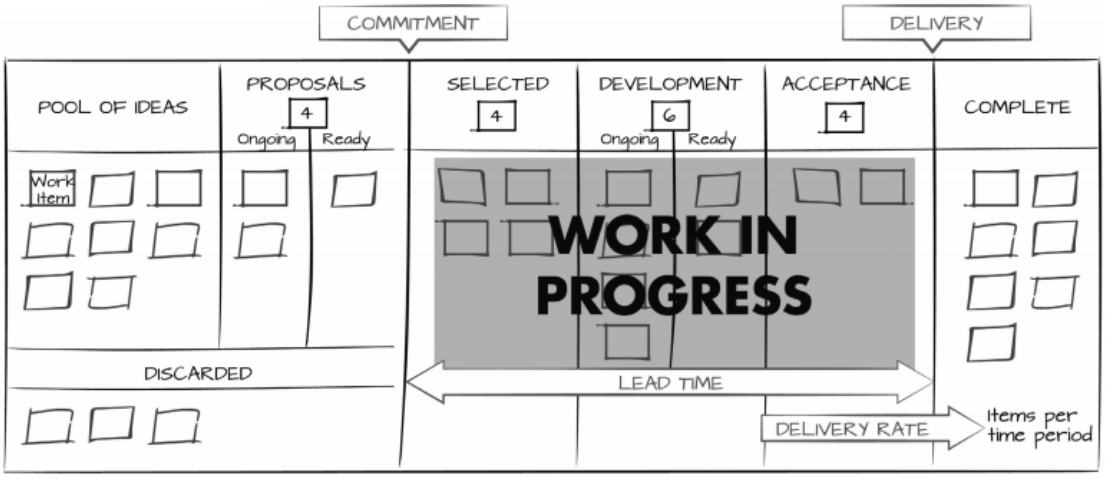
joka virtaa läpi projekti ilman, että sitä organisoidaan johonkin tiettyyn julkaisuun tai muuhun kategoriaan (Agile Alliance 2015c).

Andersson & Carmichael (2016) mukaan ihmiset vastustavat muutosta. Kanbanin kolme muutosten hallinta pääperiaatetta onkin kehitetty tämä mielessä. Aloita sillä mitä teet nyt. Tämä tarkoittaa sitä, että prosessit on ymmärrettävä ja kuinka niitä hoidetaan. Kunnioita nykyisiä rooleja velvollisuuksia ja titteleitä ja tavoittele parannusta toimintoihin vähittäisen muutoksen kautta.

Yksi Kanbanin pääperiaatteista on saada tietty tehtävä kulkemaan koko prosessin läpi mahdollisimman nopeasti. Valmiin tehtävän on tarkoitus tuottaa arvoa asiakkaalle, täten voidaankin katsoa, että Kanbanin keskiössä on siten asiakas (Agile Alliance 2015c). Andersson & Carmichael (2016) katsovat, että työtehtävien määrää on rajoitettava, jotta työn alla olevat tehtävät saadaan toimitettua nopeasti. Rajoitusten määrä on hyvin riippuvainen projektin työntekijöiden määrästä ja siitä, kuinka paljon eri työvaiheita yhdellä tehtävällä järjestelmässä on. Tämä tehtävän työn rajoittaminen vähentää kehittäjien tarvetta vaihtaa turhaan työtehtäviä, joka vie yleensä paljon aikaa ja energiaa. Rajoitukset luovat myös luonnollisen vetosysteemin. Tällä tarkoitetaan sitä, että työtehtävät vedetään mukaan kehitykseen, kun jollakin kehittäjällä on aikaa ottaa työ vastuulleen. Normaalien projektien voidaan katsoa toimia päinvastaisesti. Työtehtävä työnnetään projektiin sisään, riippumatta siitä onko kenelläkään aikaa sitä toteuttaa. Rajoitukset tuovat helposti näkyviin prosessissa olevia ongelmia, sillä jos prosessia on jokin ongelma alkavat työtehtävät yleensä kasaantua tiettyyn kohtaan prosessia.

Kanbanin on tarkoitus mahdollistaa prosessin parantaminen. Jotta tällainen on, mahdollista, täytyy prosessi ymmärtää. Ymmärtäminen pitää sisällään projektin läpinäkyväksi tekemisen. Lisäksi prosessia ja sen vaiheita täytyy tulkita (Lehtonen Teijo, Seppo Tuomivaara, Ville Rantala, Marja Känsälä, Tuomas Mäkilä, Tero Jokela, Kaisa Könnölä, Matti Kaisti, Samuli Suomi, Minna Isomäki & Marko Ylitolva 2014). Prosessia kuvataan yleensä visuaalisella taululla, jossa näkyvät työtehtävät ja niiden tilat. Taulussa voi projektista riippuen olla hyvinkin erilaisia tiloja. Lisäksi tilojen määrää ei ole mitenkään pakotettu tiettyyn määrään. Power (2014) kertoo, että heidän projektissaan

käytettiin tiloja suunniteltu, valmis, työn alla, valmis ja hyväksytty. Yksi mahdollinen esimerkki taulusta on esitetty kuvassa 2 (Andersson & Carmichael 2016). Tehtävät lähtevät taulun vasemmasta laidasta liikkeelle, josta ne siirtyvät eteenpäin oikealle aina kun edellinen tila valmistuu.



Kuva 2 Kanban-taulu (Andersson & Carmichael 2016).

Taulun vasempaan laitaan tulee tila, jossa tehtävät ovat vasta idean asteella tai tehtävät ovat jo jollain tasolla määritelty. Tämä lista voi olla järjestetty prioriteettien mukaan mutta se ei ole lainkaan pakollista. Kyseisellä tilalla voi olla useita nimiä, esimerkiksi työlista (backlog), suunniteltu tai ehdotukset.

Seuraavassa vaiheessa tehtävä yleensä määritellään tarkemmin. Se mahdollisesti pilkotaan pienempiin osiin, hylätään kokonaan, suunnitellaan tarkemmin tai tehdään kuten ehdotettu. Edellä mainitut vaihtoehdot eivät ole lainkaan ainoat mahdollisuudet. Tilaa voitaisiin kutsua esimerkiksi erittelyksi tai valituksi. Tilan tarkoitus on valmistella tehtävä toteutukseen. On huomioitavaa, että riippuen projektista tätä tilaa ei välttämättä ole lainkaan (Andersson & Carmichael 2016). Tilan voi korvata myös yksinkertainen valittu tai valmis. Nämä tilat indikoivat, että tehtävät ovat valmiita toteutukseen.

Tilaan toteutuksessa, työn alla tai kehityksessä kuuluu usein varsinaisen toteutuksen ohella myös muutakin, yleensä testausta ja testien kirjoittamista. Kun kehittäjä kokee tehtävän olevan, valmis siirtyy se tilaan valmis, valmis testattavaksi tai jotain vastaavaa. Työtehtävä ei yleensä vielä ole valmis vaan se vaatii vielä testaamista. Kyseinen tila ei ole välttämätön mutta se on melko usein käytössä (Lehtonen ym. 2014), koska sillä saadaan erotettua helposti erotettua tehtävät, joiden toteutus on valmis mutta ei välttämättä vielä testattu.

Yleensä viimeisenä oleva tila on ratkaistu, hyväksytty tai valmis. Tässä tilassa työtehtävä on kokonaan hoidettu valmiiksi. Kun kehittäjä on siirtänyt tehtävän tähän tilaan, hän ottaa jonkun uuden tehtävän työn alle taulun vasemmasta reunasta. Tähän on muitakin vaihtoehtoja riippuen mitä projektissa on sovittu. On myös mahdollista, että testaus on priorisoitu ja kehittäjä ottaa jotain testaustilasta.

On tärkeää seurata ohjelmiston kehityksen kulkua. Power (2014) esittää, että on metriikkaa, joka kertoo mahdollisista tulevaisuudennäkymistä ja toinen kertoo kehityksen historiasta. Molemmista saa tärkeää tietoa. Tulevat indikaattorit auttavat ennakoimaan mahdollisia ongelmia tai pullonkauloja. Historiatiedosta voidaan oppia ja korjata jo toteutuneet ongelmat.

Läpimenoaika kuvaa sitä kuinka kauan yksittäisen tehtävän kestää mennä kaikkien tilojen läpi. Se siis kuvaa aikaa ideasta valmiiksi toiminnallisuudeksi. Power (2014) kertoo, että seuraamalla läpimenoaikaa he saivat paremman käsityksen siitä, kuinka kauan tehtävien valmiiksi saattaminen kestää. Tämä antoi heille saman käsityksen mitä heidän asiakkaansa näki. Seuraamalla läpimenoaikaa voidaan arvioida kuinka kauan vastaavien toiminnallisuuksien tekeminen kestää. Ongelmana voidaan nähdä, että kun tehdään monimutkaisia asioita niin niiden ennustaminen ei onnistu suoraan läpimenoajan perusteella.

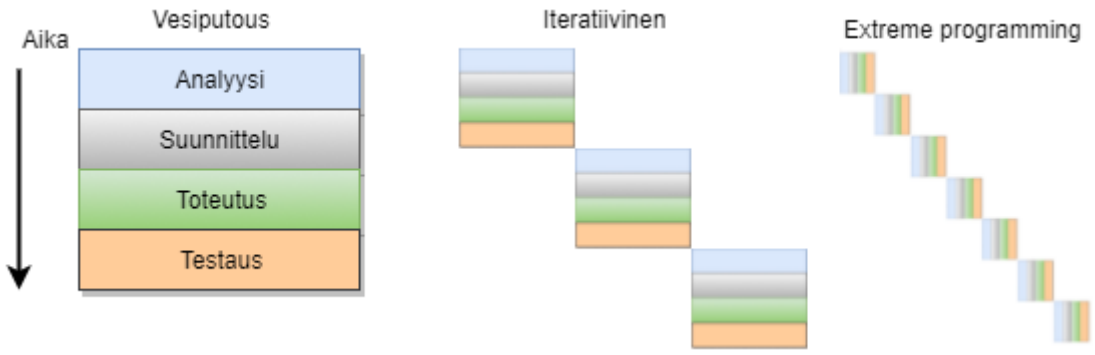
Kumulatiivinen virtauskaavio (CFD) esittää kuinka paljon työtehtäviä on eri vaiheessa tiettynä ajanhetkenä. Power (2014) näkee kaavion hyvänä työkaluna niin tulevan ennustamiseen kuin historiankin seuraamiseen. Kaaviosta voidaan lukea, kuinka hyvin tiimi on toiminut. Kaaviosta näkee myös hyvin pullonkauloja. Kaaviosta voi ennustaa,

koska tiimi saa valmiiksi tietyn verran tehtäviä. Ennustamisessa täytyy kuinkin olla hyvin varovainen, sillä se ei kerro mitään työn alla olevien tehtävien monimutkaisuudesta (Power 2014).

2.3 Extreme programming

Ohjelmistokehityksessä yksi ongelmallisimpia asioita on muuttuvat vaatimukset ja tarpeet. Ohjelman tilaaja pystyy harvoin listaamaan kaikki mahdolliset toiminnot, joita hän ohjelmaan haluaa tai tarvitsee. Lisäksi täydellisisä vaatimuksia on usein vaikea saada paperille muuttumattomina. (Wells 2009). XP eli extreme programming yrittää ratkaista muun muassa näitä ongelmia. Se on ketterä tapa kehittää ohjelmistoja käyttäen hyväksi useita hyväksi todettuja käytäntöjä (Beck 1999).

Beck (1999), kuvaa artikkelissaan kuinka kehitystä tehdään pieninä paloina jatkuvasti. Kuvassa 3 Beck esittää eroa vesiputousmallin, iteratiivisen spiraalimallin ja extreme programming –menetelmän välillä. Yksi kehityssykli voi kestää muutamista tunneista viikkoihin riippuen sisällöstä.



Kuva 3 Extreme programming suhteessa muihin kehitysmalleihin (Beck 1999).

Extreme programming nojaa hyvin vahvasti tiimityöhön. Tiimi pyrkii löytämään ongelmiin aina mahdollisimman hyviä ratkaisuja. Tiimityö korostuu varsinkin pareittain

tehtävässä ohjelmoinnissa. Kommunikointi on suuressa roolissa myös. Ilman kunnon kommunikointia tiimi ei voi toimia kunnolla. Kommunikoinnin helpottamiksesi ja väärinymmärryksien vähentämiseksi kehitystiimi sijaitsee samassa tilassa. Tilassa toimii myös asiakkaan edustaja, jolta saa tarvittaessa tilaajan näkemyksen. (Wells 2009)

Ohjelmiston toimivuuden mittarina on testit. Extreme programming onkin testilähtöistä kehitystä. Testit kirjoitetaan ennen varsinaista toteutusta. Kaikki toiminnot testataan ja kaikkien testien on mentävä läpi, jotta uudet toiminnot voidaan yhdistää mukaan ohjelmistoon. Yhdistäminen tapahtuu integroinnin kautta ja tämän voi tehdä vain yksi kehittäjä kerrallaan. Syy tälle on se, että kaikki uusi koodi tulee näin testattua erikseen eikä kahdesta eri paikasta tullutta uutta koodia jouduta testaamaan samaan aikaan. (Wells 2009) Taulukossa 1 on esitetty extreme programming –menetelmän pääperiaatteet.

Taulukko 1. extreme programming –menetelmän pääperiaatteet (Beck 1999).

Suunnittelupeli	Asiakas valitsee tehtävät vaatimukset ja vain ne toteutetaan.
Pienet julkaisut	Järjestelmä laitetaan tuotantoon ennen kuin se on kokonaan valmis. Uusi julkaisuja tehdään usein.
Metaforien käyttö	Järjestelmän kuvauksissa voidaan käyttää metaforia, jotta asioita olisi yksinkertaisempi selittää.
Yksinkertainen suunnittelu	Suunnittelu perustuu yksinkertaisuuteen. Tee vain yksinkertaisen asia, joka ratkaisee ongelman.
Testit	Kaikki toiminnot yksikkötestataan. Asiakas tekee toiminnalliset testit käyttötapauksille.
Uudelleenkirjoitus	Ohjelmistoa parannetaan uudelleen kirjoittamalla ja koodia muokkaamalla. Testien on mentävä läpi muutoksien jälkeen.
Pariohjelmointi	Kaksi henkilöä yhdessä tuottaa tuotantoon menevän koodin.
Jatkuva integrointi	Uusi koodi integroidaan ohjelmistoon mahdollisimman nopeasti. Testien täytyy mennä läpi tai uutta koodia ei hyväksytä.
Yhteinen omistajuus	Kehittäjät korjaavat virheitä mistä tahansa koodista, jos näkevät sen olevan hyödyksi

Jatkuu...

...Jatkuu.

Asiakas paikalla	Asiakas on kehitystiimin saatavilla koko ajan.
Ei ylitöitä	Ylitöitä ei saa tehdä kahta viikkoa putkeen. Ylityön tekeminen kertoo yleensä ongelmista prosessissa.
Yhteinen työtila	Kehittäjät työskentelevät yhteisessä työtilassa.
Säännöt (just rules)	Kehittäjät sitoutuvat noudattamaan yhteisiä sääntöjä. Sääntöjä voidaan tarpeen mukaan muuttaa, kunhan sovitaan, kuinka asiat hoidetaan jatkossa.

3 TEKNIIKAT

Tässä kappaleessa on kuvattu päätekniikat, joita ohjelmistossa on käytetty.

3.1 XML ja XML-skeema

Extensible Markup Language eli XML W3C:n kehittämä avoin tiedon kuvauskieli. Sillä voidaan kuvata tietoa ihmisen ja koneiden ymmärtämässä muodossa. XML-kielen suunnitteluperiaatteisiin kuului muun muassa se, että se on helposti käytettävissä internetissä, XML-dokumentit ovat helposti luotavissa ja on oltava helppo kirjoittaa ohjelmia, jotka prosessoivat XML-dokumentteja. (W3C 2008).

XML-dokumentti on rajoittuneempi versio SGML:stä, koska se kuvaa osittain SGML-standardin. Se tarkoittaa standardia yleistä merkkauskieltä ja se on myös ISO-standardi. (W3C 2008)

XML-dokumentilla on muutamia sääntöjä, joita sen on noudatettava. Dokumentti voi olla oikean muotoinen (valid) ja hyvin muodostettu (well formed). Oikean muotoisen dokumentin määrittää tyyppimäärittely, joka on esiteltävä ennen ensimmäistä XML-elementtiä. Tyyppimäärittelyllä voidaan antaa rajoitteita XML-rakenteeseen. Tyyppimäärittely pakottaa dokumentin sisällön olemaan määrityksessä kuvatun mukainen. Jos määrittely ja varsinainen XML-dokumentti eivät vastaa toisiaan ei dokumentti ole oikein muodostettu. Sellaisessa dokumentissa on myös vain yksi juurielementti. Oikein muodostettu dokumentti seuraa myös XML-syntaksia. XML-elementtien aloitus- ja lopetustagien on oltava oikeassa järjestyksessä. Alkavaa tagia pitää aina seurata lopettava tagi ennen kuin uuden alkavan tagin voi luoda. Tagit voivat olla sisäkkäisiä. Jälkimmäisenä esitelty tagi pitää sulkea ennen aikaisemmin esiteltyä tagia. (WC3 2008)

XML-dokumentin ei ole pakko olla validi se voi olla vain hyvin muodostettu. Sellainen dokumentti on syntaksiltaan oikea XML-dokumentti mutta siltä voi puuttua

tyyppimäärittely tai sen sisältö ei vastaa tyyppimäärittelyä. Jos dokumentti ei ole hyvin muodostettu se ei ole silloin XML-dokumentti. (W3C 2008)

XML-dokumenttia tulkkaavien prosessien täytyy pitää huoli siitä, että tulkattava XML on oikein muotoinen tai hyvin muodostettua. Jos se ei ole kumpaakaan, täytyy prosessien antaa virhe tai kohtalokas (fatal) virhe riippuen tilanteesta. Normaalista virheestä prosessi voi toipua ja jatkaa mutta kohtalokas virhe lopettaa XML-dokumentin käsittelyn siihen paikkaan. (WC3 2008)

XML-skeema on tapa määritellä XML-dokumentin rakenne ja sisältö. Skeema itsessään on XML-dokumentti, joten sen on toteutettava samat säännöt kuin normaalinkin XML-dokumentin. Skeemalla on yleensä kaksi eri tehtävää. Sen avulla voidaan tarkistaa, onko XML-dokumentissa skeemassa määritellyn kaltainen sisältö tai skeeman avulla voidaan luoda XML-dokumentti juuri halutun kaltaiseksi. (WC3 2004)

Skeemassa voidaan määrittää minkä nimisiä elementtejä, missä järjestyksessä ja kuinka paljon niitä pitää olla. Elementtien arvo ja minkä tyyppistä data on, voidaan myös määrittää skeemassa. Skeema rakentuu yksinkertaisista ja monimutkaisista elementeistä. Yksinkertainen elementti on yksi tietorakenne, esimerkiksi merkkijono. Monimutkainen rakenne pitää sisällään useita yksinkertaisia rakenteita, esimerkiksi päivämäärän ja merkkijonon. (W3C 2004)

3.2 Qt

Qt on alustariippumaton ohjelmistokirjasto ja kehitysympäristö. Sen alkuperäisillä kehittäjillä oli tarve tehdä graafinen ohjelmisto, joka toimisi Unixissa, Windowsissa ja Macintoshissa. Tämä tarve loi perustan Qt:n idealle ja mahdollisti sen kehityksen (Blanchette, J & Summerfield, M. 2008: 13-14). Qt:lla on ollut vuosien mittaan useita eri omistajia mutta alun perin Qt:n on kehittänyt Norjalainen Trolltech. Qt:n ensimmäinen julkinen versio julkaistiin toukokuussa 1995. Nokia osti Trolltechin vuonna 2008. Nokialla oli useita Symbian-käyttöjärjestelmällä varustettuja puhelimia ja yksi

MeeGo/Harmattan-puhelin, joissa molemmissa Qt toimi ohjelmien kehitysympäristönä. Nokian valitessa Windows Phonen pääasialliseksi käyttöjärjestelmäkseen puhelimiin tuli Qt:sta Nokialle taakka ja se myytiin Digialle vuonna 2012. The Qt Company erityi Digiasta omaksi yrityksekseen 2014. The Qt Company on Digian tytäryhtiö. (Qt Company 2016a)

Monet mieltävät Qt:n pelkästään käyttöliittymien suunnitteluun tarkoitetuksi alustaksi. Qt kyllä pitää sisällään peruskomponentit, joilla voidaan rakentaa monimutkaisiakin käyttöliittymiä työpöytäkäyttöön, mutta Qt on paljon muutakin. Qt sisältää tietokantojen käsittelytoiminnot, XML-kirjaston, verkkokirjaston ja paljon muuta. Qt:lla voi käytännössä tehdä lähes kaiken mitä työpöytäsovellukselta voi odottaa. Qt ei tosin ole rajoittunut työpöytäkäyttöön, vaan sitä voidaan käyttää myös sulautetuissa laitteissa ja esimerkiksi autoissa. (Qt Company 2017)

Qt:n suurimmaksi hyödyksi voidaan sanoa sen alustariippumattomuuden. Sama lähdekoodi toimii useilla alustoilla muokkaamatta tai lähes muokkaamatta. Mobiilijärjestelmien ja työpöytäjärjestelmien ristiin käyttäminen on mahdollista, mutta käytettävyys ei välttämättä ole tällöin optimaalinen kaikille järjestelmille. Qt tukee tällä hetkellä käytännössä kaikkia moderneja käyttöjärjestelmiä, olivat ne sitten mobiililaitteissa tai normaaleissa tietokoneissa. Muutamia esimerkkejä on Windows, Linux, Android ja IOS. (Qt Company 2016d)

Toisena isona hyötynä on se tosiasia, että Qt:lla kehitetty ohjelmisto käyttää lopulta natiivia koodia ja on täten nopeampi kuin vastaava hiekkalaatikolla ajettava ohjelma. Kehityskielenä toimii yleensä C++, myös Pythonia voi käyttää (PySide documentation 2016). C++-kielen käyttö tuo myös omat ongelmansa, sillä kehittäjän täytyy itse hallita muistinkäyttöä. Erilaiset fiksut pointterit vähentävät kehittäjän tarvetta huolehtia muistinkäytöstä ja täten vähentävät ongelmia, joita pointterien huolimaton käyttö normaalisti aiheuttaa (Macieira 2009).

Qt-ohjelmistoja voi kehittää usean eri lisenssin alla (Qt Company 2016b). Siitä on saatavilla kaupallinen versio, jolloin Qt:n komponentteihin voi tehdä haluamiaan muutoksia, eikä niitä tarvitse jakaa open source -yhteisölle. Lisäksi on tarjolla LGPL-

lisenssin alainen versio, jolloin kehitysympäristön saa ilmaiseksi käyttöön, mutta tällöin on muutamia rajoitteita, joita pohtimaan joutuvat varsinkin kehittäjät, jotka haluavat pitää lähdekoodinsa suljettuna. Kirjastoon ei saa tehdä muutoksia ilman, että sen julkaisee vapaaseen käyttöön. Lisäksi staattinen linkkaaminen kirjastoon ei ole sallittua, sillä silloin kirjoitettu lähdekoodi muuttuu LGPL-lisenssin alaiseksi (Qt Company 2016c). Yrityksille tehtävän räätälöidyn ohjelman kanssa avoimen lähdekoodin ratkaisu tulee hyvin harvoin kyseeseen.

4 TESTAUS

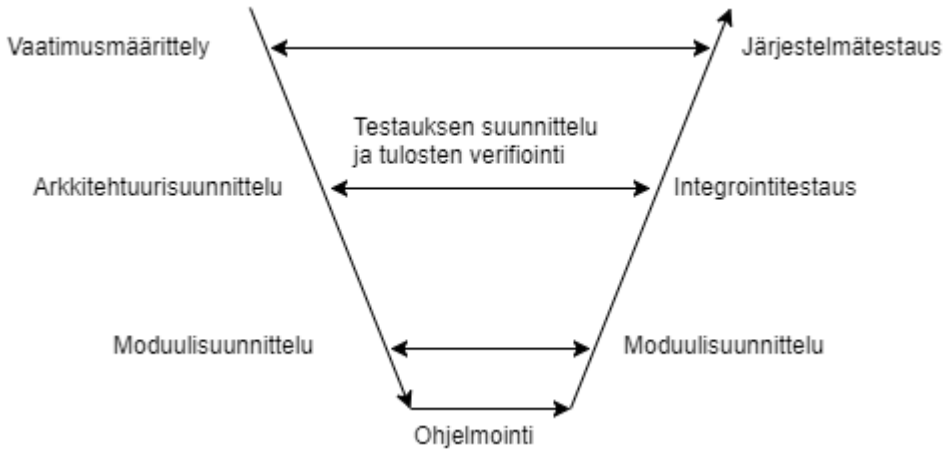
Testauksesta puhuttaessa on ensin hyvä määritellä muutama termi. Nämä termit ovat virhe (error, mistake, bug), vika (fault, defect) ja häriö (failure). Virhe on ohjelmiston poikkeamista määrittelystä. Perry (1999:6-7) määrittelee vian sisältyvän ohjelmistoon, mutta sillä ei ole vaikutusta ennen kuin se vaikuttaa käyttäjään jollain tavalla. Vika on täten virheellisen ohjelmakohdan suorittamista. Vika, joka vaikuttaa käyttäjään on taas häiriö (Perry 1999:6-7). Häiriön voitaneen siis katsoa olevan vian erityistapaus. Perry (1999:6-7) katsoo, että vika yleisimmin johtuu puutteellisesta toteutuksesta verrattuna määrittelyyn, toteutus puuttuu ohjelmistosta kokonaan tai jotain on toteutettu ohjelmistoon, vaikka sitä ei ole määrittelyssä vaadittu.

Myers (1976) esittää testaamisen prosessina, jossa ohjelmistosta löydettäisiin virheitä. Hänen mukaansa testitapaus, jossa ohjelmisto suorittaa oikean lopputuloksen ilman virheitä on epäonnistunut testi. Myersin mukaan testaajien tulisi pyrkiä testaamaan siten, että etsitään ongelmia eikä pyritä tarkistamaan, että ohjelmisto toimii kuten pitää. Myers pitää testaamista tuhoavana, jopa sadistisena prosessina ja täten tämä selittää sen miksi monet ihmiset kokevat sen vaikeaksi.

Dijkstra (1970) esittää, että testaamalla voidaan todistaa virheen olemassaolo mutta ei koskaan niiden puuttumista. Jotta voidaan olla aivan varmoja, että koodissa ei virheitä täytyy testata kaikki mahdolliset syötteet. Kaikkien syötteiden testaaminen ei ole lainkaan järkevää, sillä tähän kuluu valtavasti aikaa. Dijkstra (1970) antaakin esimerkin, että yksinkertaisessa kahden luvun kertolaskussa menisi 10000 vuotta kun kaikki mahdolliset vaihtoehdot käydään läpi. Nykyisin laskentatehoa on reippaasti enemmän kuin Dijkstran aikaan, mutta nykyiselläkin laskentateholla tällainen laskutoimitus kestäisi toivottoman kauan. Kantava ajatuksena kuitenkin on se, että koska kaikkea syötteitä ei ole mahdollista testata, on syytä valita testattavat arvot järkevästi.

Ohjelmistoja onkin syytä testata koko sen kehitysprosessin ajan. Kuvassa 4**Virhe. Viitteen lähdettä ei löytynyt.** on esitetty ohjelmistokehityksen v-malli. Tämän kyseisen

tavan linkittää ohjelmiston eri kehitysvaiheet ja eri testauksen vaiheet on esittänyt Forsberg ja Mooz (1995).



Kuva 4 Ohjelmistokehityksen v-malli (Forsberg ym. 1995).

V-mallissa ohjelmiston kehitysprojekti jaetaan kahteen osaan, suunnitteluun ja testaukseen. Näiden kahden välille jää varsinainen ohjelmiston toteutus. Mallissa on useita vaiheita, jossa jokaista suunnitteluvaihetta vastaa aina testausvaihe. Näin jokainen suurempi suunnitelma tulee myös testattua. Testit määritellään samaan aikaan suunnittelun kanssa. Testit ovat täten ajan tasalla koko ajan. V-malli nojaa hyvin vahvasti perinteiseen vesiputousmalliin tuoden siihen testauksen osaksi jatkuvaa prosessia. Taina (2009) listaa mallin hyviksi puoliksi sen, että testausta tehdään koko projektin aikana eikä vain projektin lopussa, budjetin loppuessa ohjelmistoa on jo testattu ja regressiotestauksen helppo suorittaminen testausdokumentaation avulla.

V-mallissa nähdään myös huonoja puolia (Taina 2009), jotka ovat pitkälti samoja kuin vesiputousmallissa. Näitä on muun muassa kankea eteenpäin menevä prosessi, joka ei salli vaatimusten muuttumista. Asiakas ei näe valmista ennen kuin projekti on lopussa. Aikataulun pettäessä projektia on vaikea saada takaisin aikatauluun. Dokumentaatiota tehdään enemmän kuin vesiputousmalliin perustuvissa projekteissa.

4.1 Mustalaatikkotestaus

Mustalaatikkotestauksessa ohjelmaa ajatellaan mustana laatikkona (Myers 1976). Testaaja ei tiedä ohjelmiston sisäisestä toiminnasta mitään. Testaaja on kiinnostunut löytämään toiminnallisuudet, jotka eivät toimi kuten ne on määritelty toimimaan. Testitapaukset perustuvat täysin ohjelmiston määrittelyyn. Testaaja voi vaikuttaa sisään menevään syötteeseen ja nähdä lopputuloksen.

Williams (2006) näkee, että mustalaatikkotestauksella pyritään testaamaan virheellisiä tai puuttuvia toiminnallisuuksia, rajapintojen virheitä, virheitä rajapintojen käyttämissä datan rakenteissa, ohjelman omituiseen käyttäytymiseen tai tehokkuuteen liittyviä ongelmia ja lopuksi alustukseen ja lopetukseen liittyviä asioita. Kun näitä asioita testataan, voidaan määritellä, toimiiko ohjelmisto määrittelyjen mukaan. Williams (2006) katsoo, että testin kirjoittajan olisi syytä olla eri henkilö kuin ohjelmiston kirjoittaja. Mielellään sellainen henkilö, joka ei tiedä lainkaan lähdekoodin logiikkaa. Ohjelmoija kirjoittaisi todennäköisesti testin, joka testaa sen mitä ohjelma tekee. Toinen henkilö saattaa kirjoittaa testin siitä mitä asiakas haluaa. Asiakkaan vaatimukset ovatkin yksi asia, joita tulisi testata hyväksymistesteissä.

4.2 Lasilaatikkotestaus

Lasilaatikkotestauksessa (Paakki 2003) testaajalla on käytössään ohjelmiston lähdekoodi. Lasilaatikkonimitys tulee siitä, että ohjelmistoa ajatellaan laatikkona, jossa on läpinäkyvät seinät. Seinien läpi nähdään ohjelmiston sisäinen logiikka. Lasilaatikkotestausta kutsutaan myös rakenteelliseksi testaamiseksi. Testauksessa syötteet valitaan siten, että ohjelmiston eri haarat käydään läpi.

Paakki (2003) listaa erilaisia kattavuusmittoja sille kuinka kattavasti ohjelmiston eri polut käydään läpi. Polkukattavuudessa käydään kaikki mahdolliset polut läpi. Tällaisen tason saavuttaminen on käytännössä hyvin harvinaista. Täydellinen lausekattavuus saavutetaan

(Naik & Tripathy 2008) jos kaikki lauseet on ajettu vähintään kerran. Päätöskattavuudessa jokaisen ehdon kaikki eri vaihtoehdot käydään läpi.

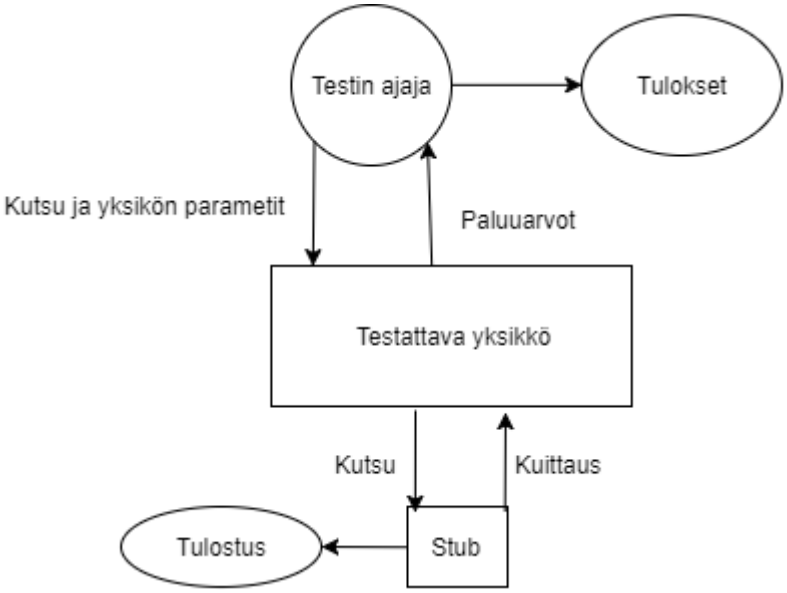
4.3 Yksikkötestaus

Yksikkötestauksessa tutkitaan yleensä ohjelmiston pienimpien osien toimintaa. Osa voi olla esimerkiksi moduuli, luokka tai metodi. Näitä osia testaamalla varmistutaan siitä, että pienet osakokonaisuudet toimivat oikein. Tässä kohtaa ei vielä olla kiinnostuneita suurempien kokonaisuuksien testaamisesta. Naik ym. (2008) kertovat, että yksikkötestausta on kahdenlaista: staattista ja dynaamista.

Staattinen yksikkötestaus voidaan jakaa kahteen osaan, katselmointiin ja läpikäyntiin. Naikin ym. (2008) mukaan katselmointi on tuotteen läpikäyntiä askel kerrallaan muiden kuin tuotteen tekijän toimesta. Jokaisella askeleella on ennalta määrätyt tavoitteet. Läpikäynnin esitteli Fagan (1999). Läpikäynnissä koodin kirjoittaja käy läpi tiimin kanssa ohjelman toteutuksen manuaalisesti tai simuloidusti käyttäen ennalta määrättyjä tilanteita. Molemmissa tavoissa lopputulos on lopulta sama. Ohjelmiston toteutus on käyty läpi muidenkin kuin koodin kirjoittaneen henkilön toimesta. Henkilöt, jotka käyvät koodia läpi ilmoittavat mahdollisista ongelmista. Nämä ongelmat tarpeen mukaan korjataan, yleensä korjauksesta vastaa koodin alkuperäinen kirjoittaja. Mahdollisia ongelmia voi olla esimerkiksi looginen virhe ohjelmiston logiikassa, riittämätön dokumentointi kompleksisessa koodissa tai sovitusta tyylistä poikkeaminen.

Dynaamista yksikkötestausta kutsutaan myös ajettavaksi testaamiseksi. Dynaamisessa yksikkötestissä ohjelman ympärille rakennetaan testitapaus, joka kutsuu testattavaa yksikköä. Kuvassa 5 on esitetty tapa, jolla yksiköitä testataan erillisillä yksikkötesteillä (Naik ym. 2008). Testin ajaja kutsuu testattavaa yksikköä sopivilla parametreilla. Lopulta ajajalle saapuu myös paluuarvo. Paluuarvojen perusteella voidaan määrittää, onko testi ajettu onnistuneesti läpi vai ei. Yksikköä voidaan kutsua niin monta kertaa kuin on tarvetta, jotta kaikki tarpeelliset testitapaukset saadaan suoritettua. Testattava yksikkö voi kutsua toisia palveluita. Palvelut voivat olla toisia yksiköitä mutta yleensä käytetään

tynkiä (stub), jotka korvaavat toiset yksiköt. Näin vältetään mahdolliset ongelmat toisissa yksiköissä. Toiset yksiköt eivät välttämättä ole vielä valmiita tässä vaiheessa.

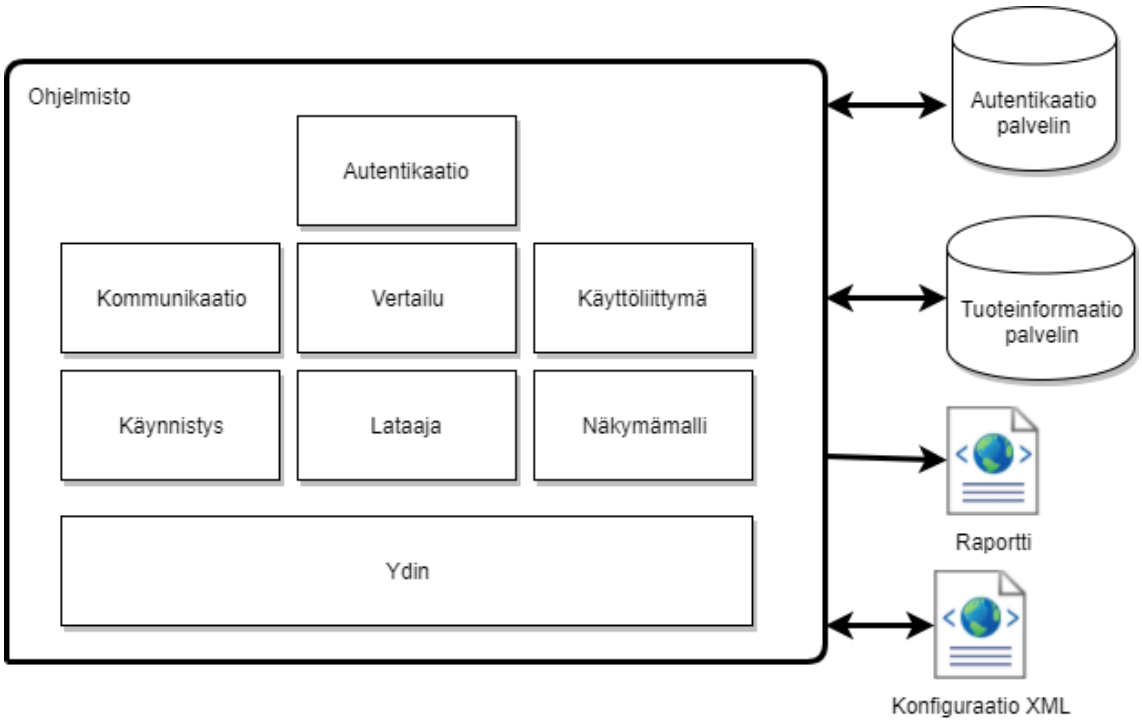


Kuva 5 Dynaaminen yksikkötestaus ympäristö (Naik ym. 2008).

Koodin kirjoittaja on yleensä yksikkötestien suorittaja. Monissa tapauksissa yksikkötesti kirjoitetaan samaan aikaan ohjelmiston koodin kanssa. Joissakin tapauksissa jopa etukäteen. Yksikkötestit voivat olla automaattisia tai niitä voidaan ajaa myös manuaalisesti. Kun automaattisia testejä ajetaan säännöllisesti tai jonkun muutoksen jälkeen nähdään helposti, että aiheuttiko muutos ongelmia jo tehtyihin yksikkötesteihin.

5 OLEMASSA OLEVAN OHJELMISTON KUVAUS

Toteutettu ohjelma on lisäkomponentti jo olemassa olevaan ohjelmistoon, joten sen täytyi noudattaa myös sen arkkitehtuuria. Ohjelmisto koostuu kokoelmasta erillisiä komponentteja. Niitä on kehityksen aikana kertynyt jo yli 200. Ne toimivat itsenäisesti mutta voivat kutsua toisen komponentin palveluja julkisten rajapintojen kautta. Keskeisin näistä komponenteista on singelton-suunnittelumallilla toteutettu lataaja. Sillä voidaan ladata muita komponentteja. Ohjelman käynnistyessä alustetaan vain muutama keskeinen komponentti. Näitä ovat ydin, lataaja, käyttöliittymä ja käynnistykseen liittyvät komponentit. Niitä ladataan tietokoneen muistiin tarpeen mukaan. Kuvassa 6 on esitetty ohjelmiston arkkitehtuuria korkealla tasolla.



Kuva 6 Ohjelmiston arkkitehtuuri korkealla tasolla.

Ohjelmiston käynnistyessä ladataan ydin, joka on vastuussa ohjelmiston tärkeimmistä toiminnallisuuksista. Se on vastuussa siitä, että käyttöliittymä ja muut tarvittavat komponentit ladataan. Ytimelle rekisteröidään paljon erilaisia palveluja, joista se pitää

kirjaa. Sen kautta hoidetaan myös hallittu ohjelmiston sammuttaminen. Käynnistys-komponentti hoitaa ohjelmiston käynnistykseen liittyviä tehtäviä. Ensimmäisenä se lataa ja alustaa kaikki komponentit, jotka toteuttavat käynnistysrajapinnan. Esimerkkinä sen muista tehtävistä on päivittää avattavaa konfiguraatiota annettujen sääntöjen mukaan.

Näkymämalli on vastuussa ohjelmiston XML-muotoisen konfiguraatiotiedoston käsittelystä. Näkymämallin vastuulla on myös ohjelmistosta löytyvien puurakenteiden rakentaminen. Ennen kuin ohjelmistolla voi tehdä mitään järkevää, täytyy sille antaa konfiguraatio. Ohjelma pystyy luomaan konfiguraation myös tyhjästä, mutta yleisen käytännön mukaan, kun uutta konfiguraatiota aletaan tehdä, otetaan pohjaksi olemassa oleva konfiguraatio koska siinä on jo perusasiat kunnossa. Konfiguraatiotiedosto on ohjelmiston tuottama lopputuote, sinne tallennetaan kaikki käyttäjän tekemät asiat. Tämä konfiguraatio lopulta ladataan automaatiojärjestelmään. Ohjelmiston näyttämät rakenteet ja käyttöliittymät tuotetaan dynaamisesti osittain tämän konfiguraation sisällön perusteella. Tämä kyseinen tiedosto voidaan tarvittaessa siirtää käyttäjältä tai koneelta toiselle.

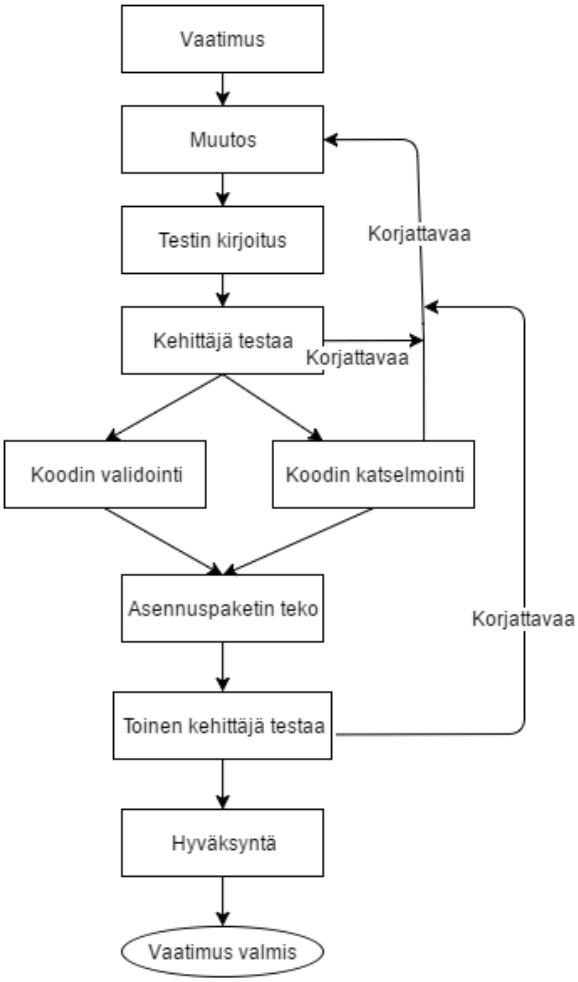
Kommunikaatiokomponentit hoitavat mahdollisia yhteyksiä erillisiin järjestelmiin tai laitteisiin. Osa niistä hoitaa myös ohjelmiston sisäistä viestintää. Erilaisiin yhteyksiin on rakennettu omat komponenttinsa koska kommunikointirajapinnat voivat olla hyvinkin erilaisia. Esimerkiksi yksi kommunikaatiorajapinnan toteuttava komponentti tarjoaa mahdollisuuden lukea XML-tiedostoja. Vertailukomponentin ainoa tehtävä on vertailla kahta asiaa keskenään ja raportoida muutokset näiden välillä. Tietystä tilasta toiseen vaihtaminen voi tuottaa tilanteen, jossa käyttäjä haluaa nähdä tehdyt muutokset tai kahden eri konfiguraationtiedoston eroavaisuudet voivat olla käyttäjää kiinnostavia asioita.

Ohjelmistoa voivat käyttää vain henkilöt, joilla on siihen oikeus. Tämä varmistetaan autentikointikomponentilla. Käyttäjäprofiilissa on määritelty käyttäjän eri tasot. Tasot määräävät mitä käyttäjä voi tehdä tai mitä hän saa nähdä. Käyttäjällä on oltava yhteys autentikointipalvelimeen tietyin väliajoin muuten ohjelmaan ei pääse kirjautumaan sisään. Tällä hetkellä palvelimeen on oltavat yhteydessä vähintään kahden viikon välein. Jos yhteyttä palvelimeen ei ole, käyttöoikeus tarkistetaan paikallisesta

väliaikaistiedostosta. Jos yhteyttä autentikointipalvelimeen ei ole muodostettu kahteen viikkoon, ei ohjelmistoon pääse enää sisään ennen kuin käyttäjä on tunnistautunut uudestaan palvelimella.

6 OHJELMISTON KEHITYSPROSESSI

Uuden toiminnallisuuden tai korjauksen kehitysprosessi on esitetty kuvassa 7. Pääperiaatteena prosessissa on, että toiminnallisuus on valmis, kun se on testattu ja toimivaksi todettu.



Kuva 7 Toiminnallisuuden kehitysprosessi.

Uusi toiminnallisuus tai korjaus lähtee liikkeelle uudesta vaatimuksesta tai bugiraportista, jossa kuvataan haluttu toiminto. Kehittäjä toteuttaa kyseisen toiminnallisuuden ja kirjoittaa sille testit. Tilanteesta riippuen testiksi voi riittää manuaalisesti ajettava uuden julkaisun yhteydessä ajettava testi tai sitten automaattisesti ajettava yksikkötesti. Testien

kirjoittamisen jälkeen kehittäjä itse testaa, että toteutettu toiminnallisuus läpäisee kirjoitetut testit. Tarvittaessa kehittäjä korjaa toiminnallisuutta, jotta se läpäisee testit. Kun kehittäjä on tyytyväinen toiminnallisuuteensa ja sen läpäistessä testit annetaan se eteenpäin. Vähintään yksi toinen kehittäjä katselmoi koodiin. Samaan aikaan koodille ajetaan automaattinen validointi tämä tarkoittaa käytännössä koodin kääntämistä. Täten saadaan automaattisesti palautetta mahdollisista syntaksivirheistä tai jos koodipohja on muuttunut versionhallinnassa eikä uuden toiminnallisuuden koodi ole enää yhteensopiva. Jos validoinnista tai katselmoinnista tulee jotain huomautettavaa, korjaa alkuperäinen kehittäjä nämä puutteet ja ajaa testit uudelleen. Kun katselmointi ja validointi on saatu hyväksytysti läpi, tehdään ohjelmistosta asennuspaketti automaattisesti. Tällä asennuspaketilla toinen kehittäjä ajaa samat testit kuin alkuperäinen kehittäjä, jotta varmistutaan, että toiminnallisuus oikeasti toimii. Tällöin päästään myös tilanteeseen, että toiminnallisuuden kehittäjä ei ole ainut henkilö, joka on testin ajanut. Hyväksytystä testistä seuraa toiminnallisuuden valmistuminen.

6.1 Scrum-kehitysprosessin käyttö projektissa

Ohjelmiston kehittämisvaiheessa käytettiin hyvinkin lähelle oikeanlaista Scrumia projektitiimin osalta. Projektin omistaja ei oikeastaan ollut mukana prosessissa niin paljon kuin Scrum edellyttää. Monet projektin omistajalle kuuluvat asiat jäivät Scrum-mestarin (Scrum master) harteille. Lisäksi vaatimukset kerättiin etukäteen käyttämättä Scrum-mallia. Muita erikoisuuksia verrattuna Scrumiin oli, että sprinttiin oli mahdollista ottaa uusia työtehtäviä sprintin ulkopuolelta. Tässä tosin noudatettiin yleensä hyvin linjausta, että jos jotain uutta otetaan sisään, täytyy jotain muuta pudottaa pois.

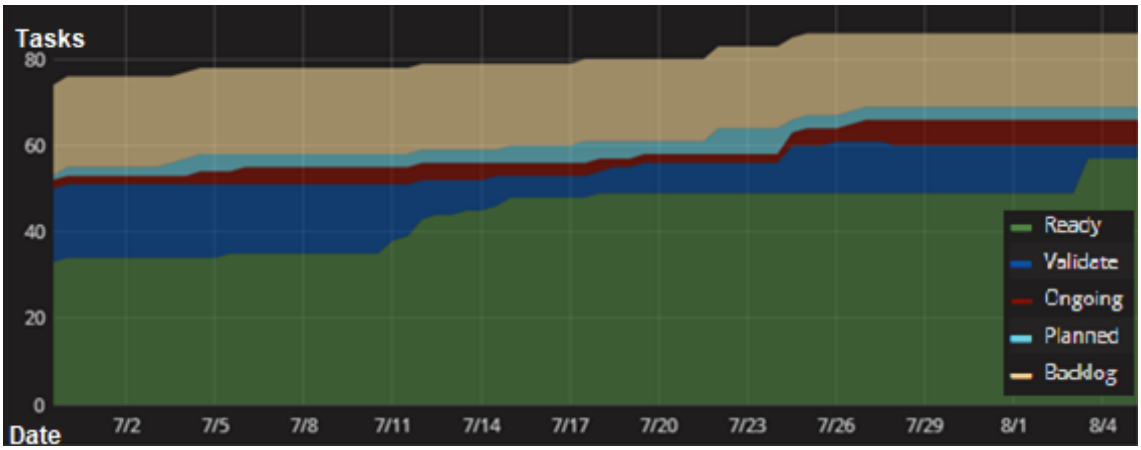
Mielestäni ohjelmiston kehittäminen toimi hyvin Scrumin käytäntöjä noudattaen. Kun projektissa vähennettiin kehittäjien määrää kahteen henkilöön, alkoi Scrumin käyttäminen tuntua hieman turhalta ja vaivalloiselta. Siitä siirryttiinkin hyvin yksinkertaiseen tehtävävetoiseen malliin, jossa tehtäviä priorisoitiin tuleviin ohjelmiston versioihin karkealla tasolla ja sen jälkeen tehtävät toteutettiin. Jos jotakin tehtävää ei

ehditty tehdä siirrettiin se seuraavaan versioon. Myös vaihtoehtoinen toiminto oli mahdollista, eli jos kaikki tehtävät tuli tehdyksi, otettiin listalta vain uusia tehtäviä työn alle.

6.2 Kanban-kehitysprosessin käyttö projektissa

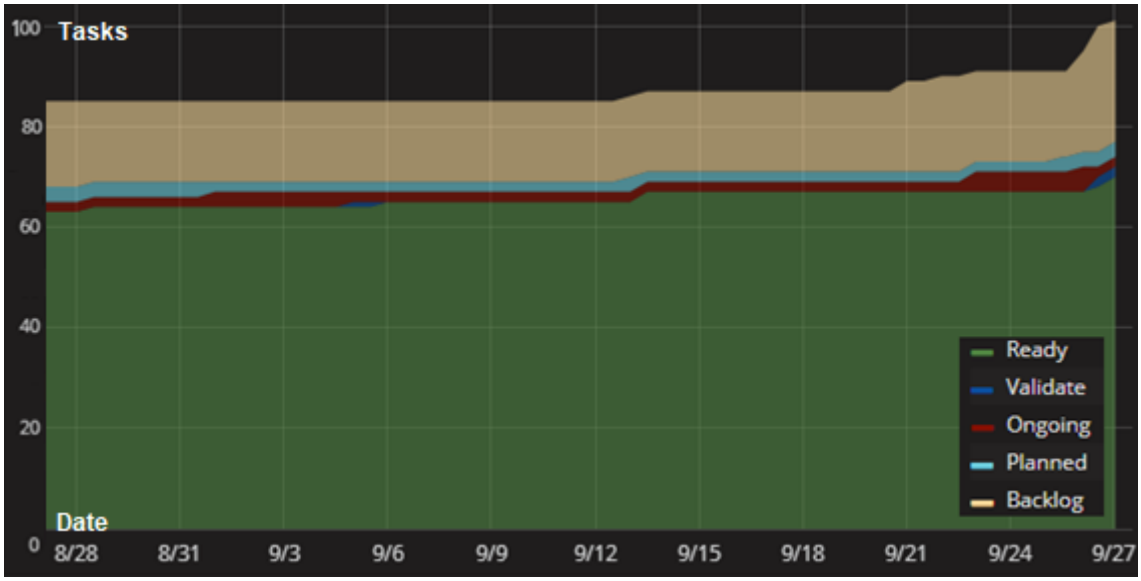
Projektissa otettiin käyttöön vuoden 2017 aikana Kanban-menetelmää, sillä ongelmana vanhassa prosessissa oli, että valmiiksi toteutetut työtehtävät jäivät pitkäksi aikaa odottamaan testaamista. Kanbanilla on tarkoitus parantaa näkyvyyttä siitä mitä kukin kehittäjä on tekemässä ja kokonaiskuvaa siitä, missä tilassa eri työtehtävät milloinkin ovat.

Kuvassa 8 on esitetty projektin tiloista automaattisesti luotu kumulatiivinen virtauskaavio ennen Kanbanin käyttöönottoa. Suurehkosta sinisestä alueesta korkeussuunnassa voidaan nähdä, että valmiiksi toteutetut tehtävät odottavat pitkän aikaa testausta. Lopulta ne tulevat testatuksi lähes samaan aikaan. Syy tällaiselle toiminnalle on ohjelmiston julkaisupäivä, sen lähestyessä kaikki oli lopulta testattava. Kuvasta käy myös ilmi se, että tehtäviä on työn alla liian monta samaan aikaan. Tämän voi nähdä katsomalla punaista aluetta. Punaisen värin pitäisi olla hyvin ohut pystysuunnassa mutta se on melko paksu varsinkin aikajanan loppupäässä. Tehtävät jäävät siis roikkumaan väärään tilaan tai ennen yhden tehtävän valmistumista otetaan seuraava jo työn alle. Ruskean värin korkeus kuvaa tehtävälistalla jäljellä olevia työtehtäviä. Vihreä väri kertoo valmistuneista työtehtävistä.



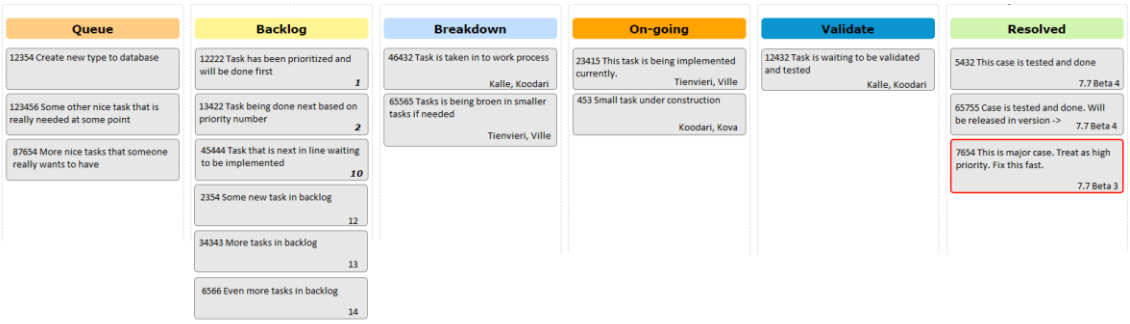
Kuva 8 Kumulatiivinen virtauskaavio projektissa ennen Kanbanin käyttöönottoa. X-akselilla tehtävät ja y-akselilla kulunut aika muodossa kk/pv.

Kuvassa 9 on esitetty projekti Kanbanin käyttöönoton jälkeen. Kanbanin käyttöönoton jälkeen testausta odottavien tehtävien määrä on vähentynyt merkittävästi. Työn alla olevien tehtävien määrä on pysynyt myös hyvin vakiona. Huolestuttavaa tosin on, että valmiiksi tulleiden tehtävien määrä ei myöskään ole noussut pitkään aikaan. Työlista on lähinnä vain kasvanut. Suurin syy sille, miksi tehtäviä ei valmistunut oli se, että projekti oli tällöin hyvin vahvasti uuden laajennuksen suunnitteluvaiheessa. Kehittäjät suunnittelivat uutta näkymää ohjelmistoon ja se käytännössä lopetti kehityksen olemassa olevilta näkymiltä. Laajennukseen tehty työ ei näy tässä kaaviossa, sillä siihen liittyvät tehtävät eivät ole päätyneet Kanban-taululle, vaikka näin syytä olisi ollut syytä. Optimaalisessa tilanteessa työtehtäviä tulisi tasaisesti valmiiksi eikä kuvassa olisi pitkiä vaakasuoria linjoja. Tämä kertoo siitä, että projektissa tehdään liian isoja työtehtäviä tai työn alla olevat työtehtävät eivät edisty lainkaan. Voitaisiin sanoa, että koko ajan ylöspäin nousevat käyrät kertoisivat projektista, jossa tapahtuu mitattavaa edistystä.



Kuva 9 Kumulatiivinen virtauskaavio projektissa Kanbanin käyttöönoton jälkeen. X-akselilla tehtävät ja y-akselilla kulunut aika muodossa kk/pv.

Kuvassa 10 on esitetty projektissa käytössä oleva Kanban-taulu. Tehtävät ovat aluksi queue-tilassa. Tämä tila sisältää tehtäviä, joiden oikeellisuudesta ei ole vielä varmuutta. Nämä tehtävät täytyy hyväksyä, jotta ne siirtyvät varsinaiselle listalle. Queue-lista ei yleensä näy koko kehitystiimille, sillä siellä olevat tehtävät eivät ole oleellisia kehittäjillä. Projektin vetäjä sekä asiakas, jolle ohjelmistoa tehdään, käyvät listaa läpi tietyn väliajoin. Listalta nostetaan uusia tehtäviä backlog-tilaan tai tehtävät suljetaan kokonaan. Backlog on priorisoitu tehtävälista, jossa tärkein tehtävä on päällimmäisenä. Tehtävät ovat numeroitu tärkeysjärjestyksen mukaan. Pieni numero tarkoittaa suurinta prioriteettia. Numero on yleensä suuntaa antava, kehittäjillä on hieman valtaa siihen mitä tehtäviä he valitsevat listalta. Tärkeimmät tehtävät saavat numerot 1-9. Toiseksi tärkeimmät menevät kategoriaan 10 – 99 ja kaikki loput saavat arvot sadasta ylöspäin. Tehtävillä voi olla myös sama prioriteettinumero. Tehtävät on priorisoitu yhdessä asiakkaan ja kehitystiimin kanssa.



Kuva 10 Projektin Kanban-taulu.

Breakdown-tilassa tehtävä määritellään tarkemmin. Tehtävät myös pilkotaan pienempiin osiin, jotta toteutus kestäisi enintään viisi työpäivää. Optimiksi asetettu tehtävän pituus on yhdestä kolmeen päivään. Kanban-listalla näkyy, kuka on ottanut asian hoitaakseen. On-going-tilassa tehtävää toteutetaan ja koodi katselmoidaan. Validate-tila on varattu testaukselle ja testitapauksen tarkastamiselle. Käytännössä tämä tarkoittaa sitä, että toinen kehittäjä ajaa testin läpi ja tarkastaa, että testi on järkevä. Tarvittaessa puutteellinen testi kirjoitetaan uudelleen tai jos toteutuksesta löytyy ongelmia, niin nekin korjataan. Resolved-tila kertoo siitä, että tehtävä on nyt valmis. Tehtävän kohdalla lukee ohjelmistoversio, jossa tehtävä julkaistaan.

6.3 Ohjelmiston laadunvarmistus

Projektin alussa ohjelmiston testaukseen ei oikein kiinnitetty suurtakaan huomiota. Ohjelmistoa kehitettiin eteenpäin ja minkäänlaisia testejä ei tehty. Voitaneen sanoa, että kehitystä vietiin eteenpäin erilaisten prototyyppien avulla. Lopulta saavutettiin piste, kun todettiin, että ohjelmisto on tarpeeksi kypsä, niin sitä on syytä alkaa testata. Testien kirjoittaminen oli tässä vaiheessa kehitystä valtava urakka, joka kuitenkin oli pakko tehdä. Kaikista valmiiksi saaduista toiminallisuuksista kirjoitettiin testit, joilla voitiin tarkastella, että toiminnallisuudet toimivat määritysten mukaan. Testit kirjoitettiin Testlink-työkaluun, jolla voidaan hallinnoida testejä. Testlinkillä on helppo myös muokata olemassa olevaa testiä toimintojen muuttuessa. Ajettavat testit olivat kaikki

manuaalisesti ajettavia. Testejä ajettiin tarpeen mukaan eli aina ennen kuin ohjelmisto annettiin asiakkaalle käytettäväksi. Alkuun kaikki testit ajettiin jokaisella kerralla. Tämä ei alkuun ollut suuri ongelma, sillä testien määrä oli kohtuullinen.

Projektin kuluessa vaatimukseksi nousi, että kaikki tehdyt muutokset tai toiminnallisuudet vaativat testin. Uusi toiminnallisuus vaati käytännössä aina uuden testin kirjoittamista tai olemassa olevan testin päivittämistä. Tämän vaatimuksen johdosta testien määrä kasvoi hurjasti. Jos muutos oli bugikorjaus, viittaus olemassa olevaan testiin saattoi olla riittävä. Tämän takia joistakin testeistä tuli myös liian monimutkaisia koska yhteen testiin lisättiin uusia vaiheita testaamaan jotain uutta toimintoa. Tässä vaiheessa testien suorittamista jouduttiin rajoittamaan, kun ohjelmisto haluttiin antaa eteenpäin loppukäyttäjille, sillä kaikkien testien ajamisessa olisi mennyt kohtuuttoman kauan. Lisäksi kaikkia testejä ei olisi ollut mielekästä testata, sillä ohjelmistoon tehdyt muutokset eivät välttämättä kohdistuneet kuin tiettyihin testeihin. Testeistä alettiin ajamaan nyt aina pieni vakiokokoelma, jolla varmistuttiin siitä, että ohjelmiston kriittiset perustoiminnot toimivat. Lisäksi ajettiin kaikki testit, joita oli muokattu tai testiin liittyvää toiminnallisuutta oli muokattu.

Projektin testauksesta vastasivat pääosin samat henkilöt, jotka ovat ohjelmistoa kehittäneet. Testejä pyrittiin jakamaan siten, että testin kirjoittaja ei suorita testiä, jonka oli kirjoittanut. Tämä oli varsinkin projektin loppupuolella ollut hieman hankalaa koska kehittäjiä oli vain muutama. Lisäksi kehittäjät muokkasivat testejä hyvin vapaasti, joka saattoi johtaa siihen, että yhtä testiä oli muokannut kaikki kehittäjät.

Testlink-ohjelman päivittyminen projektin kuluessa aiheutti myös ongelmia. Olemassa olevaa testiä ei voinut projektin alkuvaiheessa lainkaan muokata. Testistä piti tehdä uusi versio. Tällöin vanha versio jäi muistiin ja myös kaikki ajetut testit jäivät talteen. Päivityksen yhteydessä tullut muutos mahdollisti olemassa olevien testien muokkaamisen ilman uuden version tekemistä. Tämä aiheutti ongelmia, jos testi oli ajettu ja sitä oli muokattu. Tällöin ei ollut enää mahdollista nähdä muokatun version vanhempaa sisältöä.

Projektin testaus ja laadunvarmistus kehittyi koko ajan parempaan suuntaan. Tästä hyvänä esimerkkinä voidaan pitää jatkuvan integroinnin järjestelmän käyttöönottoa

(Continous integration system). Järjestelmä tarkkailee versionhallintaympäristöä ja kääntää lähdekoodin jokaisesta muutoksesta. Tällä tekniikalla löydetään hyvin nopeasti kohdat lähdekoodista, jotka antavat kääntäjästä virheitä. Suurin hyöty on tilanteissa, kun lähdekoodissa on konflikteja ja kehittäjä on vahingossa laittanut v Lisäksi järjestelmä rakentaa asennuspaketin kyseisestä muutoksesta. Asennuspakettia ei tehdä välttämättä jokaisesta muutoksesta, sillä asennuspaketin tekemisessä kuluu noin 30 minuuttia. Muutokset, jotka tulevat asennuspaketin tekemisen aikana menevät jonoon ja kun on aika tehdä seuraava asennuspaketti, otetaan siihen mukaan kaikki uudet muutokset.

Projektissa oli käytössä koodikatselmointi. Käytettävä työkalu oli Gerrit. Katselmointi oli osa kehitysprosessia. Jokainen tehty muutos täytyi katselmoida ja hyväksyä jonkun muun tai muiden kehittäjien toimesta. Kehittäjillä oli mahdollista kommentoida koodia ja antaa parannusehdotuksia. Alkuperäinen kehittäjä voi myös vastata kommentteihin ja perustella omaa kantaansa, jos koki olevansa oikeassa. Koodi hyväksytään antamalla sille arvosana -2 ja +2 väliltä. +2 tarkoittaa suoraa hyväksyntää eli koodi on kunnossa. +1 tarkoittaa sitä, että kehittäjän mielestä koodi on hyvä mutta hän haluaisi jonkun muun vielä varmistavan tämän. Usea +1 ei tarkoita, että tila muuttuisi +2 tilaan vaan jonkun täytyy antaa +2 hyväksyntä manuaalisesti. Nolla-tilanteessa käyttäjällä ei ole halua ottaa kantaa koodin laatuun mutta hän saattaa kysyä jotain. Negatiiviset -1 ja -2 tarkoittavat sitä, että koodissa on korjattavaa. -1 Tilanteen kehittäjä korjaa yleensä itse. -2 on hyvin harvoin käytetty ja se tarkoittaa, että tämä koodi ei missään tilanteessa saa päästä versiohallintaan. Tämän arvion antaja joutuu todennäköisesti itse tekemään jotain muutoksia koodiin. Kun koodin muutos oli onnistuneesti hyväksytty Gerrit-järjestelmässä, niin silloin koodi siirtyi automaattisesti versionhallintaan.

Projektin testaus ja laadunvarmennus saatiin hyvälle tasolle. Muutamia parannusehdotuksia olisi automaattiset käyttöliittymän testaukset. Tällä tavalla testejä voitaisiin ajaa huomattavasti useammin kuin manuaalisia testejä. Ongelmana voidaan nähdä ylläpidettävyys ja hinta. Ison järjestelmän testien kirjoittaminen kestää kauan ja lisäksi monet kaupalliset testiympäristöt ovat kohtuuttoman kalliita. Testien ylläpidettävyys on hintaakin suurempi ongelma, sillä osa käyttöliittymän testiympäristöistä ei ymmärrä edes pientä muutosta käyttöliittymään. Tämä vaatii sitten

testien päivitystä. Jotkut järjestelmät ovat hieman kehittyneempiä eikä pienet muutokset aiheuta ongelmia jo olemassa olevaan testiin.

Huomattavasti halvemmalla päästäisiin tekemällä yksikkötestausta. Yksikkötestejä alettiinkin kirjoittamaan ohjelmiston uusin toiminallisuuksiin. Tällöin testattavat osiot voitiin suunnitella testaus mielessä pitäen. Olemassa olevan lähdekoodin yksikkötestaus tulisi varmasti olemaan melkoisen työläs ja aikaa vievä operaatio. Tätä lähdekoodia ei ole alun perin suunniteltu yksikkötestattavaksi, joten kaikkien osien järkevä testaaminen ei varmasti olisi helppoa. Käytännössä lähdekoodia pitäisi kirjoittaa uusiksi melko isoilta osilta, että yksikkötestien kirjoittaminen olisi mahdollista. Yksikkötestien hyvänä puolena voitaneen pitää sitä, että niillä saadaan kiinni ongelmia, joita kehittäjä ei voinut kuvitella muutoksen aiheuttavan.

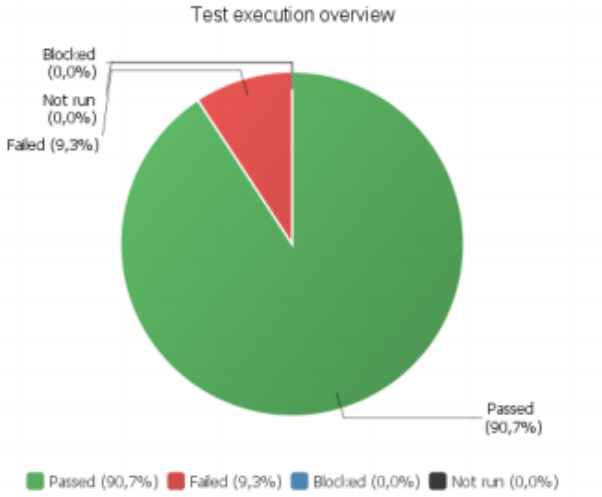
Ohjelmistoa siis testattiin vähintään kahden henkilön toimesta ennen kuin uusi vaatimus hyväksyttiin valmiiksi. Tämän lisäksi ennen uuden ohjelmistoversion julkaisua ajettiin ohjelmalle hieman kattavammat testit. Näillä testeillä oli tarkoitus saada kiinni mahdolliset ongelmat, joita joku muu korjaus tai ominaisuus on aiheuttanut. Tällaisia virheitä löytyy ajoittain. Ajettavilla testeillä on neljä mahdollista tilaa, jotka ovat: ei ajettu, läpäisty, estetty ja epäonnistunut. Kaikki testit alkavat tilasta ei ajettu. Testaajat vaihtoivat testin tilaa aina tarpeen mukaan.

Läpäisty tilanne saavutetaan, kun kaikki testin vaatimat asiat onnistuvat. Kun ongelmia tulee eteen, muutetaan testit tilaksi epäonnistunut. Tämä tilanne vaatii aina selkeän syyn. Se täytyy raportoida testin kommenttikenttään. Lisäksi testaajan täytyy luoda uusi Mantis bug tracker -raportti. Kyseiseen työkaluun kirjataan kaikki uudet vaatimukset ja virheet. Estetty-tilanne on melko harvinainen. Siinä testitapaus on yleensä hyvin puutteellinen tai jokin poikkeava tilanne estää testin ajon. Esimerkkinä voisi olla tarvittavan laitteiston saatavuus tai ohjelma ei edes käynnisty. Myös mikä tahansa muu tilanne, joka estää testin suorittamisen testitapauksessa kuvatulla tavalla. Estetyssä tapauksessa on samat vaatimukset kuin epäonnistuneessa testitapauksessa eli luodaan uusi bugiraportti ja kommenttikentän kautta linkitetään testi siihen. Kuvassa 11 on esitetty epäonnistunut testitapaus testiraportissa.

View dependencies usage in spesific view		Failed
Test plan:	Project 6.7 RC1	
TestLink URL:	Link	
Test description:	This case validates that when view dependency is changed the data in view is updated correctly	
Notes:	Mantis ticket: 123456 Column A value is not updating when changing dependency	

Kuva 11 Testin tarkempi kuvaus testiraportissa.

Kun kaikki testit on ajettu, laaditaan niistä testiraportti. Siihen tulee yhteenveto testituloksista. Kaikkien tulosten jakauma esitetään ensin piirakkakaaviomuodossa ja sen jälkeen ne listataan yksitellen taulukossa. Testin yhteydessä on käyttäjän kirjoittama viesti testin lopputuloksesta. Testin onnistuessa viestiä ei yleensä ole mutta estetyssä- ja epäonnistuneessa tilanteessa ongelmasta on lyhyt kuvaus, sekä suora linkki Mantis bug tracker-ohjelmistoon, josta löytyy tarkemmat tiedot ongelmasta. Kuvassa 12 on esitetty pieni ote testiraportista.



Details	Name	Automatic | Manual	Passed | Failed | Blocked | Not run	Test Execution %	Passed %
	Project 6.7 RC1	0 / 54	49 / 5 / 0 / 0	100	90

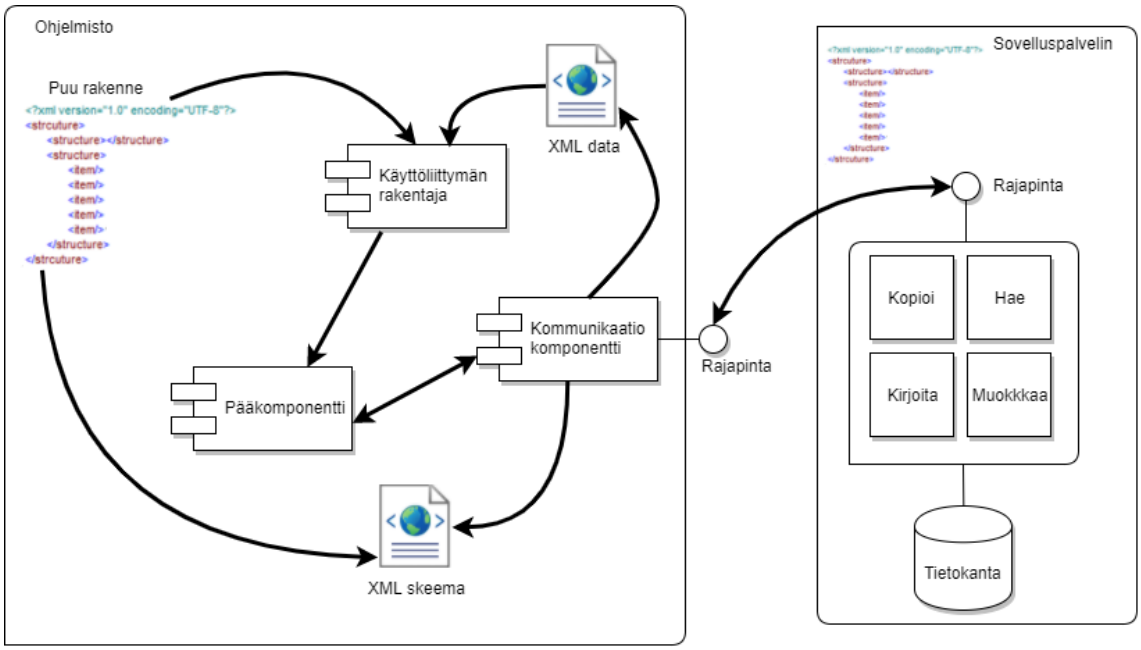
Test Name	Status
Import from database	Passed
Codes import dialog functionality	Passed
Add code	Passed
Viewer functionality	Passed
Attributes created when enabling code	Passed
Attributes import ignoring unsupported attributes.	Passed
Find next available item	Failed
Modbus/TCP wizard Import configuration data	Passed
Modbus/TCP omitted addresses	Passed

Kuva 12 Ote testiraportista.

7 OHJELMISTON TOTEUTUS

7.1 Toteutetun ohjelmiston arkkitehtuuri

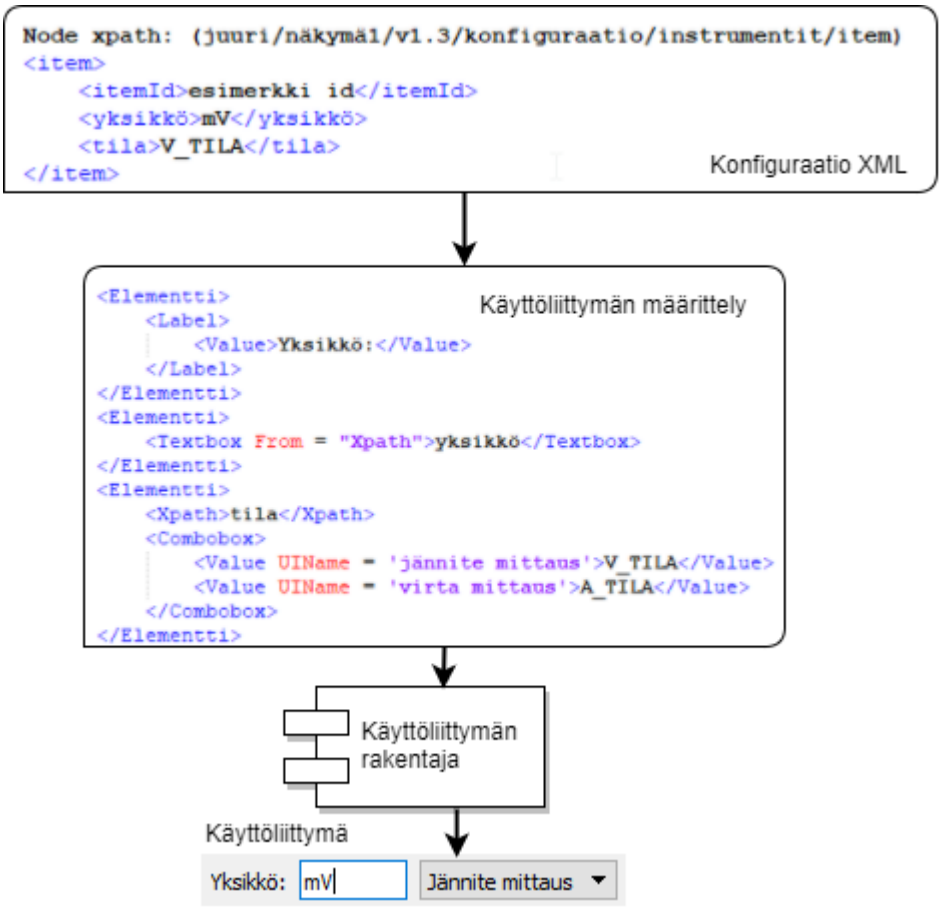
Kuvassa 13 on esitetty toteutetun ohjelmiston korkean tason arkkitehtuuri.



Kuva 13 Tuoteinformaatio-ohjelmiston korkean tason arkkitehtuuri.

Ohjelmisto koostuu palvelinpään toiminnallisuudesta ja käyttäjän koneelle asennettavasta ohjelmasta. Samalla palvelimella on tietokanta- ja sovelluspalvelin. Ohjelmiston sisällä on komponentteja, joilla hoidetaan erilaisia asioita. Pääkomponentti on vastuussa siitä, että käyttäjälle näytettävät tiedot ovat ajan tasalla. Se pitää kirjaa myös käyttäjän tekemistä muutoksista. Kommunikaatiokomponentti hoitaa kaiken tietojenvälityksen ohjelman ja palvelimen välillä. Muut komponentit voivat käyttää kommunikaatio komponentin palveluita tarpeen mukaan. Palvelut ovat tiettyjen tietojen hakemista ohjelmistosta tai tietojen tallentamista ohjelmasta tietokantaan.

Kommunikaatiokomponentin avulla voidaan ohjelmaan hakea XML-muotoinen puurakenne, jossa kaikki ohjelman käyttämät tiedot sijaitsevat. Nämä tiedot on jaettu näkymiin. Ne noudattavat ennalta määrättyä muotoa, joka on määritelty XML-skeemassa. Näkymät halutaan näyttää käyttäjälle tietyllä tavalla. Sitä varten ohjelmassa on komponentteja, jotka hoitavat käyttöliittymän näyttämisen. Yksinkertaisten näkymien käyttöliittymät luodaan käyttäen näkymän rakennuskomponenttia. Se osaa tehdä XML-määrittelystä käyttöliittymän. Käyttöliittymän luonti edellyttää, että konfiguraation tiedot ovat tietyn muotoista. Vapautta sen rakenteeseen tarjoaa näkymäkonfiguraatio-määritteet, jotka mahdollistavat hyvin monipuolisia tapoja kertoa, miltä käyttöliittymä näyttää. Monimutkaisemmat näkymät vaativat erillisen käyttöliittymäkomponentin, joka osaa näyttää tiedot halutulla tavalla. Kuvassa 14 on esitetty, kuinka instrumentin tiedoista tuotetaan käyttöliittymä dynaamisesti käyttäen näkymän määrittelyä.



Kuva 14 Käyttöliittymän luonti dynaamisesti.

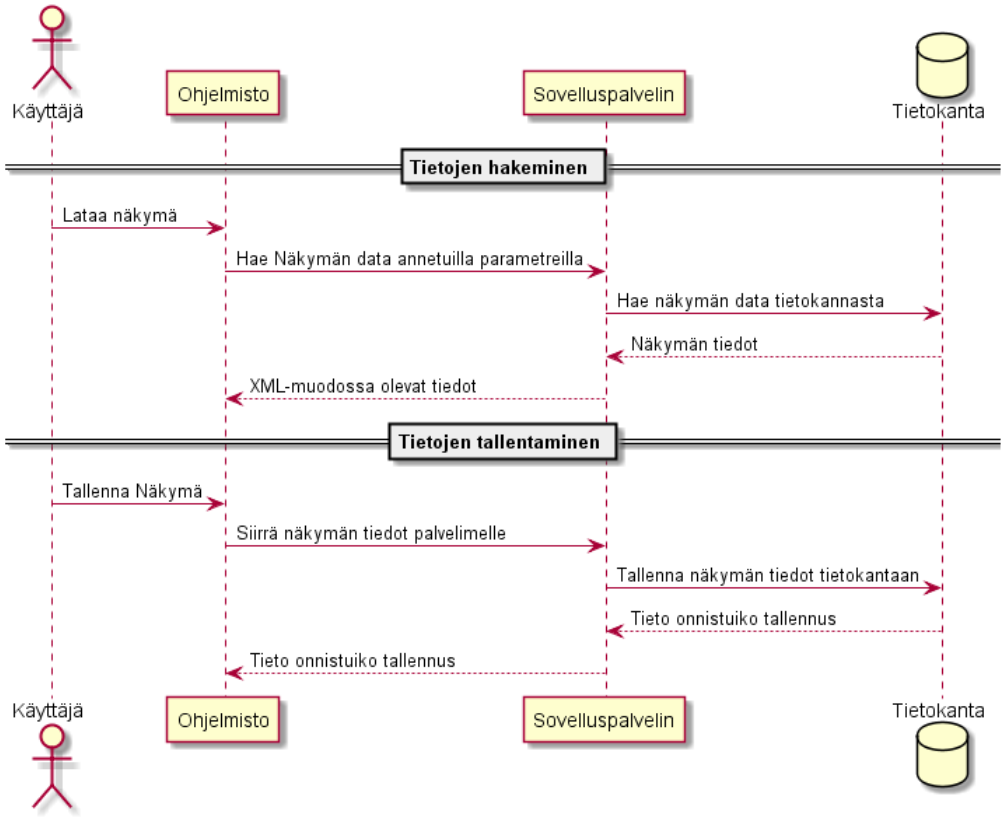
Konfiguraatiossa, joka on XML-muotoinen, on kuvattu sen hetkisen arvon, jonka loppukäyttäjä on tietylle komponentille asettanut. Käyttöliittymän määrittely hoidetaan myös erillisen XML-muodossa olevan tiedoston kautta. Siinä on, määritelty millaisilla komponenteilla arvot halutaan näyttää ja mistä ohjelmisto hakee arvot varsinaisesta konfiguraatiosta. Sijainti perustuu XPath-polkuun. Sijainnin voi ilmoittaa absoluuttisena polkuna tai se voi olla suhteellinen edellisiin käyttöliittymän määrittelyihin. Yleensä kun on tiedossa, että tiettyjä rakenteita on vain yksi niin tälle annetaan absoluuttinen sijainti. Rakenteille, joita voi olla useita samanlaisia tai hieman erilaisia annetaan suhteellinen sijainti. Käyttöliittymän määrittelyssä voi myös rajoittaa arvoja, joita käyttäjä voi syöttää. Esimerkiksi tekstikenttään voitaisiin sallia vain numeeriset arvot tai alasvetovalikkoon määritellään vain tietyt arvot. Käyttöliittymän rakentaja muodostaa näistä kahdesta lähtöarvosta halutun käyttöliittymän.

Tietokannan ja ohjelmiston välissä oleva sovelluspalvelin ottaa vastaan ohjelmistolta tulleet pyynnöt ja vastaa niihin määrätyllä tavalla. Sovelluspalvelimena on käytetty JBoss 6.1 Enterprice Application Platformia.

Palvelimella sijaitsee PostgreSQL-tietokanta. Siitä on käytössä versio 8.4.20. Tietokanta sisältää kaiken ohjelmiston varsinaisen sisällön. Sisällöllä tarkoitetaan tietoa, jota ohjelman käyttäjä tuottaa. Tieto on tallennettu XML-muodossa. Tietokannassa on myös määritelty XML-skeemarakenteita, joissa on määritelty varsinaisten tietojen rakenne. Näitä skeemoja käyttäen ohjelma osaa käyttäjän pyynnöstä luoda uutta tietoa aina oikeassa muodossa. Nämä skeemat ovat versioituja. Tästä on hyötyä, kun halutaan päivittää skeemojen rakennetta. Käytännössä kaikki skeemojen rakenteeseen tehdyt muutokset vaativat uuden version tekoa, sillä vanhempaa versiota voidaan käyttää jossain. Käytetyn skeeman muokkaaminen voisi aiheuttaa erikoisi ristiriitatilanteita käytettäviin tietoihin.

7.2 Kommunikaatio palvelimelle

Kaikki ohjelman ja palvelimen välinen kommunikaatio tapahtuu SOAP-protokollan avulla (W3C 2007). Kommunikaatio tapahtuu asynkronisesti. Ohjelmistossa on määritelty tiedosto, jossa kerrotaan tarvittava osoite, autentikointitiedot ja aikakatkaisun arvot. Näillä tiedoilla ohjelmisto osaa toimittaa kaikki pyynnöt oikealla palvelimelle. Ohjelma lähettää pyynnön, jossa on tunnus, versionumero ja pyynnön tiedot. Palvelin tulkkaa saamansa pyynnön ja käy hakemassa sitä vastaavan tiedon tietokannasta. Lopulta palvelin vastaa pyyntöön palauttaen vastauksen ennalta määritellyssä muodossa. Kaikki kyselyt tapahtuvat XML-muodossa. Kuvassa 15 on esitetty yksinkertaistettu sekvenssikaavio, jossa esitetään kuinka käyttäjän tekemät muutokset siirtyvät SPD-ohjelmistosta tietokantaan ja kuinka ohjelmiston näkymien tiedot saa ladattua ohjelmistoon.



Kuva 15 Tietojen lataaminen ja tallentaminen.

Liitteessä 1 on esitetty kysely kokonaisuudessaan, jolla voidaan pyytää tietyn näkymän tietyn version tiedot. Tässä kyselyssä tarvitaan näkymän tunniste ja versio pakollisina tietoina. Lisätietona voi lisäksi antaa profiilin tunnisteen, joka kertoo minkä profiilin tiedot haetaan. Jos profiilien tunnistetta ei anneta, haetaan kaikilla mahdollisilla profiileilla. Niiden käyttötarkoitus oli alun perin hyvä idea mutta niiden käyttäminen jäi ainoastaan yhteen näkymään ja muiden näkyminen kanssa nämä aiheuttavat lähinnä ongelmia.

Taulukossa 2 on esitetty tärkeimpien kyselyjen tunnukset ja lyhyet kuvaukset siitä mitä kyselyt mahdollistavat. Ohjelmiston dokumentaatiosta löytyy lisää vähemmän käytettyjä kyselyjä.

Taulukko 2. Ohjelmiston ja palvelimen välisten kyselyjen listaus.

Viestin tunnus	Viestin kuvaus
GetViewVersions	Hae listaus kaikista versioista, joita tietystä näkymästä on saatavilla.
GetNextUniqueID	Hae uusi yksilöllinen tunniste. Tunnisteella yksilöidään näkymän sisällä olevia tietoja. Tunniste on yksilöllinen kaikkien käyttäjien kesken.
GetViewComments	Hae tiettyyn näkymän versioon liittyvät kommentit, joita käyttäjät ovat tuottaneet.
GetViewInfo	Hae näkymään liittyvää tietoa. Tieto pitää sisällään muun muassa tiedon siitä mitä versiota XML-skeemasta näkymä käyttää.
GetViewData	Hae näkymän tietyn version tiedot tietokannasta.
CreateRevision	Luo näkymästä uuden version perustuen edellisiin tietoihin. Näkymälle luodaan samalla uusi versionumero.
SaveData	Näkymän tiedot korvataan uusilla käyttäjän antamilla tiedoilla. Tallennettujen tietojen versionumero pysyy samana.

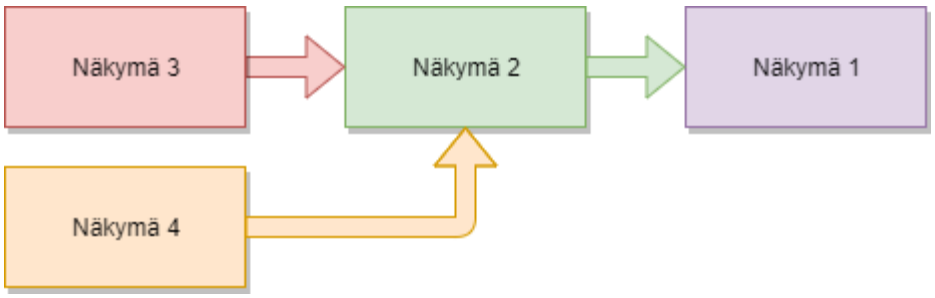
7.3 Näkymien lataaminen ja näyttäminen

Ohjelmiston tarkoituksena on säilyttää käyttäjän tuottamaa sisältöä. Tämä sisältö voidaan ottaa käyttöön muissa ohjelmiston osissa tai jopa täysin muissa ohjelmissa. Koska muut palvelut käyttävät sisältöä, täytyy tämän sisällön eri elementit olla yksilöitävissä. Sisällön yksilöitävyys on hoidettu versioimalla. Ohjelmistossa on erilaisia näkymiä, jotka sisältävät eri tyyppistä tietoa. Näkymät voivat myös olla riippuvaisia toisista näkymistä. Näkymien välille tehtiin viittaus, joka mahdollisti sen, että näkymien tietoja ei tarvinnut monistaa useaan paikkaan käytettäväksi. Kuvassa 16 on esitetty tapa, jolla näkymän sisällössä viitataan toiseen näkymään. Huomioitavaa tässä on se, että yksi näkymä voi viitata useampaan kuin yhteen näkymään.

```
<Versioning ViewVersion = "3.0">
    <Dependency>
        <ViewID>Näkymä1</ViewID>
        <ViewVersion>2.0</ViewVersion>
    </Dependency>
    <Dependency>
        <ViewID>Näkymä2</ViewID>
        <ViewVersion>2.5</ViewVersion>
    </Dependency>
</Versioning>
```

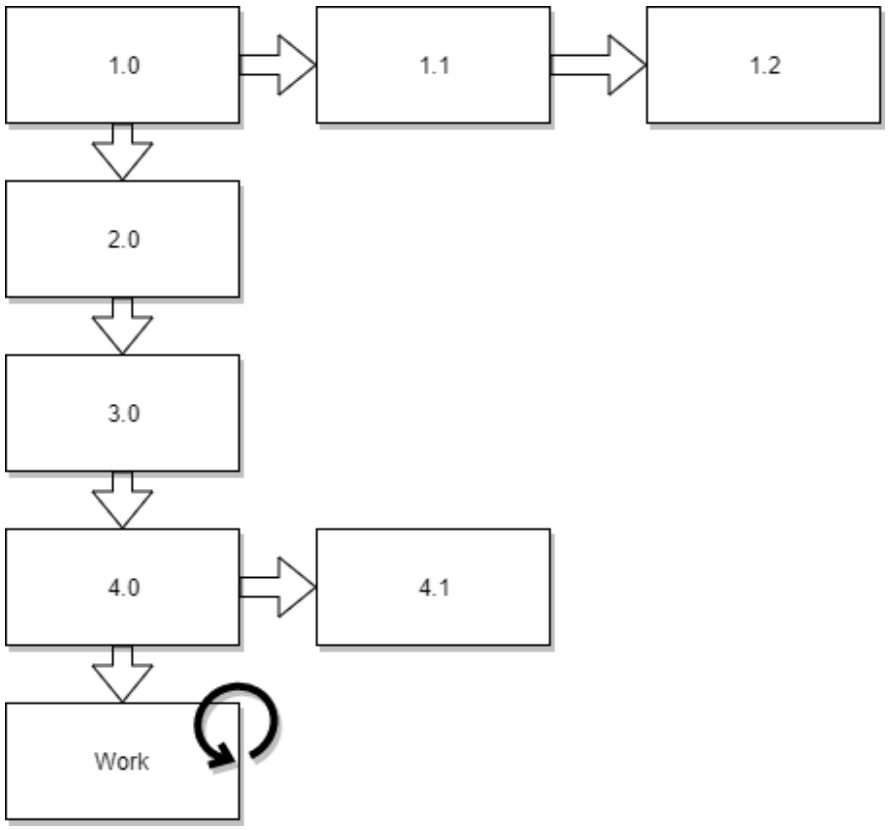
Kuva 16 Näkymän riippuvuuksien määrittely.

On myös mahdollista, että näkymä on riippuvainen kahdesta tai useammasta näkymästä toisen näkymän kautta. Tällä hetkellä näkymillä on aina vain riippuvuus suoraan yhteen näkymään. Mahdollinen toinen riippuvuus tulee sitten ensimmäisen riippuvuuden kautta. Teknisesti olisi mahdollista, että näkymä olisi suoraan riippuvainen kahdesta näkymästä. Tällaista tilannetta ei ohjelmassa vielä ole tullut vastaan. Näkymien mahdolliset riippuvuussuhteet on esitetty kuvassa 17.



Kuva 18 Näkymien riippuvuussuhteita.

Näkymien sisältämät tiedot haetaan aina tietokannasta, johon ne on tallennettu versioituna. Ohjelmistolla niistä voidaan luoda uusia versiota. Kuvassa 18 on esitetty yhden näkymän mahdolliset versiot. Jokaisella näkymällä on aluksi vain työversio, Work. Työversio on aina viimeisin saatavilla oleva versio mutta kun muita versioita ei ole niin se on myös ainut versio.



Kuva 17 Näkymän versiointi.

Tähän työversioon käyttäjät pääasiassa tekevät muutoksiaan. Työversioon tehdyt muutokset kirjoitetaan aina edellisen muutoksen päälle. Tämä mahdollistaa useat muutokset ilman, että pitäisi tehdä uusi versio jokaisen muutoksen jälkeen. Koska työversiota ei ole julkaistu sitä ei voi viedä muihin ohjelmiin, jotka käyttävät SPD-ohjelmistossa tuotettua tietoja, ilman erityisoikeuksia. Nämä oikeudet on saatavilla vain muutamalla ohjelmiston pääkäyttäjällä. Työversion katsotaan olevan valmis vasta kun se on julkaistu. Kun käyttäjät ovat tyytyväisiä, että työversio alkaa olla valmis, siitä voi julkaista uuden standardin.

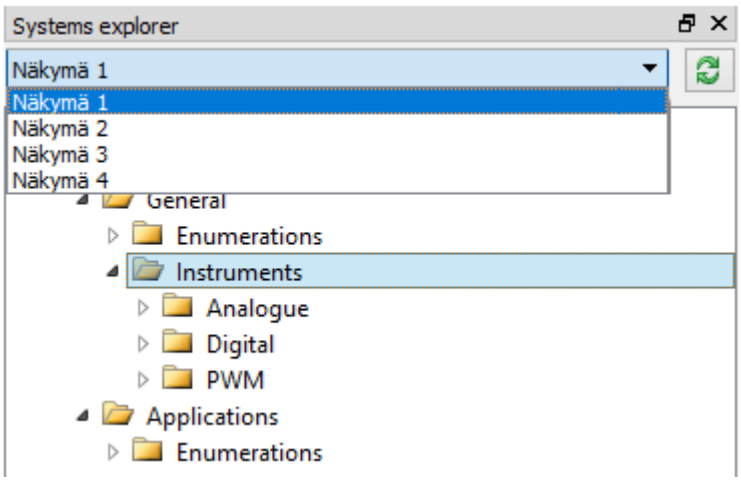
Standardin julkaiseminen tarkoittaa käytännössä kopion ottamista sen hetkisestä työversiosta. Uusi standardi numeroidaan aina muodossa x.0, jossa x on yhtä suurempi luku kuin edellinen standardi. Standardin julkaisun hetkellä juuri tehty standardi ja työversion sisältö ovat identtisiä pois lukien versionumeroa. Työversiota voi tämän jälkeen muokata taas normaalisti. Voidaan ajatella, että työversio on aina viimeisin versio näkymän tiedoista.

Standardeista voi tarvittaessa tehdä myös standardikohtaisia julkaisuja (revision). Muutos kohdistuu aina valittuun standardiin. Standardikohtaisten julkaisujen luonti eroaa työversiosta siten, että vain yhdessä istunnossa tehdyt muutokset ovat mahdollisia. Näiden muutosten jälkeen tehdään aina uusi standardikohtainen julkaisu. Sen teko kasvattaa jälkimmäistä numeroa aina yhdellä. Standardikohtaisen julkaisun pohjaksi otetaan aina edellinen julkaisu ja muutokset lisätään siihen. Standardikohtaisen julkaisun tekeminen on yleensä melko harvinaista, niitä tehdään vain silloin kun olemassa olevaan standardiin on päässyt suurempi ongelma, joka pitää saada korjattua.

Näin jälkikäteen ajateltuna standardikohtaisten julkaisujen tekeminen olisi pitänyt olla vastaavanlaista kuin työversion tekeminen, tällöin olisi ollut mahdollista useamman käyttäjän tehdä muokkauksia ja versionumero kasvaisi vasta sitten kun käyttäjät ovat sitä mieltä, että kaikki muutokset on saatu mukaan.

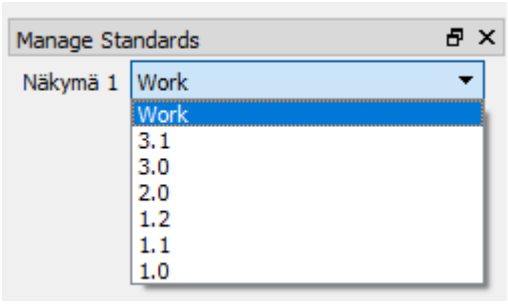
Oleellisena komponenttina näkymässä on Systems explorer, joka koostuu puurakenteesta ja pudotusvalikosta. Puussa on tarjolla kaikki näkymään kuuluvat tietorakenteet. Nämä on esitetty kansioissa. Kansioiden sisällä on niihin kuuluvat tarkemmat tiedot.

Valitsemalla joku puun lapsista avautuu käyttöliittymä, jossa näytetään valitun lapsen tiedot. Näkymien vaihtaminen tapahtuu ohjelmistossa yksinkertaisesti puun yhteydessä olevasta pudotusvalikosta. Tämä on esitetty kuvassa 19. Valitsemalla uusi näkymä vaihdetaan puuhun kyseisen näkymän sisältö.



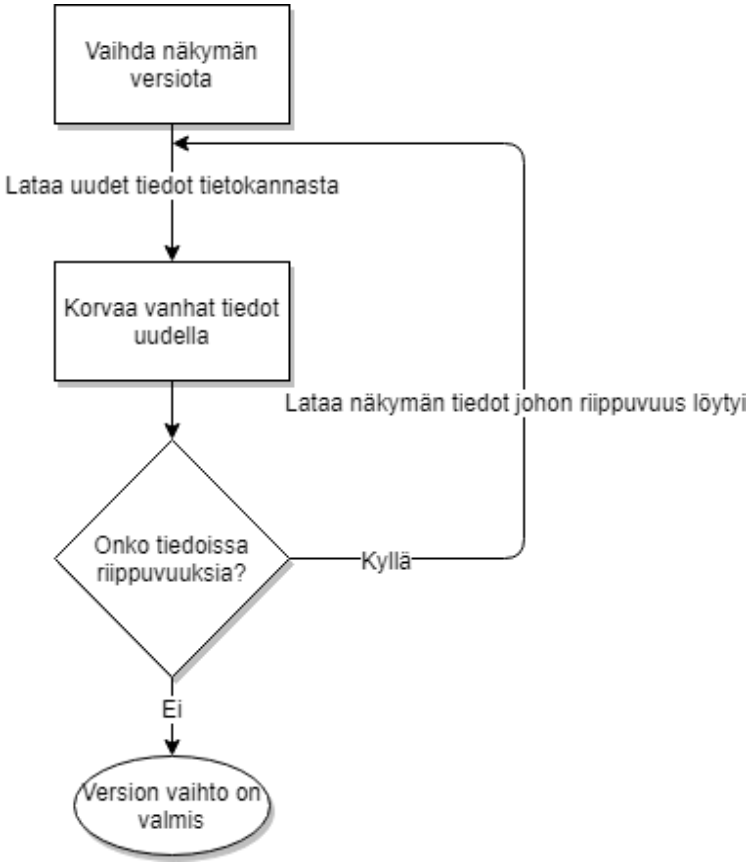
Kuva 19 Näkymän valinta ja näkymän datan listaus.

Oletuksena ladattavan näkymän versio on työversio. Syy tähän on yksinkertaisesti se, että työversiota käyttäjät muokkaavat useimmin. Näkymän versiota voi vaihtaa pudotusvalikosta, joka on sijoitettu Systems explorer-ikkunan alle. Version vaihtaminen korvaa sen hetkiset tiedot uusille. Version vaihtamisen jälkeen puun tiedot ja rakenne päivitetään vastaamaan uutta valittua versiota. Jos käyttäjällä on tallentamattomia muutoksia, nämä muutokset menetetään, jos version vaihtaminen viedään loppuun asti. Muutosten hävittämisestä varoitetaan. Version valinta on esitetty kuvassa 20.



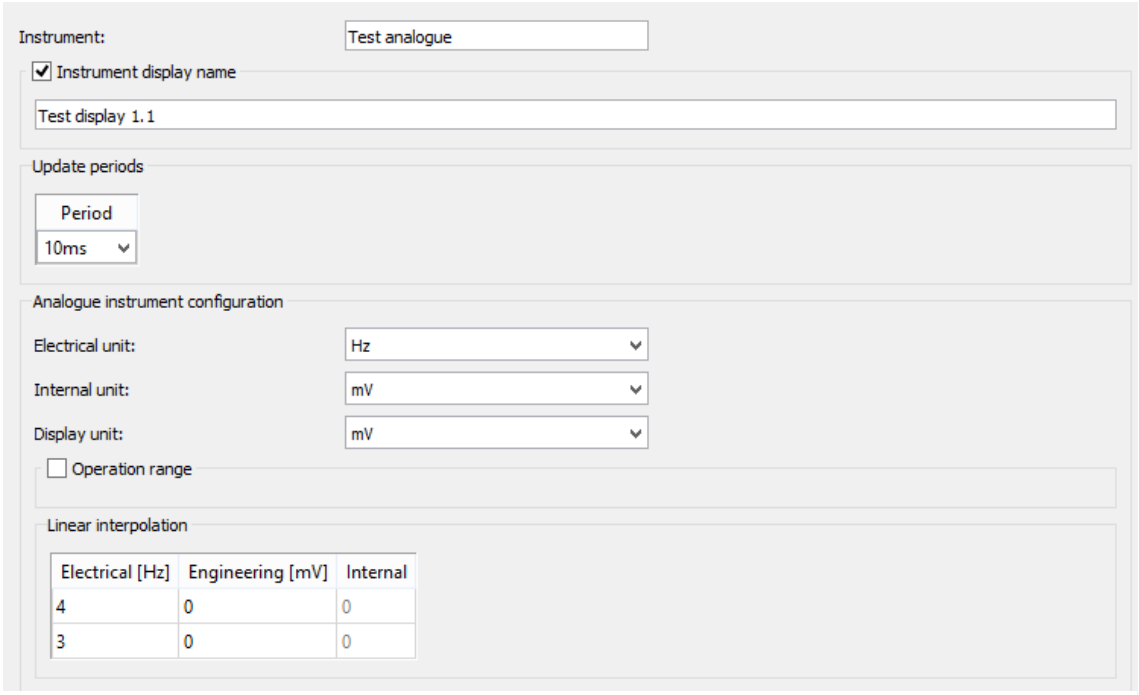
Kuva 20 Version valinta.

Versiot ovat järjestetty nousevassa järjestyksessä ja työversio on aina päällimmäisenä. Näkymän tietojen lataamisen jälkeen tarkistetaan, onko näkymä riippuvainen jostain toisesta näkymästä. Kun näkymällä on jokin riippuvuus, täytyy tämä tietty näkymän versio hakea. Tätä tarkastelua jatketaan niin kauan, kun saavutaan tilanteeseen, jossa uudella näkymällä ei ole enää riippuvuuksia. Tällöin version vaihtaminen on valmis. Kaikkien näkymien haetut tiedot viedään niille varattuun XML-rakenteeseen. Tämän jälkeen valittua näkymän versiota voi käyttäjä alkaa käyttämään. Version vaihtamisen ja riippuvuuksien aiheuttama näkymien versioiden lataaminen on esitetty kuvassa 21.



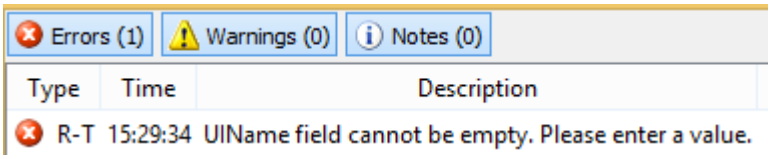
Kuva 21 Näkymän riippuvuuksien lataaminen.

Yksittäinen näkymä aktivoidaan valitsemalla System explorer-puusta lapsikomponentti tai kansio, jolle on määritelty käyttöliittymä. Kuvassa 22 on esitetty yhden mahdollisen instrumentin käyttöliittymä. Käyttöliittymässä on käytetty tekstikenttiä, alasvetovalikkoja, taulukkoa sekä valintaruutuja. Lisäksi käyttöliittymäkomponenttien ryhmittelyssä on käytetty näkymättömiä asettelukomponentteja, jotka asettavat näkyvät komponentit sopivalle etäisyydelle toisistaan.



Kuva 23 Yhden instrumentin käyttöliittymä.

Jokaiseen komponenttiin voidaan määritellä tarvittaessa validointiehtoja. Esimerkiksi tietty kenttä ei saa olla tyhjä tai taulukon ensimmäinen arvo pitää olla pienempi kuin seuraava arvo. Jos ehdot eivät täyty niin ohjelma antaa tästä virheviestin kuten kuvassa 23 on esitetty.



Kuva 22 Virheviesti puuttuvasta arvosta.

8 PARANNUSEHDOTUKSIA

Tässä kappaleessa käydään läpi asioita, joissa on huomattu ongelmia projektin aikana ja kuinka näitä asioita olisi voinut parantaa tekemällä ne toisella tavalla.

Tietokannaksi olisi mahdollisesti voinut harkita esimerkiksi jotain nosql-dokumenttikantaa. Nykyinen PostgreSQL-kanta, jossa on XML-tietoa sisällä, on turhan raskas. Lisäksi PostgreSQL ei mahdollista osittaista XML-tiedon suoraa syöttämistä yhteen tietokannan soluun vaan koko solun sisältö on kirjoitettava sinne kokonaisuudessaan uudestaan. XML on myös hyötykuorman kannalta melkoisen huono tallennusmuoto. Varsinkin suurilla tietomäärillä hyötykuorman osuus on suhteettoman pieni, koska XML-kielen rakenteet aiheuttavat melko paljon ylimääräistä kuormaa ja tekstipohjaisena muotona se vie paljon tilaa. Toisaalta, koska kannan sisältö on muodossa, jota työkalu voi käyttää suoraan, niin vastaanottavan tai lähettävän ohjelmiston ei tarvitse tehdä muunnosta toiseen muotoon. Tämä mahdollistaa nopeamman tietojen siirtämisen tietokannasta ohjelmistoon.

Tietyissä näkymissä olisi ollut hyvä käyttää valmiiden Qt-alustan tarjoamien helppokäyttömallien sijaan alustan paremmin muokattavissa olevia malleja. Puunäkymässä, jossa on yli 10000 riviä, näkymän ja mallin suorituskyvyn heikkous on helposti havaittavissa. Puun rakentaminen kestää noin 30 sekuntia, joka on todella pitkä aika odotuttaa käyttäjää ennen kuin mitään voi tehdä. Yllä olevassa tapauksessa on käytetty QStandardItemModel-komponenttia, joka tarjoaa helpon tavan syöttää puurakenteeseen eri muodossa olevaa tietoa. Kyseistä mallia on käytetty yhdessä QTreeview-luokan kanssa. Sen valinta osui kohdalleen mutta QStandardItemModel olisi ollut syytä rakentaa itse periyttäen se QAbstractItemModel-rajapinnasta. Tällä omalla luokalla oltaisiin päästy eroon kaikesta turhasta käsittelystä. Lisäksi tietojen käsittely olisi ollut merkittävästi verran helpompaa itse tehtyjen rajapintojen avulla. Ylläpidettävyys olisi ollut huomattavasti helpompaa periytetyllä mallilla. Huonona puolena olisi ollut kehityksen hitaus alkuvaiheessa, sillä aivan suoraviivaista uuden mallin ottaminen käyttöön ei ole.

Puun rakennuksen hitaus ei tosin pelkästään johdu käytetyistä malleista ja näkymistä vaan myös tavasta, jolla tietoa malliin syötetään. Tällä hetkellä kaikki tiedot syötetään malliin heti, vaikka tiedon voisi syöttää malliin myös vasta tarvittaessa. Yksi esimerkki tällaisesta toiminnasta on kuvattu Fowlerin (2011) esittämässä lazy loading -patternissa. Toinen mahdollisuus latauksen nopeuttamiseen olisi täyttää puun sisältö erillisissä säikeissä. Puussa on pahimmassa tapauksessa vain kolme tasoa. Suurin osa sisällöstä keskittyy tasoille kaksi ja kolme. Pelkästään ensimmäisen tason puukomponenttien lataamisen kirjoittaminen säikeiden avulla mahdollistaisi huomattavasti nopeamman puun täytön. Mahdollisia rajoitteita tai ongelmia voisi aiheuttaa useat samanaikaiset XML-arvojen kyselyt.

Tähän mennessä asiakas on ollut hyvin tyytyväinen komponentteihin, jotka on saatu valmiiksi. Tyytyväisyys ei tosin ole ollut aina taattu ensimmäisestä versiosta lähtien. Osia toiminnallisuuksista on jouduttu tekemään uudestaan tai jatkokehittämään, jotta asiakas on ollut tyytyväinen. Tähän johtaneita syitä on ollut ollut useita. Näitä ovat olleet mahdolliset väärinkäsitykset toteuttajan ja asiakkaan välillä. Aina kaikki tieto ei ole aina liikkunut muuttumattomana kehittäjälle. Asioita on saatettu käydä läpi palavereissa, jossa toteuttaja ei ole ollut mukana ja sitten kokousmuistioon syötettäessä asia on saattanut muuttaa muotoaan. Asiakas ei ole aina ensimmäisellä kertaa tiennyt mitä on halunnut, vaan mielipide on perustunut hataraan mielikuvaan asiasta. Tällaiset tilanteet ovat korjautuneet siten, että kun asiakas on päässyt itse kokeilemaan toiminnallisuutta, niin tällöin on saatu palautetta, että toiminnallisuuden pitääkin toimia toisella tavalla. Tätä iterointia on joissakin tapauksissa jouduttu tekemään useampaan kertaan.

Koska ohjelmistoa kehitettiin toteutusvaiheesta lähtien ketteriä menetelmiä käyttäen, oli huomattavasti nopeampaa reagoida uusiin toteutuspyyntöihin kuin olisi ollut mahdollista perinteistä vesiputousmallia käyttäen. Kokonaisuutena kehityksen ei voida sanoa olleen ketterää kehitystä sillä asiakas oli sitoutunut pidemmäksi aikaa ohjelman rahoittamiseen. Oikeassa ketterässä kehityksessä rahoituksen jatkamisesta olisi päätetty aina iteraation valmistumisen yhteydessä. Varsinkin ketterän ohjelmistokehityksen julistuksessa (Agile manifesto 2001a) esitetyt kaksi asiaa toteutuivat tämän ohjelmiston kehityksen yhteydessä todella hyvin. ”Kokemuksemme perusteella arvostamme: Vastaamista

muutokseen enemmän kuin pitäytymistä suunnitelmassa” ja ”toimivaa ohjelmistoa enemmän kuin kattavaa dokumentaatiota.” Nämä kaksi asiaa vähensivät huomattavasti turhan työn tekemistä ohjelmiston kehityksen aikana.

Koska ohjelmistossa on edelleen monia näkymiä, joita voitaisiin toteuttaa, on syytä miettiä tärkeimpiä asioita, joita voitaisiin parantaa tulevaisuudessa. Turhien prototyyppiversioiden tekeminen on ensimmäinen asia, jota tulisi välttää. Ensimmäinen toteutettu prototyyppi jää helposti taustalle käyttöön. Prototyyppien tekeminen on ihan hyväksyttävää mutta oman kokemukseni mukaan ne on syytä hylätä kokonaan ja aloittaa varsinainen kehitys puhtaalta pöydältä. Jos ensimmäinen prototyyppi on hyvin puutteellinen, kuten ne usein ovat, tulee uusien toiminnallisuuksien kehittäminen sen päälle hyvin hankalaksi. Vaikka kehittäminen ei olisi hankalaa niin yleensä ylläpidettävyys kuitenkin kärsii. Tämä tuli esille tehdyssä puunäkymässä, joka perustui ensimmäiseen prototyyppiin, johon on jälkikäteen lisätty valtavasti erilaisia ominaisuuksia. Nyt virheiden korjaaminen tai uusien ominaisuuksien lisääminen olisi hyvin aikaa vievä prosessi.

Kun uusia toiminnallisuuksia aletaan toteuttamaan, olisi syytä miettiä kunnolla tapa, jolla kaikki tarvittavat ominaisuudet saadaan tehtyä. Mielellään kannattaa vielä hieman yli-suunnitella ratkaisu. Tällä tarkoitetaan sietä, että ratkaisusta tehdään yleispätevämpi tai monikäyttöisempi kuin aluksi tuntuu järkevältä. Tätä ylisuunnittelua ei kannata tehdä pienimpiin asioihin vaan lähinnä suuriin kokonaisuuksiin. Pienissä asioissa niiden tekeminen liian monimutkaisina kostautuu pidempänä kehitysaikana. Lisäksi kaikkien ratkaisujen ei aina tarvitse olla täysin yleispäteviä. Suuremmissa kokonaisuuksissa hyvä suunnittelu säästää lopulta monelta ongelmalta.

Ohjelmiston ollessa osa vielä isompaa kokonaisuutta olisi ollut syytä ottaa suunnitteluun ja kehitykseen mukaan enemmän mielipiteitä muilta henkilöiltä. Nyt ohjelmisto on hyvin pitkälle yhden tai muutaman ihmisen suunnittelun tulosta. Tämä asia liittyy ehkä enemmän prosessiin, kuinka ohjelmistoa kehitettiin. Tässä prosessissa olisi parannettavaa tulevaisuudessa.

Ohjelmiston kehityksen aikana on usein priorisoitu toisia toiminnallisuuksia toisten yli. Tämä on aivan normaalia kehitystä ja tätä tapahtuu lähes päivittäin. Välillä ongelmaksi muodostuu se, että asiakas pitää monia eri asioita tärkeämpinä kuin ohjelmistoa kehittävä taho. Tällöin ohjelmiston laatu tai ominaisuudet saattavat kärsiä. Esimerkkinä tällaisesta voidaan ottaa vaatimus, joka ohjelmistolla oli melkein alusta asti, liittyen usean käyttäjän mahdollisuuteen tallentaa tietokantaan tietoa samaan aikaan. Saman paikan muokkaaminen ei tarvinnut olla mahdollista samaan aikaan, vain selkeästi eri kokonaisuuksien tallentaminen. Asiakas näki, että tämä toiminnallisuus ei ole lopulta niin tärkeä ja nyt tallentaminen on mahdollista vain kokonaisuus kerrallaan. Tämä on johtanut siihen, että kun useat käyttäjät haluavat muokata samaa tietoa, vain se, joka tallentaa nopeimmin saa omat muutoksensa läpi ilman suurta vaivaa. Kyseinen toiminnallisuus on edelleen tulevien asioiden listalla mutta sen toteutusajankohdasta ei ole mitään tietoa.

9 JOHTOPÄÄTÖKSET

Diplomityössä toteutettiin täysin räätälöity ohjelmisto kohdeyrityksen tarpeisiin. Ohjelmistolla hallinnoidaan suurta tuoteinformaation määrää. Työn alussa tutustuttiin aihealueen teoriaan käytetyiden tekniikoiden ja ketterien kehitysmetodien osalta. Lisäksi ohjelmiston testaamisen teoriaa tutkittiin kirjallisuudesta, koska testaaminen on myös hyvin tärkeä osa-alue ohjelmiston kehittämisessä. Koska ohjelmisto on osa isompaa kokonaisuutta, kuvattiin myös osa pääohjelmiston rakenteesta, jotta saatiin parempi kokonaiskuva kehyksestä, jossa ohjelmistoa kehitettiin. Diplomityön päätavoite oli ohjelmiston kehittäminen ja ohjelman kehitysprosessin kuvaaminen. Täten näitä molempia kuvataan omissa kappaleissaan. Ohjelmistoa kehitettiin usean vuoden ajan ja sinä aikana on tehty paljon hyviä ratkaisuja, mutta on myös huomattu asioita, jotka eivät toimi niin hyvin. Parannusehdotuksia ohjelmiston tiettyihin toteutustapoihin tai ohjelmiston kehitysprosessiin on pohdittu yhden kappaleen verran.

Voidaan sanoa, että diplomityölle asetetut tavoitteet saavutettiin, sillä toteutettu ohjelmisto saatiin valmiiksi työpöytäsovelluksen, palvelinpuolen kuin myös niiden välisen kommunikaation osalta. Ohjelmiston kehitysprosessin kuvauksessa seurattiin projektin vaiheita, kun käytettiin Scrumia ja lopulta siirtymistä Kanbaniin. Syynä ketterän menetelmän vaihtamisella oli Scrumin koettu kankeus verrattuna Kanbaniin. Testaus kuului olennaisena osana kehitysprosessiin ja jokainen toiminnallisuus on testattu aina vähintään kahden henkilön toimesta.

Tätä kirjoittaessa ohjelmiston kolme näkymää on saatu tuotantokäyttöön. Neljäs näkymä on myös valmiina mutta sitä ei ole otettu tuotantoon koska projektille ei ole löydetty uutta omistajaa edellisen omistajan vaihdettua toisiin työtehtäviin. Toivottavasti viimeinenkin näkymä saadaan oikeaan käyttöön, jotta siihen tehty työ ei valu hukkaan. Tuotantoon ehtineistä näkymistä ei ole tullut bugiraportteja pitkään aikaan. Tämä osaltaan kertoo siitä, että ohjelmiston vakaus ja kypsyys ovat hyvällä tasolla.

LÄHDELUETTELO

Agile Alliance (2015a) Agile 101. [online] [12.9.2017] Saatavissa: https://www.agilealliance.org/agile101/

Agile Alliance (2015b). [online] [13.9.2017] Saatavissa: https://www.agilealliance.org/glossary/scrum/

Agile Alliance (2015c). [online] [14.9.2017] Saatavissa: https://www.agilealliance.org/glossary/kanban/

Agile manifesto (2001a) Ketterän ohjelmistokehityksen julistus [online]. [4.5.2016] Saatavissa: http://agilemanifesto.org/iso/fi/manifesto.html

Agile manifesto (2001b) Ketterän ohjelmistokehityksen periaatteet [online]. [12.9.2017] Saatavissa: http://agilemanifesto.org/iso/fi/principles.html

Andersson David J & Andy Carmichael (2016). Essential Kanban Condensed. Seattle, Lean Kanban University Press. ISBN: 978-0-9845214-2-5

Beck Kent (1999) Embracing Change with Extreme Programming. Computer, 32(10) s. 70-77.

Blanchette, J & M. Summerfield (2008). C++ GUI Programming with Qt 4, 2. painos. Massachusetts. Prentice Hall. 13-14 s. ISBN: 978-0-13-235416-5

Edsger W. Dijkstra (1970) Notes on structured programming. Technische Hogeschool Eindhoven, T.H.-Report 70-WSK-03

Fagan M. E. (1999) Design and code inspections to reduce errors in program development. IBM systems journal Vol 38 No 2 & 3 s. 258 - 287.

Forsberg Kevin & Harold Mooz (1995). The Relationship of System Engineering to the Project Cycle. Center For Systems Management, Cupertino.

Fowler Martin (2011) Patterns of enterprise application architecture. Pearson Education, Inc. Boston. ISBN: 0-321-12742-0

Lehtonen Teijo, Seppo Tuomivaara, Ville Rantala, Marja Känsälä, Tuomas Mäkilä, Tero Jokela, Kaisa Könnölä, Matti Kaisti, Samuli Suomi, Minna Isomäki & Marko Ylitolva (2014). Sulautettujen järjestelmien ketterä käsikirja. Turku, Painosalama Oy. ISBN: 978-951-29-5837-5

Macieira Thiago (2009). Count with me: how many smart pointer classes does Qt have? [21.10.2017] Saatavissa: http://blog.qt.io/blog/2009/08/25/count-with-me-how-many-smart-pointer-classes-does-qt-have/

Myers, G.J. (1976) The Art of Software Testing. New York, John Wiley & Sons, Inc. ISBN: 978-1118031964

Naik Kshirasagar & Priyadarshi Tripathy (2008). Software testing and quality assurance John Wiley & Sons, Inc., Hoboken, New Jersey. ISBN: 978-0-471-78911-6

Paakki Jukka (2003) [online]. Ohjelmistotuotanto luentomateriaali. [21.8.2017] Saatavissa: https://www.cs.helsinki.fi/u/paakki/ohtuk03-luento12.pdf ja https://www.cs.helsinki.fi/u/paakki/ohtuk03-luento13.pdf

Perry William E. (1999). Effective methods for software testing, 2. painos. New York. Wiley computer publishing. 6 – 7 s. ISBN: 0-471-35418-X

Power Ken (2014) Metrics for Understanding Flow. XP 2014 Conference. Experience report. [12.9.2017] Saatavissa: https://www.agilealliance.org/wp-content/uploads/2015/12/ExperienceReport.2014.Power_.pdf

PySide documentation (2016) [20.10.2016] Saatavissa: http://wiki.qt.io/PySide

Qt Company (2017). Qt 5.x Documentation [online]. [4.5.2016] Saatavissa: http://doc.qt.io/qt-5/

Qt Company (2016a). 20 Years of Qt Code [online]. [20.9.2016] Saatavissa: https://www.qt.io/qt20/

Qt Company (2016b). Qt Licensing. [20.9.2016] Saatavissa: http://doc.qt.io/qt-5/licensing.html

Qt Company (2016c). Obligations of the LGPL. [20.9.2016] Saatavissa: https://www.qt.io/qt-licensing-terms/

Qt Company (2016d) [online]. Supported Platforms. [20.9.2016] Saatavissa: https://doc.qt.io/qt-5/supported-platforms.html

Schwaber Ken & Jeff Sutherland (2016). The Scrum Guide [online]. [13.9.2017] Saatavissa: http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf

Scrum.org (2017) [online] [14.9.2017]. Saatavissa: https://www.scrum.org/resources/what-scrumbut

Taina Juha (2009) [online]. Ohjelmistoprosessit ja ohjelmistojen laatu. [28.8.2017] Saatavissa: https://www.cs.helsinki.fi/u/taina/opol/k-2009/pdf/luku-6_2.pdf

Toyota Motor Corporation (2017). Just-in-Time -Philosophy of complete elimination of waste. [online]. [18.9.2016] Saatavissa: http://www.toyota-global.com/company/vision_philosophy/toyota_production_system/just-in-time.html

Tuovinen Antti-Pekka (2017). Ohjelmistoprosessit ja ohjelmistojen laatu kurssimateriaali. [online] [12.9.2017] Saatavissa: https://courses.helsinki.fi/sites/default/files/course-material/4482439/luennot_k17_6.pdf

Wells Don (2009). Extreme Programming: A gentle introduction. [online] [19.10.2017] Saatavissa: http://www.extremeprogramming.org

W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). (2008). [online] [5.9.2017] Saatavissa: https://www.w3.org/TR/2008/REC-xml-20081126/

W3C. SOAP 1.2 Specification. (2007). [online] [27.8.2019] Saatavissa: https://www.w3.org/TR/soap12/

W3C. XML Schema. (2004). [online] [4.5.2016] Saatavissa: https://www.w3.org/XML/Schema/

Wikimedia.org (2009) [online]. [13.9.2017] Saatavissa: https://upload.wikimedia.org/wikipedia/commons/5/58/Scrum_process.svg

Williams Laurie (2006) [online]. Testing Overview and Black-box Testing Techniques. [28.8.2017] Saatavissa: http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf

LIITTEET

LIITE 1. Näkymän datan lataaminen

<table>
<tr><td>Viestin tyyppi</td><td>GetViewData</td></tr>
<tr><td>Viestin versio</td><td>1</td></tr>
<tr><td>Kuvaus</td><td>Kysy XML tietorakenne näkymän tietylle versiolle</td></tr>
<tr><td>Pyynti kysely</td><td>
<table>
<tr><th>Kenttä(polku)</th><th>Kuvaus</th><th>Määrä</th></tr>
<tr><td>Request/</td><td>Kyselyn juuri</td><td>1</td></tr>
<tr><td>Request/View</td><td>Näkymän nimi</td><td>1</td></tr>
<tr><td>Request/Version</td><td>Näkymän versio</td><td>1</td></tr>
<tr><td>Request/Profiles</td><td>Lista näkymän profiileista, jotka palautetaan.</td><td>0-1</td></tr>
<tr><td>Request/Profiles/Profile</td><td>Profiilin nimi</td><td>0-*</td></tr>
</table>
<pre>
<Request>
  <View>Näkymä1</View>
  <Version>3.4</Version>
  <Profiles>
    <Profile>General</Profile>
    <Profile>Application</Profile>
  </Profiles>
</Request>
</pre>
</td></tr>
<tr><td>Vastaus kyselyyn</td><td>
<table>
<tr><th>Kenttä(polku)</th><th>Kuvaus</th><th>Määrä</th></tr>
<tr><td>Response</td><td>Kyselyn juuri</td><td>1</td></tr>
<tr><td>Response/Data</td><td>Jokainen vastaus pitää sisällään data noden</td><td>1</td></tr>
<tr><td>Response/Data/*</td><td>Kyselyyn saatava vastaus XML muodossa</td><td></td></tr>
<tr><td>Response/Error</td><td>Virhetiedot jos jotain meni vikaan</td><td>1</td></tr>
</table>
<pre>
<Response>
  <Data>
      <NäkymänData>Hyötydataa</ NäkymänData >
  </Data>
  </Error>
</Response>
</pre>
</td></tr>
<tr><td>Virheet</td><td>Geneeriset virheviestit</td></tr>
</table>