TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ELINA LUKKARINEN

OHJEISTUKSEN MUODOSTAMINEN JA KÄYTTÖÖNOTTO OHJELMISTOPROJEKTISSA

Diplomityö

Tarkastaja: professori Tommi Mikkonen
Tarkastaja ja aihe hyväksytty Tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa 12. elokuuta 2015

TIIVISTELMÄ

ELINA LUKKARINEN: Ohjeistuksen muodostaminen ja käyttöönotto ohjelmisto-projektissa
Tampereen teknillinen yliopisto
Diplomityö, 45 sivua, 2 liitesivua
Joulukuu 2015
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Tommi Mikkonen

Avainsanat: ohjeistus, ohjelmointiohjeistus, laadunvarmistus

Jotkin projektit vaativat, että laatuun kiinnitetään tavallista enemmän huomiota. Syitä tähän voivat olla projektin suuri koko, kiire, tai se, että projekti on luonteeltaan sellainen, että virheet voivat aiheuttaa paljon haittaa.

Ratkaistavana ongelmana työssä on laadun varmistaminen tilanteessa, jossa laatu on uhattuna. Tähän on olemassa monia erilaisia keinoja. Laadunvarmistuskeinojen olemassaolo ei valitettavasti tarkoita, että ne olisivat käytössä. Laadunvarmistuskeinojen hyödyntämisen välineeksi valittiin ohjeistus. Sen avulla voidaan esitellä ja ottaa käyttöön projektille sopivia keinoja.

Työssä esitellään lyhyesti laadunvarmistuskeinoja, joista jotkut liittyvät ohjelmistotuotantoon ja jotkut enemmän ohjelmakoodiin. Lisäksi kuvataan miksi ohjeistus on tarpeen, millainen sen tulisi olla ja miten ihmiset saataisiin noudattamaan sitä. Eräässä ketterässä maksujärjestelmäprojektissa laatu oli uhattuna, ja laadun varmistamiseksi projektille luotiin ohjeistus. Tämän ohjeistuksen pohjana olevat laadunvarmistuskeinot esitellään, samoin kukin ohje perusteluineen. Lisäksi käydään läpi ohjeistuksen käyttöönotto.

Ohjeistuksen käyttöönotossa oli ongelmia, koska siihen ei käytetty tarpeeksi resursseja, ja siksi ohjeiden laatua on vaikea arvioida. Käyttöönotto siis epäonnistui. Tästä kuitenkin opittiin jotain, ja käyttöönottokokemukset ja jatkokehitysajatukset niin käyttöönottoon kuin ohjeisiinkin liittyen on koottu auttamaan muita ohjeistuksen käyttöönottoa suunnittelevia. Työn avulla voi välttää joitakin käyttöönoton ongelmia ja saada valmiuksia laatuohjeistuksen laatimiseen projektille, joka sellaista tarvitsee.

ABSTRACT

ELINA LUKKARINEN: Creation and Deployment of Coding Standards in a Software Project
Tampere University of Technology
Master of Science Thesis, 45 pages, 2 Appendix pages
December 2015
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Tommi Mikkonen

Keywords: programming directions, quality control

Some projects require spending greater than the usual amount of attention on quality. The reasons for this can be tight schedule, the size of a project, or the project being one where errors can cause much harm.

The problem to solve was ensuring quality in a situation where quality is threatened. Quite a few means have been developed for this. Sadly, the existence of quality assurance methods does not mean they are being used. A set of instructions was chosen for introducing and taking into use the suitable methods.

Some means to assure quality are presented in this thesis, connected to either software engineering process or programming. In addition, the use of instructions is motivated, the form of the instructions and the way to get people to use them is considered. In one agile payment software project, the quality was at peril, so a set of instructions was created. The quality assurance methods backing these instructions and all the instructions with their justifications are introduced. It is also explained how the instructions were taken into use.

There were some problems with taking the instructions into use, since not enough resources were used for it. Thus the quality of the instructions is difficult to measure. Taking the instructions to use failed, but something was learned of it. The experiences and thoughts for further development for taking instructions into use and forming instructions are recorded. This thesis can help to avoid some problems with introducing new methods, and give some inspiration on creating quality ensuring instructions for a project that needs them.

ALKUSANAT

Tämä diplomityö on kirjoitettu projektista, josta minä ja kolme muuta opiskelijaa pääsimme tekemään diplomityöt. Työ on toivottavasti myös jatkossa hyödyksi yrityksessä, jossa se tehtiin. Oli hienoa päästä isoon ja mielenkiintoiseen projektiin heti työuran alussa, ja tästä on hyvä jatkaa.

Diplomityö on nyt valmis, ja valmistuminenkin lähellä. Vaikea sanoa, mikä tähän johti. Varmaan sillä oli merkitystä, että kotona sain kysymyksiini perusteellisia vastauksia, oppia hoksaamisen ilon, ja huomata, että kaikkea eivät tiedä vanhemmatkaan. Isoveljeäni olen seurannut pienestä asti, mutta väitän kuitenkin, että saman koulutusalan valitsin itsenäisesti. Tärkeämpää kuin ala oli kyllä päästä TTY:lle. Ensimmäisestä ohjelmointikurssista lähtien olin varma siitä, että valitsin oikean alankin. Alan mielenkiintoisuudesta huolimatta olisi opiskelu ollut puisevaa ilman kampuksen yhteishenkeä ja erinäisiä kerhoja ja niiden kautta saatua vertaistukea. Luulen, että ilman TTY:tä olisin valmistunut jotenkin vähäisempänä kuin nyt.

Olen pakottanut likimain kaikki tuttuni kuuntelemaan ajatuksiani tähän työhön liittyen. Sain kuulla rohkaisua ja hyviä vinkkejä, ja keventää sydäntäni puhumalla. Mitä enemmän olette päätyneet kuuntelemaan, sitä suurempi kiitos ja anteeksipyyntö. Erityiskiitos vielä tarkastajalleni, joka kärsivällisesti vastaili sähköposteihini ja antoi palautetta työn etenemisestä ja suunnitelmista.

Tampereella, 24.11.2015

Elina Lukkarinen

SISÄLLYSLUETTELO

1. JOHDANTO

Tämän diplomityön viitekehys on ohjelmistoyritys, joka toteuttaa maksujärjestelmän, jolla palvelun tarjoaja voi ostojen mukaan laskuttaa asiakkaita. Järjestelmää on jatkossa tarkoitus myydä asiakkaille. Järjestelmä tulee luonnollisesti käsittelemään rahaa. Näin ollen virheet voivat johtaa kustannuksiltaan merkittäviin seurauksiin. Järjestelmän laatu on tärkeää sekä virheiden välttämisessä, että asiakastyytyväisyyden varmistamisessa. Tyytyväiset asiakkaat mahdollistavat sen, että järjestelmälle on kysyntää ja sitä päästään jatkokehittämään.

Projektin aikataulu oli tiukka, sillä se haluttiin saada nopeasti valmiiksi. Tästä syystä projektiin päätettiin ottaa mahdollisimman paljon kehittäjiä töihin mahdollisimman nopealla tahdilla. Jos lopussa olisi jäänyt aikaa, kehittäjät olisivat voineet siirtyä muihin projekteihin. Näin ollen kehittäjiä oli monta, ja sitä mukaa kun uusia rajapintoja saatiin määriteltyä tehtäviksi, otettiin projektiin mukaan uusia kehittäjiä, jos heitä oli saatavilla.

Järjestelmän laatu on tärkeä ominaisuus rahan käsittelyn luotettavuuden kannalta. Vaikka laadukkaan koodin päälle on helpompi kehittää ja virheet ovat aikaa vieviä korjata, kiire ja tuottavuuden vaade voi helposti ajaa tinkimään laatuvaatimuksista. Monen eri kehittäjän projektissa koodin kirjoittamiseen voi olla monta tyyliä, mikä laskee koodin luettavuutta ja laatua. Suuri osa projektin jäsenistä ei ennestään tunne toisiaan. Projektissa on käytössä Scrum-menetelmä, joka on tuttu kaikille kehittäjille ja auttaa siten kommunikoinnissa. Kehittäjäyrityksellä ei ole olemassa omaa tyyliopasta. Laadun varmistamiseksi tarvitaan tässä kriittisessä tilanteessa jotakin lisää.

Tässä diplomityössä on tartuttu laadun varmistamisen ongelmaan tutkimalla erilaisia laadunvarmistamiskäytäntöjä ja ehdottamalla ratkaisuksi ohjeistusta, joka muodostetaan projektin tarpeisiin. Tätä ohjeistusta noudattamalla voidaan turvata projektin laatu joustamatta tiukasta deadlinesta. Työssä esitellään laadunvarmistuskeinojen lisäksi ohjeistuksen muodostamista ja maksujärjestelmäprojektiin laadittu ohjeistus. Ohjeistuksen noudattamisen motivointiin otetaan myös kantaa, sillä työn tuottavuuden vaatimus voi johtaa hyvienkin ohjeiden noudattamatta jättämiseen.

Tämän diplomityön luvussa 2 esitellään laadunvarmistamiskäytäntöjä, joita on harkittu ohjeistukseen mukaan otettaviksi. Käytännöistä esitellään erityisesti ohjelmistotuotantoon liittyvinä scrum, pariohjelmointi, jatkuva integrointi ja test driven development. Enemmän ohjelmakoodiin liittyviä esiteltäviä käytäntöjä ovat testaus, kommentointi, katselmoinnit ja refaktorointi. Luvussa 3 otetaan kantaa ohjeistukseen ohjelmistoprojektissa. Käsiteltävät asiat ovat syyt ohjeistuksen tarpeelle, ohjeistuksen rakenne ja sisältö sekä

motivointi ohjeiden noudattamiseen. Luku 4 esittelee projektin, johon ohjeistus tehdään. Siinä kuvataan projektiryhmä, järjestelmä ja projektin liiketoimintakriittiset osiot sekä muut vaatimukset kuten aikataulu. Luvussa 5 käsitellään laadun varmistamista projektissa eli esitellään muodostettu ohjeistus ja suunniteltu käyttöönottomenetelmä perusteluineen. Luvussa 6 arvioidaan miten ohjeistuksen käyttöönotto onnistui ja miten se vaikutti ohjelman laatuun, eli mitkä olivat työn tulokset. Luvussa 7 esitetään yhteenveto.

2. LAADUNVARMISTAMISKÄYTÄNTÖJÄ

Monissa ohjelmointiprojekteissa on käytössä erilaisia ohjenuoria ohjelmointiin liittyen. Näillä ohjeilla pyritään varmistamaan muun muassa, että ohjelmakoodi on luettavaa, laajennettavaa, testattavaa ja toimivaa eli yhdellä sanalla laadukasta. Ja projektissa, johon tämä työ on tehty, erityisesti rahaa käsittelevän ohjelmakoodin laatu haluttiin varmistaa. Laadulle on monia määritelmiä. Esimerkiksi ISO-standardeissa laatu määritellään siten, että laadukas tuote täyttää sille asetetut toiminnalliset ja ei-toiminnalliset vaatimukset [1, ss. 168-172]. Tässä työssä laatu tarkoittaa sitä, että ohjelma täyttää vaatimuksensa ja valmistuu ajallaan eikä sen kehittämisestä tule hidasta virheiden määrän tai epäselvän ohjelmointityylin takia.

Ohjelmakoodin laadun parantamiseksi laaditut ohjeet voivat olla koodin ulkonäköön liittyviä kuten se, mihin kohtaan laitetaan koodilohkon aloittava kaarisulje, tai hieman abstraktimpia mutta edelleen koodiin liittyviä kuten foreach-silmukoiden käyttö for-silmukoiden sijaan. Lisäksi on ohjeita, jotka eivät liity suoraan ohjelmakoodiin vaan ohjelmointiprosessiin. Monet tällaiset ohjeet liittyvät paitsi koodin laatuun myös ohjelmoijan työn tuottavuuteen. Mutta koska virheet ovat sitä kalliimpia mitä myöhemmin ne havaitaan [2], tuottavuuteen ja koodin laatuun liittyvät ohjenuorat eivät yleensä ole keskenään ristiriidassa, sillä laaduton koodi ei ole tuottavaa koodia.

Tässä luvussa esitellään erilaisia laadunvarmistamiskäytäntöjä sekä ohjelmistotuotantoprosessiin että tarkemmin ohjelmointiin ja koodiin liittyen. Monet esitellyt käytännöt ovat liitettävissä johonkin ketterään toimintamalliin, joita esimerkiksi Agile Alliance listaa [3]. Suuri osa käytännöistä on tosin ollut olemassa jossain muodossa jo ennen ketterän kehityksen aatetta. Luvussa esitellään laatuun liittyviä ketterän kehityksen käytäntöjä ja lisäksi katselmoinnit, kommentointi ja muistilistat laadun varmistamisen apuna. Ketterän kehityksen käytäntöjä on valikoitunut esiteltäviksi, koska projekti toteutettiin ketterästi, Scrum-menetelmää mukaillen. Myös Scrum esitelläänkin tässä luvussa, sillä vaikka se ei suoranaisesti keskity laadun varmistamiseen, sen käyttö kuitenkin vaikuttaa laatuun.

Vaikka erikseen dokumentoitu ohjeistus ei olekaan erityisen ketterää, on ketterässä kehityksessä kuitenkin olemassa laatuun liittyviä hyviä käytäntöjä, jotka sopivat sitä kautta hyvin projektiin. Nämä käytännöt eivät myöskään ole itsestään selviä tai sellaisia, joista kaikilla olisi yhtenäinen käsitys, joten niiden esittelyä ja määrittelyä ohjeiksi on syytä harkita.

2.1 Ohjelmistotuotantoon liittyvät käytännöt

Tässä kohdassa esitellään ohjelmiston tuottamisen menetelmiä ja käytäntöjä. Pääosin pysytään siis erossa suoraan lähdekoodiin liittyvistä asioista.

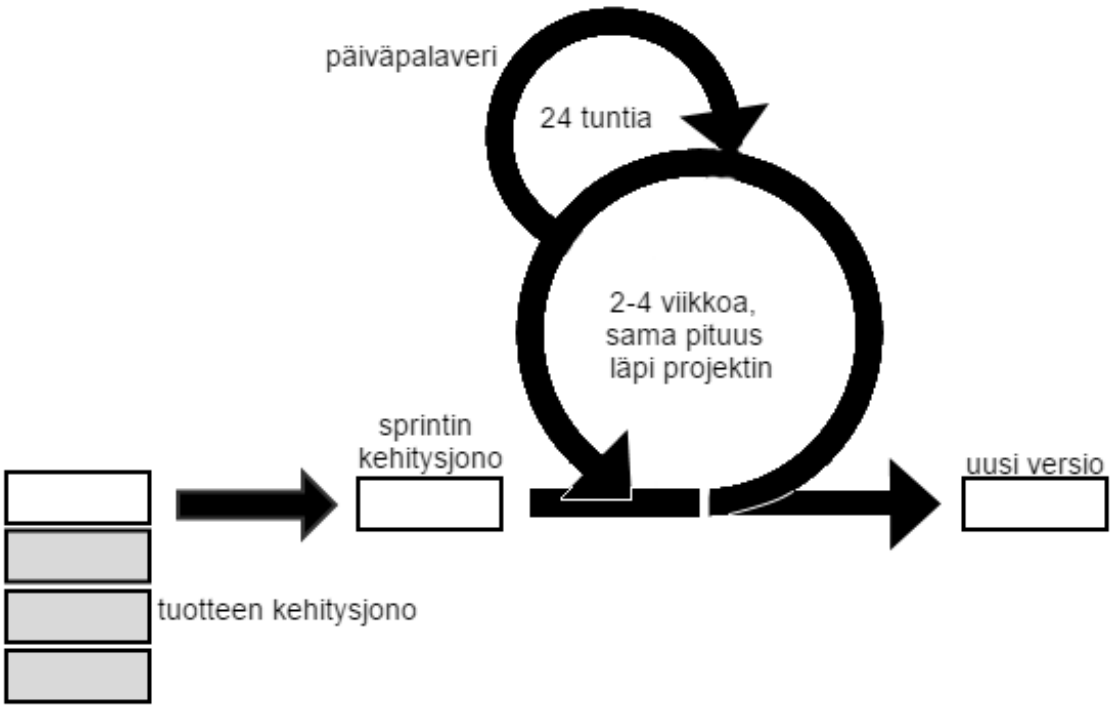
Ensimmäisenä esitellään scrum, joka ei varsinaisesti ole laadunvarmistuskäytäntö, mutta projektia määrittävänä menettelytapana se kuitenkin väistämättä vaikuttaa laatuun. Tämän jälkeen esitellään pariohjelmointi, ohjelmointimenetelmä, jonka on joissain tutkimuksissa todettu parantavan ohjelmakoodin laatua. Sitten esitellään jatkuva integrointi, jonka toteuttaminen vaatii tekniikan käyttöä. Viimeisenä esitellään test driven development (TDD), joka on ohjelmointitapa, jossa testien kirjoittamisella on suuri merkitys. Scrum määrittää koko projektia ja on tyypillisesti alusta asti käytössä, pariohjelmointi liittyy toteutusvaiheisiin, integrointi on ajankohtaista aina kun on saatu jotain toteutettua ja TDD on käytössä läpi projektin ohjelmointivaiheen.

2.1.1 Scrum

Scrum on ketterän kehityksen menetelmä, joka perustuu ihmisten väliseen kommunikointiin ja työn rytmittämiseen. Scrumissa ajatellaan, että asiakkaan toiveiden muuttuminen ja jonkinasteinen väärinymmärtäminen on väistämätöntä. Kuva 2.1 esittelee scrumissa käytetyn työn rytmityksen yksikön, sprintin, sekä sen, miten tehtävät etenevät prosessissa. Koska asiakkaan välitön ja täysi ymmärtäminen oletetaan lähes mahdottomaksi, kuvassa näkyvää tuotteen kehitysjonoa, joka on lista kaikesta, mitä ohjelmassa voidaan tarvita, ei yritetäkään saada heti valmiiksi. Aluksi siinä on vain alusta asti tunnetut ja parhaiten ymmärretyt vaatimukset. Kun ohjelmoidessa huomataan mahdollisia puutteita, pyydetään asiakkaalta lisätietoja, ja toisaalta ohjelmasta tehdään versioita, joita näytetään asiakkaalle, ja näiden avulla saadaan lisätietoa asiakkaan toiveista. [4]

Scrum määrittelee joitakin rooleja. Tuoteomistaja on henkilö, joka maksimoi tuotteen ja tehdyn työn arvon. Hän on vastuussa tuotteen kehitysjonosta – sen tehtävien laadusta ja sen järjestämisestä. Tämän työn hän voi myös delegoida. Scrummaster on scrum-menetelmän käytöstä vastuussa oleva henkilö. Hän auttaa menetelmän noudattamisessa niin, että siitä saadaan mahdollisimman suuri hyöty. Kehitystiimiin lasketaan tuotetta tekevät henkilöt, jotka ovat vastuussa siitä, miten tuotteen kehitysjonon kohdat toteutetaan. Sen koko tulisi olla kolmen ja yhdeksän hengen väliltä. Siinä ei ole erillisiä rooleja esimerkiksi testaajille. [4]

Työn rytmitys perustuu scrumissa sprintteihin. Sprintti on korkeintaan kuukauden mittainen ajanjakso, jonka aikana luodaan uusi versio tuotteesta. Sprinttien kesto pysyy samana läpi ohjelman kehityksen. Uusi sprintti alkaa aina heti edellisen sprintin päättymisen jälkeen. Sprinttiin liittyvät sen suunnittelu, päivittäiset palaverit, kehitystyö, sprintin katselmointi ja retrospektiivi. Sprintin alkaessa sille määritellään tavoite, josta ei luisteta, eikä laadusta tingitä. Laajuutta voidaan selventää ja siitä voidaan neuvotella tuoteomistajan kanssa. Kukin sprintti on tavallaan itsenäinen projekti. Tätä kautta ohjelmaa kehitetään iteroiden paremmaksi, aina edellisen sprintin lopputuotteesta saadun palautteen perusteella. [4]



Kuva 2.1: Scrum – sprintin rakenne.

Sprintin suunnittelussa sovitaan ne tehtävät, jotka tehdään sprintin aikana. Näistä tehtävistä muodostuu yhteinen tavoite, jonka saavuttamisen jälkeen ohjelmistosta on valmiina uusi tuoteversio (Increment). Tehtävät tulevat tuotteen kehitysjonosta, johon niitä luodaan sitä mukaa, kun asiakkaalta tulee uusia toiveita tai vanhat selkiytyvät. Näitä valittuja tehtäviä ja sovittua tavoitetta kutsutaan nimellä sprintin kehitysjono. [4]

Päiväpalaveri on viidentoista minuutin tapaaminen, jossa kehitystiimi tapaa, keskustelee siitä, mitä kukin on tehnyt, onko ongelmia, ja mitä kukin aikoo tehdä ennen seuraavaa päiväpalaveria. Päiväpalaverin jälkeen ihmiset jäävät esimerkiksi pyytämään neuvoja, joita he tarvitsevat suoriutuakseen päivän tavoitteesta. [4]

Sprintin päätteeksi on sprintin katselmointi, jossa käydään sidosryhmien kanssa läpi, mitä edellisessä sprintissä tehtiin, ja keskustellaan mahdollisesti seuraavan sprintin suunnittelua varten tuotteen kehitysjonon tehtävien priorisoinnista. Tämän jälkeen on sprintin retrospektiivi, jossa taas keskitytään siihen, miten edellisessä sprintissä asiat tehtiin ja miten työskentelytapoja voisi parantaa. [4] Kuva 2.1 esitteli scrumin sprintin rakenteen.

Scrumissa on myös tärkeää, että kaikki kehitystiimiläiset ymmärtävät, mitä tarkoittaa, kun jokin tehtävä tuotteen kehitysjonosta tai uusi versio ohjelmasta on valmis. Valmiin määritelmä voidaan sopia yhteisesti. Ohjelman versioiden suhteen on kuitenkin vaatimus, että uusi versio toimii yhdessä vanhojen kanssa ja että se on testattu sekä dokumentoitu. [4]

2.1.2 Pariohjelmointi

Pariohjelmoinnissa kaksi ohjelmoijaa toimii parina. Parista toinen kirjoittaa ohjelmakoodia samalla selittäen, mitä tekee. Parin tarkkaileva osapuoli pitää mielessä suuret linjat ja kommentoi koodin ratkaisuja. Parilla on käytössään vain yksi tietokone. Rooleja vaihdetaan säännöllisesti. Pariohjelmoinnissa ohjelmoija puhuu koko ajan ääneen eli selittää ratkaisuaan, joten hän joutuu tarkemmin ajattelemaan, mitä on tekemässä, ja tämä voi selventää koodin rakennetta. Samoin erilaisiin vaikeampiin ratkaisuihin on aina saatavilla kaksi mielipidettä hieman eri näkökulmista, joten koodin voi toivoa olevan laadukkaampaa. Kaksi ohjelmoijaa huomaa myös helpommin, jos ohjelmalogiikassa on virhe. [5]

Pariohjelmoinnin voi liittää muihin ketteriin käytäntöihin. Eräs pariohjelmointivariantti on ping-pong-ohjelmointi, joka tukee ohjelmointityyliä, jossa testit kirjoitetaan ennen koodia. Ping-pong-ohjelmoinnissa ensin kirjoitetaan yksikkötesti, sitten vaihdetaan näppäimistön haltijaa, joka kirjoittaa yksikkötestin toimimaan saavan koodin ja sen jälkeen seuraavan yksikkötestin, jonka jälkeen hän luovuttaa näppäimistön takaisin. [5]

Eräänlaista pariohjelmointia voi mieltää olevan myös nopeiden suunnittelusessioiden, vaikka ne ovatkin oma, pariohjelmoinnista riippumaton käytäntönsä. Suunnittelusessiot perustuvat siihen, että ohjelmoijat heti huomatessaan kauaskantoisemman suunnittelupäätöksen etsivät käsiinsä toverin, jonka kanssa miettiä, mikä olisi paras ratkaisu. [6]

Pariohjelmoinnin hintaa ja hyötyjä on tutkittu. Eräs 295 hengen tutkimus osoitti, että aloittelevat ohjelmoijat pääsivät ylläpitotehtävissä samaan laatuun yksittäisten kokeneiden ohjelmoijien kanssa, kun he toimivat parina. Aikaa aloittelijaparilla meni tehtävän suorittamiseen saman verran, kuin yksittäisellä kokeneella ohjelmoijalla, eli työtunteja käytettiin enemmän. Pariohjelmoinnin on myös muissa tutkimuksissa todettu parantavan koodin laatua ja hieman pienentävän aikaa, joka ohjelman kirjoittamiseen kuluu [7-9]. Ei ole kuitenkaan varmaa, onko pariohjelmointi kaikissa tilanteissa laadukkaamman koodin tae, tai ainakaan merkittävästi laadukkaamman [9; 10].

2.1.3 Jatkuva integrointi

Kehittäjien koodin jatkuva yhdistäminen niin, että kaikki pysyy keskenään synkronoituna, ei ole yksinkertaista. Siksi on kehitetty erilaisia keinoja varmistaa, että koodin tila pysyy ehyenä. Menetelmä, joka ketterässä kehityksessä tunnetaan jatkuvana integrointina (Continuous Integration), pyrkii helpottamaan koodin pitämistä ajan tasalla ja silti toimivana. Erilaiset versionhallintatyökalut auttavat ohjelmoijien koodien yhdistämisessä hallitusti. Jatkuva integrointi onkin laadunvarmistamiskäytäntö, joka perustuu pitkälti automatisointiin, eli käyttöönoton jälkeen sen eteen ei tarvitse optimitilanteessa juuri nähdä vaivaa. Jatkuva integrointi sopii myös hyvin käytettäväksi yhdessä scrumin kanssa. Scrumin iteratiivisen ja tietyllä tapaa nopeatempoisen luonteen vuoksi myös ohjelmoijien koodin helppo yhdistäminen on toivottavaa.

Yksi tärkeimmistä asioista on se, että ennen uuden ominaisuuden yhdistämistä ohjelman pääversioon voidaan jotenkin varmistaa, etteivät koodin valmiit osat mene rikki. Tähän liittyy käännöstyön automatisoiminen, jota varten on kehitetty useita työkaluja. Automatisoitu käännös ajetaan esimerkiksi aina, kun koodin pääversioon tehdään muutos. Tällöin automaattikääntäjä ottaa koodin versionhallinnasta, yrittää kääntää sen ja ajaa sille testit. Automaattisesti ajettavat testitkin ovat siis osa jatkuvan integroinnin prosessia, sillä ilman niitä voitaisiin tietää vain, kääntyykö koodi vai ei, eikä mitään siitä, toimiiko vanha koodi uuden kanssa.

Jatkuvan integroinnin tavoitteena on vähentää ongelmia, jotka syntyvät, kun useat ohjelmoijat työstävät samaa ohjelmaa. Fowler [11] esittää, että integraatiossa tulevien virheiden määrä vähenee, sillä kaikkien ohjelmoijien koodia ei integroida kerralla sprintin lopussa. Integraation aiheuttama vaiva kasvaa eksponentiaalisesti integrointitaajuuden funktiona. Virheet huomataan helpommin ja aikaisemmin, koska yhteiseen koodiin lisättävät osat ovat pieniä ja yhdistelmä testataan. Edellä mainitun lisäksi Gap Inc.:in siirtymistä jatkuvaan integrointiin kuvaavassa artikkelissa mainitaan, että koska koodi on aina käännettävässä kunnossa ja koska siinä ovat kaikkien muutokset, saadaan julkaisujen taajuutta kasvatettua. Näin saadaan lyhennettyä aikaa, joka kuluu, ennen kuin asiakas näkee jonkin ohjelman osan valmiina. Asiakas tulee myös käyneeksi varmemmin läpi kaikki uudet ominaisuudet, kun yhdessä versiossa niitä ei ole erityisen monta. Näin erityisesti määrittelyn väärinymmärtämisestä johtuvat ongelmat tulevat esille nopeammin. Jatkuvalla integroinnilla saadaan myös suurissa ympäristöissä etuja liittyen tiimien väliseen kommunikointiin. Yksi tiimi ei voi sprintin ajan tehdä parannuksia ominaisuuteen, jonka toinen tiimi aikoo poistaa, koska tämä huomataan heti. [12]

Jatkuvan integroinnin voi viedä myös niin pitkälle, että koodi paitsi käännetään myös julkaistaan asiakkaan näkyville. Tällöin pitää tietysti pitää huolta siitä, että virheet saadaan varmasti kiinni ja että automaattisen käyttöönoton estyminen huomataan. Jos automatisointi johtaa siihen, että käyttäjät joutuvat kärsimään rikkinäisestä ohjelmasta, sitä ei ole toteutettu hyvin.

2.1.4 Test driven development

Test driven development (TDD) on ohjelmointitapa, jossa ohjelmistokehitys perustuu siihen, että ensimmäisenä tehdään ohjelmoitavaa ominaisuutta varten yksikkötesti, joka ei ajettaessa toimi, sen jälkeen kirjoitetaan tarpeeksi koodia, että testi saadaan ajettua onnistuneesti, ja tämän jälkeen koodia refaktoroidaan. Refaktorointi jatkuu, kunnes koodi on tarpeeksi selkeää. Ohjenuorana voivat olla esimerkiksi Kent Beckin yksinkertaisen suunnittelun säännöt [13]. Sama prosessi toistetaan kullekin toteutettavalle ominaisuudelle. Myös löydettyjä virheitä korjatessa voi hyödyntää TDD:tä. Ensin kirjoitetaan testi, joka ei toimi virheen takia, sitten korjataan virhe ja tarkistetaan, että testi menee läpi. Näin saa vahvistettua, että korjaus toimii, ja toisaalta huomataan helposti, jos sama virhe esiintyy jostain syystä uudestaan. [14]

Päähyöty, jota TDD:llä tavoitellaan, on virheiden vähentäminen. Myös koodista voi toivoa tulevan selkeämpää, koska refaktorointi on tärkeä osa prosessia. TDD:stä on monia, keskenään ristiriitaisia tuloksia antavia tutkimuksia. Kokoomatutkimus, joka lajitteli aiempia tutkimuksia niiden tarkkuuteen ja relevanssiin perustuen, tuli tulokseen, jonka mukaan TDD parantaa laatua ja kehitystahti joko hidastuu tai pysyy samana. [15]

2.2 Ohjelmakoodiin liittyvät käytännöt

Kaikki esitellyt käytännöt pyrkivät toki ohjelmakoodin laadun parantamiseen. Tähän kohtaan valitut keinot ovat koodilähtöisiä. Testit ovat itsessään ohjelmakoodia, kommentit kirjoitetaan koodin sekaan, katselmoinnissa käydään läpi koodia ja refaktoroinnissa valmiiksi kirjoitettua koodia muokataan selkeämmäksi muuttamatta sen ulospäin näkyvää toiminnallisuutta.

Käytännöt on esitelty siinä järjestyksessä, jossa niitä projektin aikana käytetään. Testaus voi toki painottua kehityssyklin loppupuolelle, mutta sillä voi myös aloittaa.

2.2.1 Testaus

Testaus on kenties luontevin tapa yrittää vähentää virheiden määrää. Koodin toiminnan varmistaminen formaaleilla menetelmillä on raskasta. Vaihtoehtona on ohjelman toiminnan oikeellisuuden varmistaminen seuraamalla sen toimintaa eli testaamalla.

Test driven developmentissa (TDD) testien kirjoitus on osa ohjelmointiprosessia, niin että testit kirjoitetaan ennen niitä ohjelman osia, jotka tekevät ne toimiviksi. Tämän lisäksi on olemassa muitakin tapoja käyttää ja kirjoittaa testejä. Yksikkötestejä, jotka vastaavat TDD:ssä ennen ohjelmoinnin aloittamista kirjoitettavia testejä, voi kirjoittaa myös jonkin osan valmistumisen jälkeen. Ohjelmoijat voivat käyttää yksikkötestejä varmistamaan, että heidän uusi koodinsa toimii eikä riko vanhoja ominaisuuksia. Yksikkötestien suorittaminen on myös mahdollista automatisoida suhteellisen helposti. Esimerkiksi jatkuva integrointi perustuu pitkälti siihen, että yksikkötestien perusteella luotetaan ohjelman toimivuuteen aiemman koodin kanssa. Kuten TDD:llä myös pelkällä yksikkötestien käytöllä tavoitellaan virheiden vähentämistä. [16]

Testejä voi kirjoittaa myös liittyen tietyn toiminnallisuuden hyväksymiseen pitkälti yksikkötestien tapaan. Tutkiva testaus (Exploratory testing) on myös tapa testata, ja se sopii yhteen ketterien menetelmien kanssa, sillä siinä kaikkia testejä ei suunnitella etukäteen vaan suunnittelu ja testien toteutus tapahtuvat tasaisesti projektin edetessä. Tutkivalla testauksella tavoitellaan sellaisten tilanteiden kattamista, joita yksikkö- tai toiminnallisuustestit eivät ota huomioon. [17]

2.2.2 Kommentointi

Koodin kommentointi on useille ohjelmoijille itsestään selvä asia. Kauhutarinat siitä, kuinka vaikea omaakin kommentoimatonta koodia voi myöhemmin olla ymmärtää, ovat osa kasvamista ohjelmoijaksi. Kommentit voivat kuitenkin helposti jäädä turhan harvoiksi tai päivittämättä ohjelman muuttuessa.

Jos kommentointia ei ajattele, voi helposti päätyä vain kertomaan koodin rakenteesta, joka olisi ymmärrettävissä ilman kommentteja. Ja jos rakenne muuttuu ja kommentteja ei päivitetä, ne johtavat harhaan. Kommentoinnin kannattaisi kuvata rakenteen sijaan sen valintaan liittyneitä päätöksiä, jotka voivat liittyä esimerkiksi ohjelman muihin osiin ja joita koodia lukemalla on vaikea saada selville. Kommentoinnin tarkoitus on siis helpottaa koodin ymmärtämistä ja sitä kautta tehdä siitä selkeämpää, mikä vähentää virheiden riskiä kyseistä osaa kehittäessä ja helpottaa virheiden korjaamista ja muuta jatkokehitystä myöhemmin [18; 19].

Kommentoinnin laadun mittaamiseen ei ole olemassa yleisesti hyväksyttyjä metriikoita. Seurauksena esimerkiksi tutkimus, jossa kehitettiin menetelmää, jolla voi ohjelmallisesti analysoida kommentoinnin laatua, käytti ohjelman tulosten varmistamiseen asiantuntijoiden mielipiteitä kunkin kommentin hyödyllisyydestä [20]. Kommentoinnin laadun mittaaminen on siis pitkälti subjektiivista, vaikka joitakin asioita voidaan myös mitata objektiivisesti. Kommentoinnin lisäksi myös muuttujien ja muiden kokonaisuuksien nimeämisellä on suuri vaikutus koodin ymmärrettävyyteen. Erään tutkimuksen mukaan nimeämisellä on kommentteja suurempi vaikutus, sillä siinä työkseen ohjelmoivat osallistujat eivät juuri lukeneet kommentteja [21].

Kommenttien automaattista generointia on myös kokeiltu. Toistaiseksi näin saatujen kommenttien laatu on kuitenkin riippuvainen ohjelmakoodissa käytetystä nimeämistyylistä. Jos nimeämistyylin ongelmia yritetään kiertää generoimalla kommentit muuten kuin suoraan ohjelmakoodista, kommentit eivät aina pidä paikkaansa [22].

2.2.3 Katselmoinnit

Katselmoinnissa käydään läpi ohjelmakoodia, joko tarkan suunnitelman mukaan ja isolla joukolla tai vain kahden kesken, kenties kokonaan ilman suunnitelmaa. Katselmoinnissa on tarkoitus löytää virheitä tai vaikkapa turhan monimutkaisesti tai epäsiististi ohjelmoituja osia ohjelmasta. Katselmoinnin tarkoitus on siis vähentää virheiden määrää ja selkiyttää lähdekoodia ja myös kasvattaa projektin muiden jäsenten koodituntemusta. Etuna testaamiseen nähden on mahdollisuus huomata tyylivirheitä sekä turhaa tai epätarkoituksenmukaista lähdekoodia ja vaikeasti testeillä huomattavia virheitä. Katselmointeja suoritetaan yleensä valmiille koodin osille. Haikala myös mainitsee, että tarkastusten käyttöönotto ryhdistää työntekoa. Tekijä ei viitsi päästää käsistään mitä tahansa, koska tietää,

että se on julkisesti läpikäytävissä. Huolellisesti suoritettujen katselmointien on osoitettu johtavan tuottavuuden parannuksiin pelkkään testaamiseen verrattuna. [1, ss. 250–254]

Michael Fagan kehitti koodikatselmointiin (Software Inspection) prosessin jo 1970-luvulla. Tämän prosessin mukaan toteutettuna katselmointi on aikaa vievää. Ensin katselmoinnille valitaan aihe ja sitä suunnitellaan. Seuraavaksi järjestetään tilaisuus, jossa katselmoitavan ohjelman osan tekijä esittelee aiheen tulevaan tilaisuuteen osallistuville. Tämän jälkeen kaikki osallistujat lukevat lähdekoodin läpi ja etsivät virheitä. Lopulta varsinaisessa katselmointitilaisuudessa lähdekoodi käydään rivi riviltä läpi ja virheet tuodaan esille. Tämän jälkeen lähdekoodi korjataan ja tarkistetaan. Katselmointitilaisuudessa on myös ennalta määrättyjä rooleja kuten sihteeri ja lähdekoodin esittelijä. [23]

Keveimmillään katselmointina voidaan pitää sitä, että joku lukaisee lähdekoodin läpi. Ohjelmointityökaluista esimerkiksi Git-versionhallinta [24] tukee Pull Request -käytäntöä (PR). Tällöin kukin kehittäjä kehittää ohjelmaa omassa haarassaan. Kukin haara on kopio päähaarasta, jossa on lisäksi kehittäjän tekemät muutokset. Kun päähaaraan halutaan yhdistää kehittäjän haara, tekee kehittäjä PR:n, joka kuvaa hänen muutoksiaan suhteessa päähaaraan. Muut kehittäjät voivat lukea ja kommentoida lähdekoodia, ja tämän varmistamiseksi voidaan esimerkiksi konfiguroida järjestelmä niin, että vasta kun joku on hyväksynyt PR:n, kehittäjän haaran muutokset voidaan yhdistää päähaaraan. Muun muassa Visual Studio Online ja GitHub tarjoavat mahdollisuuden lukemiseen ja kommentointiin graafisen käyttöliittymän avulla. [25]

Osittain automatisoitua katselmointia varten on kehitetty SCRUB-työkalu, joka yhdistää automatisoidun katselmoinnin erilaisten ohjelman rakennetta tutkivien analyysien avulla ja muiden kehittäjien mahdollisuuden kommentoida katselmoitavaa ohjelmaa. Tämä yhdistetään vielä lyhyeen tapaamiseen arvioijien ja kehittäjän kesken. Työkalun arvioitiin huomaavan noin kaksi kertaa niin paljon virheitä kuin ihmisten, ja vääriä hälytyksiä oli vain vähän. SCRUB ei ole suoraan saatavilla, sillä se on yrityksen itselleen rakentama työkalu, joka perustuu useisiin siihen integroituihin kolmansien osapuolien maksullisiin analyysityökaluihin. Erilaisia yksittäisiä analyysityökaluja kuitenkin on saatavilla. [26]

Muistilista kokoaa yksinkertaisia ohjeita, jotka tyypillisesti liittyvät ohjelmointityyliin mutta mahdollisesti myös prosessiin. Lista voi olla tarkistuslista, joka käydään tietyssä vaiheessa läpi, jotta nähdään että sen sääntöjä on noudatettu. Tällöin kyseessä on apuväline yksin tehtävään katselmointiin. Mikään ei toki estä käyttämästä listaa myös muiden lähdekoodin katselmoinnin tueksi. Jos muistilistan haluaa integroituvan tiiviisti osaksi ohjelmointiprosessia, se kannattaa pitää lyhyenä. Näin se on helpompi käydä läpi ja ohjeista tulee nopeammin luontainen osa ohjelmointia.

Ohjelmointityyliä matalalla tasolla vahtivia työkaluja on tarjolla moneen eri ohjelmointiympäristöön. Nämä toimivat sovitun ohjelmointityylin suhteen ikään kuin automaattisina muistilistoina.

Muistilistoja hyödynnetään ohjelmistotuotannossa laajasti. Käytettävyys on ohjelmistotuotannon osa, josta tuntuu löytyvän tarkistuslista lähes kaikelle. Nielsenin heuristiikkoja käytetään usein, ja esimerkiksi matkapuhelinkäyttöliittymille on tehty oma tarkistuslistansa. Ketteriin menetelmiin liittyy joitakin listoja, kuten Kent Beckin yksinkertaisen suunnittelun säännöt, joissa on neljä osaa [13]. Myös NASAlla on oma kymmenen kohdan sääntölistansa kriittisten järjestelmien ohjelmointia varten [27].

2.2.4 Refaktorointi

Refaktoroinnissa kirjoitetaan jokin osa koodia uudelleen niin, että sen toiminta ei ulkopuolelta katsoen muutu. Tarkoitus on muokata koodia niin, että siitä tulisi yksinkertaisempaa ja siten helpommin ylläpidettävää. Refaktorointi on riskialtista ilman yksikkötestejä joilla varmistetaan, että koodi toimii refaktoroinnin jälkeen samoin kuin ennen sitä.

Refaktoroinnin hyödyllisyydestä on esimerkiksi tutkimus [28], jossa huomattiin että ohjelmistokehittäjien käsitys refaktoroinnista eroaa sen määritelmästä. Kehittäjien käytössä refaktorointi tarkoitti myös sellaisia muutoksia, jotka muuttivat ohjelman osan ulospäin näkyvää rajapintaa. Refaktoroinnin kuitenkin todettiin johtavan vakaampaan koodiin. Toisessa tutkimuksessa huomattiin, että ohjelman monimutkaisuuden taso ei muuttunut refaktoroinnin seurauksena. Asiaa selittää osin se, että monet refaktoroinniksi merkityt ja versionhallintaan lisätyt päivitykset olivat sekä refaktorointia että jotain muuta, kuten yksikkötestien lisäyksiä. Refaktorointi voi myös olla koodin järjestelytyötä, joka ei yksinkertaista koodia, mutta voi parantaa luettavuutta. [29]

3. OHJEISTUS OHJELMISTOPROJEKTISSA

Ohjeistuksen voi käsittää monella tavalla. Yleisellä tasolla ohjeistus voi olla liki mitä tahansa ravintolan kassaan liimatusta ohjelapusta yrityksen tarkoin dokumentoituun alihankkijoiden hyväksymisprosessiin. Ohjelmistoprojektissa ohjeita on esimerkiksi liittyen arkkitehtuuriin, projektinhallintaan ja ohjelmointityyliin.

Ohjeilla voi myös olla monia tavoitteita. Pohjimmiltaan ohjeilla pyritään välttämään virheitä tai huonoja tapoja, erityisesti sellaisia, joista on aiemmin ilman ohjeita tullut ongelmia. Yleisiä tavoitteita suunnitelluilla ohjeilla ovat projektin budjetissa ja aikataulussa pysyminen sekä laatuun liittyvät tavoitteet. Suunnittelemattomilla, lapulle kirjoitetuilla ohjeilla voi yrittää parantaa laatua muistuttamalla tehtävistä tai auttaa ratkaisemaan ongelmia.

Maksujärjestelmäprojektissa, jota tässä työssä käytetään esimerkkinä, projektin aikataulussa pysyminen ja laadun varmistaminen ovat tärkeitä tavoitteita, ja tästä syystä sille suunnitellaan ohjeet. Tässä luvussa keskitytään erityisesti harkittuihin ohjeisiin, joilla tavoitellaan ohjelmistoprojektissa ohjelmakoodin korkeaa laatua.

3.1 Motivaatio laatua varmistavalle ohjeistukselle

Laatua parantava ohjeistus ei ole millään muotoa vain ohjelmistoalaan liittyvä ilmiö. Esimerkiksi erilaisille ihmisen ohjaamille koneille on jo pitkän aikaa kehitetty standardeja, jotta kuljettajat olisivat turvassa riippumatta siitä, kuka ohjaamon on valmistanut. Standardeja löytyy monelta muultakin alalta. Asiakaspalvelijoille taas on tyypillistä esimerkiksi työhöntulon yhteydessä antaa jokin lyhyt ohje, jossa on kirjattuna ylös yleiset tilanteet ja mitä niissä tulisi tehdä. Näin saadaan varmistettua toiminnan yhtenäisyys ja reiluus asiakkaille.

Ohjelmakoodin laatu on tärkeä monesta eri syystä. Yksi syy on asiakastyytyväisyys, joka on merkittävä asia. Jos asiakkaat eivät ole tyytyväisiä, he tuskin käyttävät samaa toimittajaa uudelleen ja kertovat muillekin, että heillä on huonoja kokemuksia. Toinen syy on tuottavuus, sillä virheiden korjaaminen on kallista. Yrityksen pitää kuitenkin olla taloudellisesti kannattavalla pohjalla, eli useinkaan ei ole syytä tavoitella täydellistä laatua vaan laadun tasoa, jonka voi saavuttaa kannattavasti ja tehokkaasti. Luonnollisesti laatu on tärkeää projekteissa, joissa virheistä on vakavia seurauksia. Sairaaloiden laitteet ja tietojärjestelmät sekä esimerkiksi lentokoneet, laivat ja muut kulkuvälineet ovat sellaisia, joiden ohjelmistojen virheet voivat uhata ihmishenkiä. Ihmisten henkilötietoja koskevat tai rahan käsittelyyn liittyvät ohjelmistot ovat myös sellaisia, joissa ei soisi olevan virheitä. Eivätkä virheet esimerkiksi peleissäkään ole mukavia, mutta siellä ne eivät uhkaa

ihmisten henkiä tai yleensä omaisuutta tai yksityisyyttä. Maksujärjestelmäprojektin virheet eivät uhkaa ihmishenkiä, mutta käyttäjien omaisuutta kyllä, eli laatu on tärkeää.

Kun aikataulu on tiukka, ohjelmakoodin selkeys voi kärsiä. Tällaisessa tilanteessa voidaan myös helposti tinkiä testien kattavuudesta tai kommentoinnista. Jokin osa ohjelmasta voi myös jäädä liian yksinkertaiseksi, kun kaikkea siihen liittyvää ei kiireessä tule mieleen. Huonolaatuinen ohjelmakoodi on projektissa todennäköistä, kun aikataulu on tiukka ja kehittäjiä on paljon ja he tulevat erilaisista taustoista. Tämä voi vaikuttaa etenkin ohjelmakoodin tyyliin ja sisäiseen rakenteeseen. On aivan mahdollista, että ilman mitään erikoistoimia ohjelmakoodi on jossain kohtaa hyvin vaikealukuista. Usean kehittäjän takia koodia myös kehitetään aiemman koodin päälle nopeaan tahtiin. Tämä muodostuu helposti ongelmaksi, jos vanhoissa ohjelman osissa on runsaasti virheitä. Nämä virheet voivat vaikuttaa löytövaiheessa jo hyvin monen ihmisen koodiin.

Laatu on tärkeää asiakastyytyväisyyden kannalta. Tämän lisäksi rahaa käsitellessä on mahdollisuus, että väärin ohjelmoiduilla rahansiirroilla hävitetty raha tulisi oikeusteitse kehittäjäyhtiölle maksettavaksi. Ja erityisesti laatu on tärkeää, jotta aikataulussa pysytään. Virheet ovat kalliita koska ne heikentävät asiakkaan luottamusta ohjelman tuottajaan ja koska niiden korjaamiseen menee paljon aikaa. Sekava tai vaikeasti luettava koodi voi aiheuttaa virheitä, ja erityisesti sen tulkitsemiseen menee kehittäjiltä aikaa. Onkin tutkittu, että ohjelmakoodin lukemiseen kuluu ohjelmaa ylläpidettäessä paljon aikaa. Microsoftilla suoritetussa tutkimuksessa tulos oli, että noin puolet ohjelmointiajasta kuluu lukemiseen [30]. Vaikka kyse ei ole ylläpidosta, joutuu muiden kanssa tiiviissä yhteistyössä ohjelmaa kehittäessä edelleen lukemaan koodia. Maksujärjestelmäprojektissa on paljon rajapintoja, joissa eri ohjelmoijien koodit kohtaavat. Lukemisen määrää vanhan koodin ylläpitoon verrattuna vähentänee se, että erikoisemmissa kohdissa pääsee kysymään suoraan tekijältä.

Maksujärjestelmäprojektissa käytössä on scrum, joka on ketterän kehityksen menetelmä. Ketterän kehityksen henkeen liittyy se, että dokumentaatiota tehdään sen verran kuin tarvitaan. Toimiva ohjelmakoodi on tärkeämpää. Samoin keskitytään yksilöihin ja kommunikaatioon tarkkaan määriteltyjen prosessien ja työkalujen sijaan, ja sen avulla pyritään saavuttamaan toimiva ja etenevä ohjelmistoprojekti. Dokumentoidut ja tarkkaan määritetyt ohjeet eivät kuulu scrum-menetelmässä määriteltyihin työkaluihin, vaikka se ei suoraan sellaisia kielläkään. Projektin laatua uhkaavassa ja toisaalta vaativassa tilanteessa ohjeet ovat väline, jolla voidaan ohjata kehittäjien huomiota laatua parantaviin asioihin. Vaikka kommunikaatio on tärkeää ja sillä voi päästä pitkälle, ovat esimerkiksi tyyliohjeet hankalia välittää tai noudattaa ilman dokumentointia. Erityisesti projektiryhmän vaihtuvuus aiheuttaa tarpeen dokumentaation olemassaololle, jotta uusille jäsenille pystytään välittämään tietoa ilman, että jäädään vain muistin varaan. Erilaisten taustojen takia laadunvarmistusmenetelmät eivät vielä ole kaikille yhtä tuttuja, ja toisaalta yhtenäiselle tyyliohjeistukselle on selvä tarve, jotta koodin lukeminen pysyy mahdollisimman helppona.

Haikala mainitsee, että ohjelmistotuotannon erityispiirteiden takia ohjelmistotuotteen laadukkaassa valmistuksessa painotetaan hyviä toimintatapoja [1, s. 174]. Tällaisia toimintatapoja pyritään projektin ohjeistuksella dokumentoimaan, vahvistamaan ja luomaan. Syyt ohjeistuksen puolesta ovat niin vahvoja, että päätettiin luoda dokumentoitu ohjeistus, joka toki saa ketterän kehityksen hengessä muuttua, jos huomataan parannettavaa.

3.2 Millainen on hyvä ohjeistus?

Maksujärjestelmäprojektissa ohjelman laatu haluttiin varmistaa. Ohjeet pitää valita tämä tavoite mielessä pohtien, mihin ottaa kantaa ja miten ohjeet muotoilee. Koodin selkeyttä, sisäistä yhteensopivuutta ja muita laatuominaisuuksia uhkaavat laajassa projektissa erityisesti kommunikoinnin puutteet ja ihmisten raskaat työtaakat.

Tämän työn käsittelemässä projektissa oli alusta asti käytössä scrum, ja se sisältää monia työskentelytapoja, jotka auttavat osaltaan laadun varmistamisessa. Nämä tavat liittyvät muun muassa kommunikointiin ja työn jaksottamiseen. Scrum oli myös tuttu ohjelmoijille. Näin ollen sen kattamiin osuuksiin, jotka liittyvät lähinnä projektinhallintaan, ei kehitetty erillisiä ohjeita.

Vaikka scrum osaltaan painottaa esimerkiksi sitä, että kerralla ei oteta liikaa tekemistä, on projektin aikataulu niin tiukka, että tietoisuus siitä painoi varmasti jossain määrin kaikkia. Kiire on uhka ohjelman laadulle, sillä kiireessä on helppo tinkiä esimerkiksi kommentoinnista, joka selkeyttäisi koodia ja voisi siten auttaa jopa virheiden ehkäisyssä. Jotta ohjelmoijat saisivat kiireestä huolimatta pidettyä kiinni laadusta, tarvitaan jotain enemmän.

Päätettiin muodostaa ohjeistus, joka sisältää erilaisia laadun varmistamiseen liittyviä menetelmiä, joita noudattaa. Erilaisia laadunvarmistusmenetelmiä esitellään seuraavassa luvussa. Tutkituista menetelmistä valittiin projektiin sopivat, joista muodostettiin ohjeistus, joka esitellään luvussa 5.

Ohjelmointistandardien noudattamista on tutkittu opiskelijoilla, lähinnä standardien opettamisen ja noudattamaan rohkaisemisen näkökulmasta. Eräässä tutkimuksessa [31] nousi esiin, että opiskelijat oppisivat mieluiten standardeja esimerkin kautta. Ensimmäistä ohjelmointikurssiaan käyvät eivät suosineet dokumentteja opetustapana, mutta vanhemmat opiskelijat hyväksyivät dokumentaation standardien oppimistavaksi selvästi paremmin. Luennot ja harjoitukset olivat uusien opiskelijoiden suosiossa; vanhemmat opiskelijat suhtautuivat niihin vähemmän positiivisesti. Tutkimukseen osallistujille tuttuja standardeja olivat nimeämiseen, kommentteihin ja dokumentaatioon liittyvät ohjeet. Työkseen ohjelmoivien voisi tutkimuksen perusteella olettaa tulevan toimeen myös dokumentteina annettujen ohjeiden kanssa, mutta muotoiluun ja noudattamisen motivointiin pitää kiinnittää huomiota.

ISO 9001 -laatustandardissa yksi keskeinen periaate on ohjeajattelu, joka edellyttää muun muassa takuita siitä, että ohjeet ovat kaikkien saatavilla eikä vanhentuneita ohjeita käytetä. Toinen periaate, näkyvyysajattelu, taas edellyttää, että laatujärjestelmän mukaisesta toiminnasta jää todisteita, jotta voidaan todistaa, että toiminta on ohjeistuksen mukaista. Standardia noudattavan yrityksen laatujärjestelmän dokumentaatioon kuuluvia ohjeita ovat laatukäsikirja, jonka sisällysluettelo voi olla ISO 9000-3 -standardista, työohjeet eri työvaiheisiin kuten määrittelyyn, eri työvaiheiden dokumentointikäytännöt sekä viitemateriaalit eli esimerkiksi työvälineiden käyttöohjeet. [1, ss. 189–191] Jos tavoitteena ei ole sertifikaatin saaminen, aivan kaikkea tuskin ohjeistetaan.

Ohjeiden muotoilu on mielenkiintoinen ongelma. Ohjeiden tulisi luonnollisesti olla helppolukuiset. Ylimalkaisia ohjeita on vaikea noudattaa, ja käytännönläheisyys sekä esimerkit auttavat ymmärtämistä. Esimerkiksi Haikala toteaa, että laatukäsikirja, joka sisältää pelkästään ISO 9001 -standardin tekstin, ei houkuttele ketään sitä lukemaan, sillä dokumentti on epäkonkreettinen ja kapulakielinen [1]. Toisaalta ohjeet eivät saa olla liian pitkät, jotta niiden lukeminen ei veisi liikaa aikaa. Erityisesti ohjeiden lukemisen kun voi mieltää hyvin tuottamattomaksi työksi. Vaikka ohjelmoijat, projektipäälliköt ja useimmat asiakkaat tietänevät, että virheet maksavat sitä vähemmän, mitä aiemmin ne huomataan [2], joskus tuottavan työn vaatimukset vain ovat akuutimmin mielessä. Kiireisenkin projektin etu on kuitenkin se, että ongelmat huomataan ajoissa, sillä suuri osa kehitysvaiheen mukaan kohoavasta korjaamisen hinnasta selittyy sillä, että korjaus vie myöhemmin enemmän aikaa.

Maksujärjestelmäprojektia varten tehdään lyhyet ohjeet, jotka voi pitää aina mielessä tai tarkistaa nopeasti. Tämän lisäksi ohjeisiin liitetään linkkejä perusteluihin niiden takana, sillä ihmiset eivät tee asioita joita eivät miellä järkeviksi. Haikala toteaa laatujärjestelmien kehityshankkeiden hiipuvan toisinaan, koska toiminnan ongelmia ja sen kehittämisen mahdollisuuksia ei voida perustella vakuuttavasti [1, s. 186]. Lisäksi ohjeisiin voi liittää toisia, erillisiä ohjeita erikoistilanteisiin. Esimerkiksi projektipäällikkö voi sitten huomauttaa ohjeen olemassaolosta erikoistilanteen koittaessa.

3.3 Ohjeiden noudattaminen

Usein on vaikeuksia noudattaa ohjeita, jos ei ymmärrä syitä niiden takana. Tällöin ohjeita saatetaan noudattaa, mutta vain pintapuolisesti, jolloin hyötyjä ei juuri ole odotettavissa. Ohjeita muodostaessa kannattaa siis ottaa projektin työntekijät mukaan ja antaa heille mahdollisuus vaikuttaa. Näin ohjeista saadaan projektitiimin yhteiset. Vaikka annetaan tilaisuus vaikuttaa, voi se jäädä käyttämättä, jos vaikuttaminen pitää tehdä oman työn ohella. Näin ollen olisi hyvä esimerkiksi varata yhteistä aikaa, jolloin kaikki kävisivät esiversiota ohjeista läpi ja miettisivät omia lisäyksiä. Toisaalta kaikki voivat olla innoissaan tekemässä ohjeita, ja sitten ne jäävät pöytälaatikkoon eikä niitä käytetä. Eräässä pro-

jektissa scrumin käyttöönottoon suhtauduttiin ensin hyvin innokkaasti, mutta kun se aiheutti lisätyötä, sitä ei toteutettu aivan oikein, ja kun hyödyt eivät olleet heti näkyvissä, niin innostus muuttui katkeruudeksi [32].

Kun on olemassa järkevä kokonaisuus ohjeita, joista on päivittäiseen käyttöön muistilista, perustelut saatavissa ja kenties yksityiskohtaisempia ohjeita erikoistilanteisiin, ollaan jo melko pitkällä varsinkin jos ohjeet on tehty yhteistyössä niiden ihmisten kanssa, jotka niitä tulevat noudattamaan. Tämän jälkeen ohjeet pitää kuitenkin esitellä ihmisille. Sähköpostin lähettäminen on yleinen tapa tiedottaa asioista. Sähköposti, jossa ohjeet ovat liitteenä, olisi siten yksi mahdollinen tapa esittelyyn. Mutta sähköpostia saadaan paljon, ja ne jäävät helposti vähälle huomiolle. Samoin ohjeiden lukeminen ja sisäistäminen jää kunkin omalle vastuulle, jolloin edessä on valinta siitä, tekeekö uusia ominaisuuksia ohjelmaan vai lukeeko ohjeistusta. Ohjeet kannattaisikin käydä joukolla läpi, jotta ihmiset saavat kuulla niiden takana olevat perustelut ja selvennyksiä mahdollisiin epäselviin kohtiin. Tutkimuksessa [32] ehdotettiinkin yhtenä keinona yksinkertaisesti ajan varaamista uuden menetelmän käyttöönottoon.

Ohjeita läpikäytäessä voi ottaa esille sen, miten niiden käyttöä tullaan valvomaan. Ja varsinkin kun ohjeiden noudattaminen vaatii vaivannäköä, pitää jonkinlaista kontrollia olla. Valvonnan lisäksi uuden menetelmän noudattamisen tueksi on ehdotettu siitä saatujen positiivisten kokemusten jakamista [33]. Tuottavuuden vaade voi ajaa oikaisemaan, vaikka ohjeet kuinka tuntisi omikseen. Scrumin käyttöönottoa case-esimerkin ja kirjallisuuden avulla käsitelleessä tutkimuksessa eräitä löydettyjä ongelmia olivat käyttöönoton vaikeudet sekä työpaineiden aiheuttama luistaminen ohjeista. Kun käytännössä huomattiin, että scrumin noudattamiseen kuluu jonkin verran aikaa eikä lisähyötyjä heti nähty, scrumista alettiin joustaa. Ratkaisuiksi ehdotettiin tiimien välistä yhteistyötä menetelmien käyttöönotossa, ohjaajaa ja allokoitua aikaa uuden prosessin käyttöönotolle. [32] Ohjaajan voi tässä tapauksessa tulkita ihmiseksi, joka osaa kertoa ohjeista ja vahtia niiden noudattamista. Haikala totesi laatujärjestelmien kehityshankkeiden usein hiipuvan, koska johto ei tue muutoksia tai sitoudu niihin, koska liian isoja muutoksia yritetään kerralla tai koska koulutuksessa ja tiedottamisessa on puutteita tai muutokset vaativat työaikaa [1, s. 186]. Syyt ovat siis samankaltaisia kuin scrum-tutkimuksessa löydetyt, ja ainakin koulutuksen ja tiedotuksen puutteissa voisi ohjaaja auttaa.

Ohjelmistotuotantoprosessiin liittyvien ohjeiden noudattamisen valvominen on vaikeampaa kuin suoraan koodiin liittyvien. Tähän on hyvä olla nimetty henkilö, esimerkiksi projektipäällikkö, arkkitehti tai kuka tahansa projektiryhmäläinen, jolle tehtävä luontevasti asettuu. Hänen tehtävänään on kannustaa noudattamaan ohjeita, auttaa siinä ja osoittaa, että asiasta välitetään.

Kaikkeen ei tarvita nimettyä valvojaa, vaan osa tästä voidaan automatisoida. On esimerkiksi työkaluja, joilla voi valvoa koodausstandardien käyttöä. Nämä työkalut pystyvät

valvomaan vain melko yksinkertaisia, koodiin sidottuja standardeja [34]. Git-versionhallintaa käyttäessä voi estää uuden ohjelman osan yhdistämisen ohjelmaan, jos kukaan ei ole hyväksynyt sitä. Jos tätä käytäntöä noudatetaan, voivat ohjelmoijat muistuttaa toisiaan lähdekoodiin liittyvien sääntöjen noudattamisesta, kun he käyvät lähdekoodin läpi hyväksymistä varten. Opiskelijoilla tehdyn tutkimuksen johtopäätöksenä tekijät arvioivat, että opiskelijoiden keskinäinen koodikatselmointi olisi hyvä tapa hyödyntää harjoitusta ja esimerkkiä koodausohjeiden opettelussa [31].

4. MAKSUJÄRJESTELMÄPROJEKTI

Maksujärjestelmän suunnittelu kesti muutaman kuukauden, ja tämän jälkeen sen toteuttamiseen oli aikaa noin yhdeksän kuukautta. Järjestelmän toiminnot on suunniteltu ennakkoon, ja osa toiminnoista voidaan viimeistellä hieman myöhemmin. Toisaalta taas voidaan toteuttaa joitain lisäominaisuuksia, jos ne vaikuttavat erityisen hyödyllisiltä. Projekti on toteuttajalle tärkeä, sillä järjestelmälle oletetaan olevan kysyntää ja siihen voi myös myydä jatkokehitystä.

Projekti haluttiin maaliin yhdeksän kuukautta ohjelmoinnin aloittamisen jälkeen. Jotta projekti valmistuisi ajoissa, toteuttaja otti heti alussa projektiin mukaan mahdollisimman paljon ohjelmistosuunnittelijoita. Ohjelmoijia oli tarkoitus vähentää loppuvaiheessa, jos näyttäisi siltä, että projekti valmistuu tehdyn työtuntiarvion puitteissa.

Toteutukseen valittiin scrum, ja .NET-ympäristön versio 4.5.2 [35], ohjelmointikielinä olivat C# [36] sekä web-käyttöliittymässä JavaScript [37] ja tietokantapalvelimena toimi Microsoft SQL Server [38] versio 2012. Tehtävien ja virheiden hallintaan sekä jatkuvaan integrointiin käytettiin Visual Studio Onlinea [39] ja versionhallintaan Gitiä.

4.1 Projektiryhmä

Projektiryhmän koko kasvoi sitä mukaa, kun ohjelmakoodia saatiin kirjoitettua siihen vaiheeseen, että uusiin rajapintoihin, sisäisiin tai ulkoisiin, päästiin käsiksi. Rekrytointien ja muista projekteista vapautuneiden ihmisten myötä päädyttiin hieman ennen ohjelman valmistumista tilanteeseen, jossa kehittäjiä oli parhaimmillaan kolme kertaa niin paljon kuin projektin alussa. Projektin deadlinea edeltävän kuukauden alussa kehittäjien työtunteja alkoi hiljalleen kulua eri projekteihin.

Ohjelmoijien joukko oli siis melko suuri ja hajanainen. Ihmisillä oli erilaisia taustoja: jotkut tulivat suoraan koulun penkiltä, toiset eri yrityksistä ja yksi henkilö oli harjoittelija. Projektissa käytetty C# ei ollut kaikille tuttu kieli; käyttöliittymässä käytetty JavaScript tosin oli kaikille käyttöliittymäohjelmoijille tuttu. Projektiryhmällä on siis varmasti eroja ohjelmointitaidoissa ja -tavoissa, ja he tulivat projektiin eri vaiheissa, jotkut myöhäänkin.

Projektiryhmässä oli joitakin selkeitä rooleja. Arkkitehti oli arkkitehtuuripalavereissa mukana, ja hän oli erityisen aktiivinen Pull Requestien kommentoija ja monessa ongelmatilanteessa ratkaisija. Arkkitehti myös jakoi tiimin chatissa ahkerasti linkkejä, joiden koki olevan relevantteja projektin ohjelmoijille. Projektipäällikkö toimi määrittelijänä ja huolehti määrittelyn välittymisestä tehtäviksi ja niiden priorisoinnista. Hän myös seuloi virheraportteja ja välitti eteenpäin ohjelmoijien tarkennuskysymyksiä määrittelyyn liittyen. Projektiryhmää varten oli varattu maksujärjestelmien asiantuntija, jonka rooli oli

auttaa määrittelyn tarkentamisessa. Loppuvaiheeseen jääneistä tehtävistä osa tosin vaati niin paljon määrittelyä, että ohjelmoijat ottivat häneen suoraankin yhteyttä.

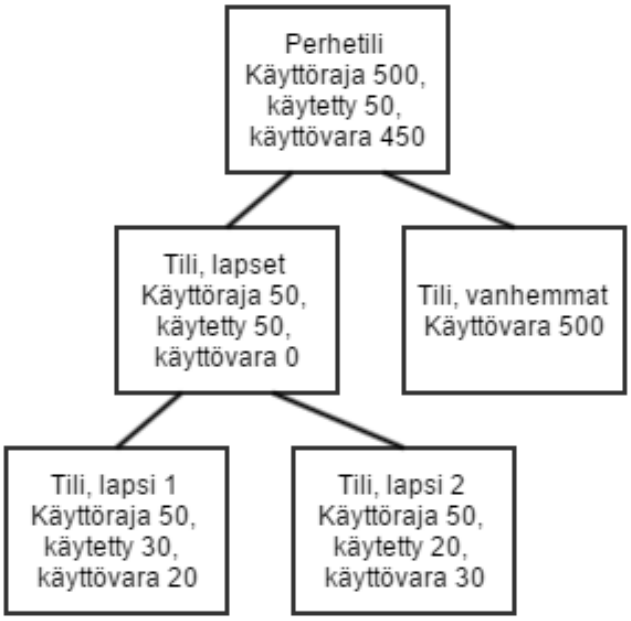
Projektiryhmässä oli yksi testaaja, joka teki järjestelmätestausta. Lisäksi oli erillinen testaustiimi, joka toimi eri tiloissa kuin kehittäjät. Virheiden raportointiin käytettiin Visual Studio Onlinea.

4.2 Järjestelmän kuvaus

Maksujärjestelmällä hallitaan järjestelmän ostajan maksutoimintaa. Tällainen järjestelmä integroituu useaan muuhun järjestelmään rajapintojen avulla. Liitynnöistä sisäinen ja ulkoinen käyttöliittymä toteutetaan itse, ja ne poikkeavat näin muista. Muut rajapinnat ovat kolmansien osapuolien hallinnoimia, joten niihin ei tehdä tässä projektissa muutoksia. Kuva 4.1 esittää liityntöjä.

Kuva 4.1: Maksujärjestelmä rajapintoineen.

Järjestelmässä on rakennettu mahdollisuus hierarkiaan. Kuva 4.2 kuvaa hierarkian. Asiakkaan tiliin voi liittyä alatilejä, esimerkiksi perheen tilin alla voi olla vanhemmille ja lapsille omat tilinsä. Jokaisella hierarkian tasolla asetetaan käyttöraja. Kun tilillä yritetään tehdä ostoksia, varmistetaan järjestelmältä, onko tililtä jo käytetty liikaa rahaa, eli onko sillä oikeus ostoon. Jokaisella hierarkian tasolla pitää olla riittävästi käyttövaraa ostoksen suorittamiseen. Onnistunut osto kirjataan muistiin ja se pienentää käyttövaraa. Jos kuvan tapaan on kaksi tiliä, joiden käyttöraja on 50 euroa, kummankin käyttö estetään, kun niiden yhteenlasketut ostot ovat 50 euroa, sillä niitä yhdistävällä tilillä ei ole enää jäljellä käyttövaraa. Tämä hierarkia tekee käyttövaran varmennuksesta oston yhteydessä monimutkaista. Hierarkiaan on myös mahdollisuus lisätä periytyviä parametreja.



Kuva 4.2: Maksujärjestelmän hierarkia.

Järjestelmässä on kaksi graafista käyttöliittymää, ja kummatkin ovat web-pohjaisia. Sisäinen käyttöliittymä on järjestelmän ostajan käyttöön tarkoitettu. Tällä käyttöliittymällä pystyy muokkaamaan asiakastietoja ja tekemään jonkin verran rahan käsittelyä, kuten luomaan laskutettavia palkkioita ja valitsemaan, mihin asiakkaan maksusuoritus käytetään. Käyttöliittymällä voidaan myös tallentaa uusia asiakkaita.

Ulkoinen käyttöliittymä on loppuasiakkaille tarkoitettu käyttöliittymä, josta he voivat tutkia omia tietojaan. Muutoksia ei tämän käyttöliittymän kautta voi tehdä. Tarjolla on myös ulkoisen käyttöliittymän rajapinta, jota hyödyntäen kuka tahansa voi tehdä omia toteutuksiaan.

Järjestelmän ylläpitäjille ei ole erityistä käyttöliittymää, mutta heillä on pääsy tietokantaan, lokeihin ja tietokantaskripteihin, joita ajamalla saa tietoa järjestelmän tilasta. Lokiin virhe-tasolla tulevista viesteistä lähtee ylläpidolle sähköposti ja tietyistä erityisen kriittisistä virheistä tekstiviesti.

Suuri osa järjestelmän toiminnallisuudesta perustuu erilaisiin päivittäin tai kuukausittain suoritettaviin eräajoihin. Kaikki eräajot toimivat lopulta ajastetusti, mutta aluksi niitä voidaan vähäisen määrän johdosta ajaa myös käsin. Ajastuksia ei siis ole pakko toteuttaa yhdeksän kuukauden aikarajan puitteissa. Järjestelmän toiminnan mahdollistaa ajantasainen tieto erilaisista tileihin liittyvistä tapahtumista. Tietoa ostotapahtumista eli maksuaineistoa tulee päivittäin, samoin koonti asiakkaiden maksusuorituksista. Näiden sisäänajon lisäksi päivittäin ajetaan allokointi, jossa maksusuorituksia jyvitetään tilin auki oleville laskuille. Laskut luodaan ja korot lasketaan määräajoin, ja korkoa kertyy vain las-

kuille, joiden maksut ovat myöhässä. Pohjautuen ajantasaisiin tietoihin asiakkaista ja heidän maksutilanteestaan järjestelmä antaa tiliä käytettäessä tiedon siitä, onko sillä ostooikeus. Järjestelmän on tuettava rinnakkaisuutta, jotta jokainen loppuasiakas voi käyttää tiliään viiveettä.

Eräajot kuten laskutus, suoritetaan muulloin kuin ruuhka-aikaan, mutta myös niiden aikana järjestelmän on noudatettava vasteaikoja. Esimerkiksi tilin käytön hyväksymiselle on sovittu vasteajat, joissa tulee pysyä. Koska järjestelmässä käsitellään rahaa ja henkilötietoja, myös tietyistä tietoturvavaatimuksista on sovittu, ja henkilötietojen kohdalla niistä määrää jo lakikin.

4.3 Projektin liiketoimintakriittiset osiot

Järjestelmä on liiketoiminnallisesti tärkeä, koska sen myyntipotentiaalia pidetään suurena ja toisaalta koska järjestelmä käsittelee rahaa. Jos rahaa käsittelevässä koodissa on virhe ja tästä syystä järjestelmän ostaja menettää tuloja, voi tuottaja jopa joutua korvausvaatimuksen kohteeksi. Ohjelman rahaa käsittelevien osien laatuun on siis syytä kiinnittää erityistä huomiota, tosin jossain määrin koko järjestelmä toki liittyy rahaan.

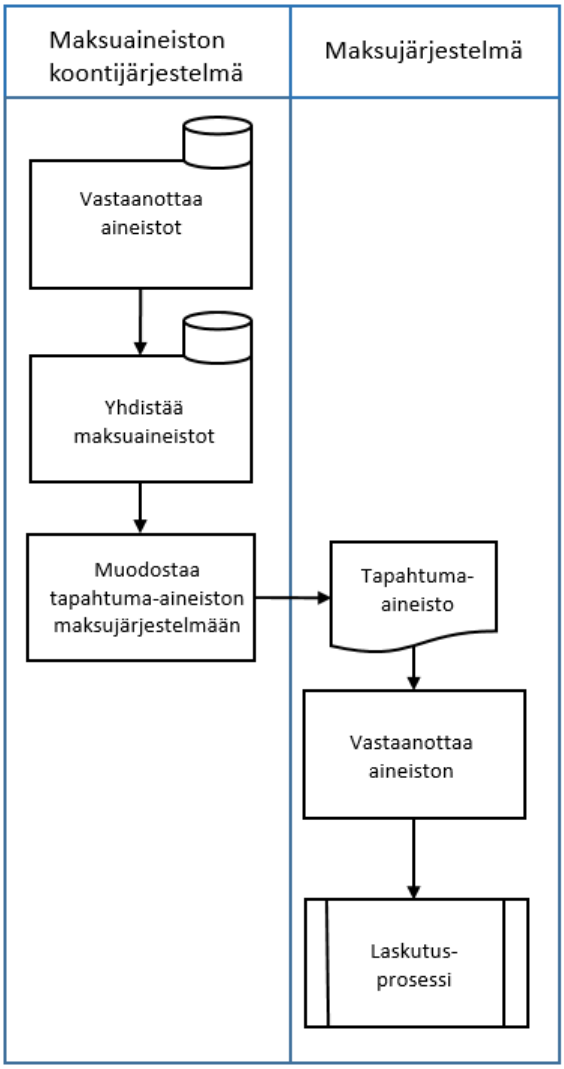
Liiketoiminnan kannalta erityisen kriittisiksi osiksi on valittu osat, jotka liittyvät hyvin tiiviisti rahaan, ja ne esitellään tässä luvussa. Kaikkein kriittisintä voi sanoa olevan tilin ostoluvan tarkistuksen, sillä jos se ei toimi, tilin käyttö estyy. Siksi sen saatavuudeksi onkin kirjattu järjestelmän ei-toiminnallisiin vaatimuksiin tiukimmat rajat. Tilin ostoluvan tarkistusta koskevat virheet on määritelty sellaisiksi, että sähköpostihuomautus ei riitä vaan ylläpito saa myös tekstiviestin.

Jotta tilejä voi järkevästi käyttää, tieto ostoluvasta pitää saada nopeasti, mutta myös eräajot, erityisesti allokointi, vaikuttavat maksujärjestelmän toiminnan oikeellisuuteen. Kun maksu allokoidaan tilille, tilin ostoihin käytettävissä oleva käyttövara nousee. Myös laskutuksen oikea toiminta on tärkeää. Jos tällaiset perustoiminnot eivät toimi, on järjestelmä käyttökelvoton. Paitsi että raha jakautuu väärin, myös kirjanpitoon voi tulla epätäsmäävyyksiä. Rahan oikean jaon selvittäminen jälkeenpäin ja kirjanpidon saattaminen ajan tasalle voivat olla hyvinkin vaivalloisia ja kalliita toimenpiteitä.

Kaikissa tilanteissa tulee pitää huolta siitä, että tilit täsmäävät ja tehdyt toiminnot jäävät lokiin. Jos on luotu virheellinen rivi laskulle, sitä ei voi poistaa tai muokata, vaan on tehtävä samansuuruinen negatiivinen rivi. Samoin jos loppukäyttäjä esimerkiksi maksaa liikaa, ei tehdystä suorituksesta voi käyttää osaa maksuihin ja muuttaa ylisuorituksen summaa jäljellä olevaksi summaksi, vaan täytyy luoda ylisuoritus ja pitää alkuperäinen suoritus muuttumattomana. Näin varmistetaan, että kirjanpidossa ei tule ongelmia.

4.3.1 Maksuaineiston tallennus

Maksuaineisto sisältää tiedot tileillä tehdyistä maksuista. Se oli ensimmäinen asia, joka järjestelmässä otettiin työn alle, ja se on esiehto monelle osalle järjestelmän toiminnasta kuten allokoinnille, kirjanpitoaineistoviennille ja laskutukselle. Kuva 4.3 esittää prosessin.



Kuva 4.3: Maksuaineiston sisäänlukuprosessi.

Maksuaineiston koontijärjestelmä vastaanottaa aineistot, jotka sisältävät tiedot tileillä tehdyistä maksuista. Se muodostaa niistä maksujärjestelmää varten tapahtuma-aineiston. Maksujärjestelmä vastaanottaa aineiston ja tallentaa sen sisällön tietokantaan. Kun on laskutuksen aika, laskut muodostetaan käyttäen tätä aineistoa. Tapahtuma-aineisto siirretään maksuaineiston koontijärjestelmästä kerran vuorokaudessa maksujärjestelmään

Alkuperäistä tapahtuma-aineistoa ei koskaan muokata tai poisteta. Mahdolliset virheet korjataan lisäämällä korjaustapahtumia. Tapahtumasta tallennetaan sisäänlukupäivä sekä

kirjanpitojärjestelmään vientipäivä. Tapahtumapäivän pitää kulkea kirjanpitojärjestelmään kautta linjan, jolloin järjestelmä vie tapahtuman oikealle kaudelle.

4.3.2 Maksusuoritusten sisäänluku ja allokointi

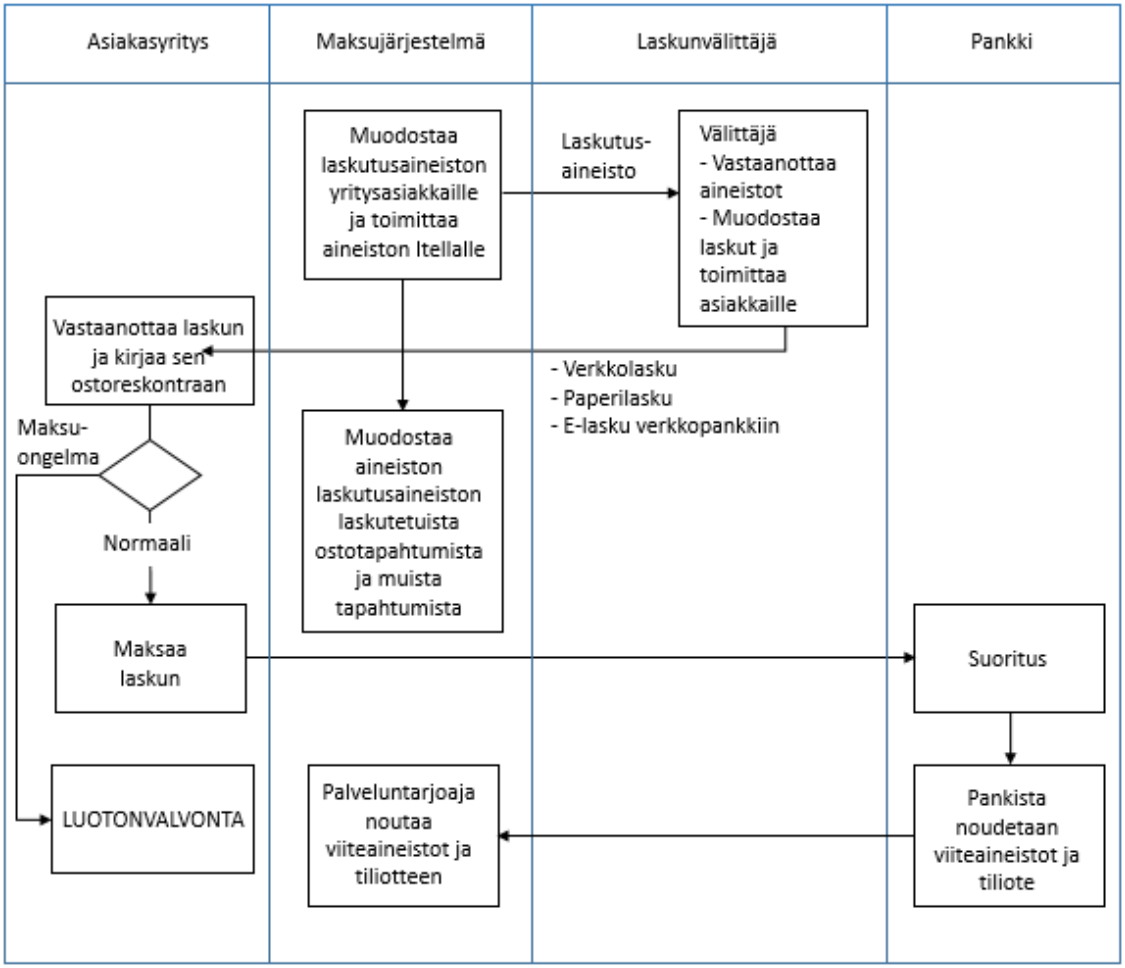
Asiakas voi tehdä maksusuorituksia koska tahansa millä tahansa summalla, sillä asiakkaalla on kullekin tilille vakioviite. Asiakkaiden suoritukset tuodaan maksujärjestelmän tietokantaan viiteaineiston ja tiliotteen kautta. Tämä aineisto saadaan pankilta päivittäin. Aineiston lukemisen jälkeen maksujärjestelmä allokoi kunkin suorituksen automaattisesti tilin sisällä, ensin erääntyneille laskuille, sitten mahdolliseen maksusuunnitelmaan kuuluville ja lopuksi muille. Laskun sisällä allokoidaan ensin koroille, sitten palkkioille ja viimeiseksi pääomalle. Jos allokoinnin jälkeen jää rahaa käyttämättä, se merkitään ylisuoritukseksi, joka nostaa tilin käyttövaraa. Vaikkei ylisuoritusta syntyisikään, avointen laskujen maksaminen luonnollisesti nostaa tilin käyttövaraa. Jos allokointi ei toimi ja pääomaa jää maksamatta, siitä alkaa kertyä korkoa ja tilin käyttövara ei nouse. Toisaalta, jos allokointi toimisi väärässä järjestyksessä, eivät korot kertyisi oikein, sillä korkoa lasketaan vain laskun avoimesta pääomasta.

Suoritus voidaan sisäisen käyttöliittymän kautta uudelleenallokoida, mikäli jollakin laskulla on esimerkiksi aiheettomia viivästyskorkoja tai palkkioita. Uudelleen jyvityksessä poistetaan aiheettomat viivästyskorot tai palkkiot eli käytännössä tehdään samansuuruiset negatiiviset rivit. Vapautunut summa kohdistetaan uudelleen maksamattomille laskuille tai siitä voidaan tehdä rahanpalautus asiakkaalle. Sisäisessä käyttöliittymässä näkyvät myös viiteaineiston suoritukset, jos viitenumeroa vastaavaa tiliä ei löydy. Sieltä niitä voi viitenumeron selvitessä siirtää muualle ja käyttää vaikka laskun maksuun.

4.3.3 Laskujen muodostus

Laskutusprosessia esittää Kuva 4.4. Laskutuksessa kullekin tilille muodostetaan lasku, sille laskutuskauden aikana kohdistuneista tapahtumista automaattisesti. Korot lasketaan avoimille laskuille ennen laskutusta. Korkoa lasketaan laskun avoimelle pääomalle. Jos asiakas on maksanut laskunsa osissa ja myöhässä, yhteen laskuun liittyvä korko voi koostua useasta osasta, joita on laskettu eri pääomille. Laskutusaineisto toimitetaan laskunvälittäjälle määrämuotoisena, ja se toimittaa laskut eteenpäin tilien omistajille.

Virheet ovat erityisen ikäviä laskutusaineistoa muodostettaessa, jos laskun loppusummasta tulee liian pieni tai suuri. Laskutus huolehtii myös mahdollisten ylisuoritusten käyttämisestä laskun loppusumman vähentämiseen. Lasku on tilikohtainen. Laskuihin liittyvät myös maksumuistutukset, joita lähetetään automaattisesti laskun erääntymisen jälkeen.



Kuva 4.4: Laskutusprosessi.

4.3.4 Tietoturva

Tietoturva on luonnollisesti järjestelmän kannalta kriittinen osa. Toteuttajayritys teettää tietoturva-auditoinnin itselleen ja kehitettävälle ohjelmistolle. Ohjelman tietoturva-auditoinnissa tarkastetaan sille asetetut ei-toiminnalliset tietoturvavaatimukset.

Järjestelmän vaatimuksissa todetaan, että:

- Järjestelmän ulkoiset rajapinnat tulee olla suojattu ainakin yleisimmiltä haavoittuvuuksilta, tarkoittaen OWASP Top-10 -listaa [40].
- Järjestelmään tehtyjä pyyntöjä ja niiden tietoja tallennetaan, jotta väärinkäytöksestä jäisi jälki.
- Käyttöoikeuksien tulee olla erilliset järjestelmän eri osille.
- Käyttäjätunnukset pitää suojata turvallisella yksisuuntaisella suojauksella, esimerkkinä bcrypt. Ne pitää myös vaihtaa tietyin väliajoin, ja salasanan turvallisuudelle sekä vaihtotiheydelle on minimivaatimukset.

Tietoturvaan on otettu kantaa arkkitehtuurissa esimerkiksi käyttämällä erillistä välittäjäpalvelinta, jonka kautta kaikki yhteydet eri palvelimille kulkevat. Näin ulkoverkossa ei ole suoraan kiinni se palvelin, jolla esimerkiksi tietokanta on säilössä.

4.3.5 Sisäinen käyttöliittymä

Toteutettavaan järjestelmään tulee sisäinen hallintakäyttöliittymä, joka toimii verkon yli. Web-käyttöliittymän kautta voi lisätä uuden asiakkaan maksujärjestelmän piiriin ja lisätä hänelle tilejä. Käyttöliittymän kautta voi myös muokata nykyisten asiakkaiden tietoja ja sulkea tilejä.

Sisäinen käyttöliittymä mahdollistaa myös erilaiset korjaustoimet. Kaikki ilman tunnettua viitettä tulleet maksut listataan käyttöliittymässä, jossa niitä voidaan siirtää tileille, jos saadaan selville, minne ne oikeasti kuuluvat. Käyttöliittymässä voidaan muutenkin luoda maksuja, esimerkiksi palautuksia, jos asiakkaalta on vaikkapa peritty liikaa korkoa. Nämä maksut voidaan joko käyttää auki oleviin laskuihin valikoiden kohteet käyttöliittymässä tai palauttaa asiakkaan tilille. Käyttöliittymän kautta voi myös kirjata palvelumaksuja.

Käyttöliittymän tekeminen aloitettiin heti ensimmäisten asioiden joukossa. Käyttöliittymä on välttämätön järjestelmän hallinnoinnissa, mutta suurin osa siitä ei kuitenkaan koske suoraan rahaa. Aikaisella aloittamisella voidaan varmistaa se, että käyttöliittymän rahaa käsittelevät osat ehditään tehdä ja testata huolella. Käyttöliittymän monimutkaisuutta kasvattavat asiakkaan luontiin liittyvät monet rajoitteet ja tarkistukset.

4.3.6 Ulkoinen käyttöliittymä

Ulkoinen käyttöliittymä on loppuasiakkaille tarkoitettu web-käyttöliittymä, jonka kautta käyttäjät eivät voi tehdä muutoksia vaan ainoastaan tarkastella tietoja. Palvelulla voi olla useita käyttäjiä tiliä kohden, ja yhden käyttäjän näkemä tieto voidaan rajata vaikkapa hierarkian kahteen eri tiliin. Käyttäjä näkee paitsi tilit, joihin hänelle on määritelty oikeus, myös näiden alta hierarkiasta löytyvät muut tilit ja niiden tapahtumat. Tieto, jonka käyttäjä näkee, sisältää hänestä järjestelmässä kirjattuna olevat perustiedot, sekä hänen tileillään tehdyt ostot sitä mukaa, kun ne kirjautuvat tietokantaan. Palvelussa voi myös hakea Excel-raportteja järjestelmän sisältämistä tiedoista.

Ulkoista käyttöliittymää alettiin tehdä vain muutama kuukausi ennen järjestelmän sovittua valmistumispäivämäärää. Sen ei tarvinnut olla valmiina kuitenkaan aivan tähän päivään mennessä: hieman vajavainenkin versio katsottiin riittäväksi järjestelmän myyntiin, kunhan se päivitettäisiin nopeasti.

4.3.7 Raportointi

Maksujärjestelmään kuuluu myös raportointia. Raportit voidaan ajaa joko komentoriviltä tai hakea sisäisellä käyttöliittymällä. Erilaisia raportteja on kymmenkunta, ja ne perustuvat lähinnä SQL-näkymiin, joista tieto haetaan Excel-taulukkoon ja käyttöliittymässä luonnollisesti myös käyttöliittymään. Yhdenlaisena raportointina voi pitää myös asiakkaille tarjottua rajapintaa, jolla voi hakea maksu- ja laskutusaineiston.

Raportit toimivat liiketoimintapäätösten tukena ja antavat tietoa järjestelmän tilasta. Jos raportointi ei toimi, voidaan joitain päätöksiä tehdä virheellisin perustein tai jokin virhe voi jäädä huomaamatta.

4.3.8 Luotonvalvonta

Järjestelmä valvoo automaattisesti erääntyneitä laskuja. Kun laskun maksu viivästyy tarpeeksi, lähetetään ensimmäinen maksumuistutus. Tämän jälkeen asiakkaan tilit laitetaan ostoskieltoon. Sitten lähetetään toinen maksumuistutus, ja jos maksu edelleen viipyy, irtisanotaan asiakkaan tili. Jos laskua ei vieläkään makseta, se siirtyy perintään, jolloin se ei enää ole maksujärjestelmän piirissä.

Luotonvalvonnan toimiminen on tärkeää, jotta mahdolliset maksuongelmat jäävät kiinni tuoreeltaan. Jos järjestelmä ei toimi, mitään tietoa ei kuitenkaan menetetä. Näin ollen, jos maksujen viivästyminen huomataan esimerkiksi asiakkaan tukipyynnön yhteydessä, voidaan ne edelleen karhuta. Maksamattomuus ei myöskään ole yleistä, joten riski on pieni.

4.3.9 Kirjanpitoaineistot

Koska maksujärjestelmä toimii rahan kanssa, on myös kirjanpito tärkeä osa järjestelmää. Järjestelmä tuottaakin ostajan kirjanpitoon useita erilaisia aineistoja.

Asiakasaineisto kuvaa asiakashierarkian, joka koostuu asiakkaista ja heidän tileistään. Kirjanpitojärjestelmä- ja laskutusaineistot ovat asiakasaineiston kaltaisia määrämuotoisia aineistoja kirjanpitoa varten. Kirjanpitojärjestelmään viedään aineisto kerran päivässä. Kirjanpitoaineistoon kuuluvat ostotapahtumat, korot, palkkiot ja niiden korjaukset. Laskutusaineisto sisältää loppuasiakaskohtaiset laskutetut ostotapahtumat. Laskutusaineiston ja laskujen tulee täsmätä keskenään.

Erilaiset kirjanpitoaineistot ovat tärkeitä, mutta jos järjestelmässä on olemassa oikea tieto, voidaan väärin luotu aineisto tehdä aina uudelleen. Uuden aineiston saaminen hyväksytyksi ja vanhan mitätöiminen on kuitenkin hankalaa. Yleensä käytäntö on, että mitään ei poisteta vaan korjaaminen tapahtuu tuottamalla vastaavia korjaustapahtumia, mihin järjestelmä ei automaattisesti pysty.

4.4 Toteutustapaan liittyvät vaatimukset

Määrittelyn lisäksi järjestelmään liittyy muita vaatimuksia. Ei-toiminnalliset vaatimukset liittyvät vasteaikaan ja saatavuuteen. Lisäksi tärkeää on projektin aikataulu, ja järjestelmän jatkokehitettävyys.

Vasteajat ja saatavuus. Järjestelmälle on määrittelyn lisäksi luotu ei-toiminnalliset vaatimukset, joissa on määritelty vasteajat käyttöliittymälle ja tilin osto-oikeuden tarkistamiselle. Järjestelmälle on myös määritelty huippukuorma, johon asti se skaalautuu ja jonka aikana vasteajoista luvataan edelleen pitää kiinni. Saatavuusvaatimukset on määritelty suunniteltujen katkojen ulkopuolella ulkoiselle ja sisäiselle käyttöliittymälle sekä tilin osto-oikeuden tarkistukselle erikseen. Ulkoisen ja sisäisen käyttöliittymän suhteen saatavuusvaatimukset eivät vaatineet erityistoimenpiteitä, mutta osto-oikeuden tarkistuksen kohdalla vaadittiin jo vaivannäköä saatavuusvaatimusten täyttymisen varmistamiseksi.

Aikataulu. Projektissa sekä vaatimukset että resursointi joustavat selvästi aikataulutavoitetta enemmän. Projektin etenemisellä on joitakin sovittuja tarkastuspisteitä: kymmenen viikkoa ennen käyttöönottoa alkaa rauhoitusaika, jonka aikana ei ole tarkoitus tehdä uusia ominaisuuksia vaan vain testata, ja korjata löydettyjä virheitä. Tietoturva-auditointi ohjelmistolle tehdään kuukautta ennen sen valmistumista. Julkaisusuunnitelman ensimmäiseen aaltoon kuuluvat tärkeimmät ominaisuudet eli maksuaineiston käsittelyyn ja kirjanpitoon liittyvät asiat, sekä tiettyjen kantatapahtumien seuranta. Julkaisusuunnitelman toiseen osaan, joka oli versio 1.0 eli ensimmäinen käyttöönottokelpoinen versio, kuuluivat pitkälti kaikki loput ominaisuudet. Osa laskutuksen ominaisuuksista, ulkoisesta käyttöliittymästä sekä yhteys laskunvälittäjän rajapintaan saivat vielä jäädä toteuttamatta. Versiossa 1.1 julkaistaan korjauksia, erilaisia raportteja sekä viimeistelty versio laskutuksesta. Valmis tämän julkaisun on tarkoitus olla kuukausi version 1.0 jälkeen. Seuraavan julkaisun on tarkoitus olla valmis kaksi kuukautta version 1.1 jälkeen.

Jatkokehitettävyys. Tiukan aikataulun takia maksujärjestelmän ominaisuuksia on priorisoitu. Järjestelmään jää siis jatkokehitettävää, vaikka sen perustoiminnallisuus onkin valmis. Erityisesti loppuasiakkaille tarjottaviin palveluihin on helppo ideoida jatkokehitystä. Myös sisäiseen käyttöliittymään voidaan tehdä muokkaustyötä esiin nousevien uusien käyttötarpeiden mukaan. Ohjelmakoodin ja sen rakenteen selkeydelle on siis selvä tilaus, jotta jatkokehitys on mahdollista.

5. PROJEKTIN OHJEISTUS

Maksujärjestelmäprojekti tehtiin tiukalla aikataululla. Tästä syystä projektiryhmä oli mahdollisimman suuri ja siinä oli myös jonkin verran vaihtuvuutta. Ihmisiä otettiin projektiin mukaan sitä mukaa, kun heitä vapautui muista projekteista tai heitä rekrytoitiin ja toisaalta kun saatiin esiehdot valmiiksi uuden kokonaisuuden aloittamiselle. Ihmiset tulivat projektiin erilaisista taustoista, ja tekivät uusia ohjelman osia toistensa tekemien osien varaan. Lisäksi ohjelma käsittelee rahaa: sen ongelmat näkyvät asiakkaille ja sitä tullaan luultavasti vahvasti jatkokehittämään, joten sen laatu haluttiin varmistaa.

Maksujärjestelmäprojektiin tehtiin ohjeistus, jotta se valmistuisi ajallaan ja laatu pysyisi korkeana. Ohjeistukseen valittiin projektin tilanteeseen mahdollisimman hyvin sopivia laadunvarmistuskäytäntöjä. Ohjeiden tekeminen aloitettiin ajallisesti projektin puolivälin jälkeen. Oletus kuitenkin oli, että projektiryhmän kasvun takia edessä on enemmän työtunteja kuin takana.

Käytännöt valittiin ja ohjeet muotoiltiin ottaen huomioon projektin haasteet ja ohjeiden sijoittuminen projektin aikajanalle. Tässä luvussa esitellään ohjeistuksen toteutus eli valitut käytännöt, muodostetut ohjeet sekä niiden suunniteltu käyttöönottotapa.

5.1 Valitut käytännöt

Ohjeisiin valittiin hyviä käytäntöjä ja mieluiten sellaisia, jotka olivat jo kokonaan tai osittain projektiryhmän käytössä. Tutkimuksessa [41] ehdotettiin kokemusten pohjalta Scrumin käyttöönottoon käytettäväksi ajaksi puolta vuotta, jolloin ei olisi kiireisiä projekteja. Scrum oli onneksi tuttu projektiryhmälle, joten siihen tutustuminen ei vaatinut aikaa.

Haikala esitti että laatujärjestelmähankkeet usein hiipuvat siihen, että yritetään muuttaa liian paljon asioita kerralla [1, s. 186]. Käyttöönottoon liittyvien tutkimusten pohjalta ohjeistuksen ei haluttu olevan täysin mullistavaa.

5.1.1 Scrum

Scrum on yksi valituista käytännöistä. Sen tavoite on kehittää kommunikaatiota asiakkaan ja tiimin välillä sekä tiimin kesken. Projekti oli jo aloitettu scrumin riisutulla versiolla, ja sen katsottiin täyttävän nämä tavoitteet. Scrumin käyttöä päätettiin siten jatkaa samalla tavalla kuin sitä oli projektissa siihen asti käytetty.

Projektin koko pysyi pitkään scrumin sallimassa tiimikoossa, mutta vuodenvaihteen jälkeen se ylitettiin. Pahimmillaan projektissa oli yhtä aikaa yli kymmenen kehittäjää. Luvussa ei ole huomioitu erillistä testaustiimiä. Tiimin jakamista harkittiin vähän projektin

puolivälin jälkeen niin, että käyttöliittymän tekijät olisivat muodostaneet oman muutaman hengen tiiminsä. Kuitenkin kehittäjät kokivat tekevänsä läheistä yhteistyötä ja kuulevansa mielellään myös käyttöliittymään liittyvät asiat, joten ajatuksesta luovuttiin.

Eroina puhdasoppisen scrumin ja käytössä olevan version välillä oli tiimikoon lisäksi se, että tehtäviä ei pisteytetty vaativuuden mukaan ja käytetty tätä hyväksi sen valitsemisessa, mitä tehtäviä ehditään tehdä sprintin aikana. Tehtävien suunnittelu ja kirjoittaminen olivat myös vahvasti pelkästään kehittäjän harteilla eikä niille ollut selkeää valmiin määritelmää. Tehtävien toimintaa ei myöskään esitelty sprintin lopputapaamisessa, joten piti vain luottaa kehittäjän sanaan, kun sanottiin, että tehtävä on valmis. Lisäksi päivittäiset tapaamiset olivat periaatteessa kiinteän mittaisia, mutta käytännössä ne venyivät. Scrumiin liittyy myös ajatus tasaisesta työtahdista. Tämä piti suurimman osan aikaa paikkansa, mutta ajoittain jotkut tiimin jäsenet tekivät esimerkiksi viikonlopputöitä ja projektipäällikkö teki lähes koko ajan ylitöitä. Tähän ei juurikaan otettu kantaa.

Projektissa käytetty scrumin versio oli kaikille jo ohjeistusta tehtäessä tuttu. Scrumin käyttö myös painottui tietyllä tavalla tapaamisiin eli se tuli väistämättä myös uusille projektiryhmäläisille esiin. Näistä syistä siihen liittyviä käytäntöjä ei dokumentoitu erikseen. Nähtiin, että ohjeistukseen ei ole syytä ottaa asioita, jotka ovat jo kaikilla hallussa. Scrumin katsottiin myös jossain määrin varmistavan määrittelyn laatua ja ymmärtämistä, joten näihin asioihin ei yritetty kehittää lisäohjeita.

5.1.2 Pariohjelmointi

Pariohjelmointi päätettiin ottaa yhdeksi suositeltavista käytännöistä. Aika- ja resursointiongelmien takia pariohjelmointia ei kuitenkaan voi käyttää kaikkeen. Kerätyn tiedon perusteella on nähtävissä, että yksi pari tekee vähemmän työtä kuin kaksi yksittäistä ihmistä samassa ajassa. [8, 15] Jos projekti olisi ollut vähemmän aikakriittinen, olisi pariohjelmointia ehdotettu käytettäväksi ainakin tärkeimpien toimintojen ydinosissa, jolloin laatu olisi ollut varmemmin hyvä. Toisaalta ohjeiden käyttöönottovaiheessa monet ydinosat oli jo tehty.

Projektin loppuvaiheessa on odotettavissa tilanne, jossa henkilöresursseja joudutaan vähentämään, koska tekeminen alkaa loppua. Tällaisessa tilanteessa toimettomaksi jääneitä henkilöitä voitaisiin ottaa muiden kehittäjien pareiksi, ja näin viimeiset tärkeät ominaisuudet saataisiin tehtyä mahdollisimman laadukkaasti ja toisaalta hieman lyhyemmässä ajassa kuin mitä yksittäinen ohjelmoija tekisi. Erityisesti loppuvaiheeseen jääneissä kriittisissä kohdissa on hyvä, että pari ohjelmoi nopeammin ja vähemmän virhealttiisti. Tätä voisi myös hyödyntää vaikeiden ja kriittisten virheiden ratkomisessa. Tietysti, jos ohjelman virheellinen osa on parin toisen jäsenen kirjoittama ja aivan outo pariksi tarjolla olevalle, ei parittaminen silloin kannata

Pariohjelmoinnin käyttöönottoon tosin kuluu varmasti hieman aikaa, ja se voi olla kalliimpaa, joten aivan lyhyen aikavälin tai ei-niin-kriittisen ohjelman osan takia sitä ei kannatta hyödyntää. Jos rekrytointi olisi helppoa tai siihen olisi käyttää aikaa, olisi tiimin kokoa voinut pariohjelmoinnin avulla vielä kasvattaa ilman, että kommunikoinnista tai työnjaosta olisi tullut liian iso ongelma.

Pariohjelmoinnin kustannukset ovat suuremmat kuin yksittäin ohjelmoinnin, mutta koska projektissa aikataulusta kiinni pitäminen on kustannuksia tärkeämpää, pariohjelmoinnin kustannusten ei pitäisi olla kynnyskysymys käytölle, varsinkin, kun sitä on mahdollisuus käyttää vain rajallisesti.

5.1.3 Katselmoinnit

Katselmointeihin liittyen käytössä oli Gitin Pull Request (PR) -käytäntö Visual Studio Onlinessa (VSO) graafisesti esitettynä. Tämä tarkoitti käytännössä sitä, että aina kun kehittäjä halusi yhdistää koodiaan päähaaraan, hän teki PR:n. Käsittelemättömät PR:t näkyivät VSO:n web-käyttöliittymässä muille kehittäjille. Ennen PR:n yhdistämistä vähintään yhden kehittäjän täytyi antaa hyväksyntänsä PR:lle. Yhdistämisen esti myös se, jos uuden koodin päähaaraan yhdistämisestä olisi seurannut konflikteja. Tästä tehtiin vielä oma sääntö ohjeisiin, koska asia koettiin tärkeäksi paitsi koodin selkeyden myös päähaaran muutosten seuraamisen kannalta.

Katselmoimista PR:ien lukemisesta teki paitsi se, että joku luki koodin läpi eikä lukijoita ollut mitenkään rajoitettu yhteen, myös mahdollisuus kommentoida kaikkea, mikä PR:ssä oli pielessä. Kommentoinnin jälkeen kehittäjän piti vastata kommentteihin tai korjata koodinsa niiden perusteella ennen hyväksynnän saamista. Jotta PR:t jaksettaisiin käydä huolella läpi, ohjeella muistutettiin vielä pitämään PR:t lyhyinä. Tämä tukee myös jatkuvaa integrointia. Katselmoinnilla tarkoitus on löytää virheitä ja saada kehittäjien tietoisuus ohjelman sisällöstä kasvamaan. Tähän liittyen ohjeissa vielä erikseen mainittiin, että PR:n tekemisestä kannattaa kertoa niille, jotka luultavasti tulevat käyttämään PR:n sisältöä tai jotka muuten ovat kiinnostuneita juuri tämän PR:n sisällöstä.

Lisäksi PR:iä käytettiin keinona pyytää mielipiteitä omasta koodista jo ennen kuin se oli tarkoitus yhdistää päähaaraan. VSO mahdollisti koodin vaivattoman vertailun nykyiseen sekä kommentoinnin. Näin PR:llä saatiin jo aikaisessa vaiheessa palautetta, jos kehittäjä oli esimerkiksi epävarma jostain suunnitteluratkaisustaan. Tässä mielessä PR-käytäntö ajaa samaa asiaa kuin pariohjelmointi: sillä on saatavissa toinen mielipide omaan työhön.

Lisäksi katselmointiin liittyi ohjeiden muistilista. Sitä oli tarkoitus käyttää niin oman koodin itsenäiseen tarkistamiseen kuin avuksi muiden PR:iä lukiessa. Tästä syystä se haluttiin pitää tiiviinä ja helposti käytettävänä.

5.1.4 Jatkuva integrointi

Tiimissä oli yksi henkilö, joka oli vastuussa jatkuvan integroinnin työkalun ylläpidosta. Koodivarastoa seuraava, muutoksista käännöksen ja sille automaattiset testit ajava jatkuvan integroinnin palvelin käänsi ja ajoi testit päähaaran lisäksi myös kehittäjien omille ominaisuushaaroille.

Jatkuva integrointi oli jo ohjeita tehdessä käytössä, ja siinä ei ollut erityisiä ongelmia. Näin ollen ainoa tapa, jolla asiaan otettiin ohjeissa kantaa, oli kehotus varmistaa, että oman haaran lähdekoodi kääntyy, ennen kuin sen yhdistää päähaaraan. Tähän ohjeeseen liittyy ajatus siitä, että koodin kääntyminen jatkuvan integroinnin palvelimella tarkoittaa, että myös testit on ajettu onnistuneesti. Paitsi kääntämisen myös testiajon epäonnistuminen aiheutti haaran merkitsemisen rikkonaiseksi. Päähaaran korjaaminen mahdollisimman nopeasti rikkoutumisen jälkeen oli tiimille sydämen asia, joten vaikka se on tärkeää, siitä ei tehty erillistä ohjetta.

Ihan äärimmäisyyksiin ei integroinnin taajuuden kanssa menty. Jopa päivittäistä integrointia päähaaraan suositellaan [42], mutta tämä yhdessä katselmointeihin liittyvän PR-käytännön kanssa olisi voinut helposti luoda ongelmia. Lyhyt PR on helppo katsoa läpi ja kommentoida. Mutta jos PR:iä on päivän lopussa 15 ja ainakin jonkun pitää kommentoida niitä ja jos jokaisen pitää reagoida oman PR:nsä kommentteihin ja lopulta vielä hyväksyä kommenttien mukaan korjatut PR:t, alkaa menetelmä olla turhan raskas. Toisaalta, koska kukaan ei jaksa lukea pitkää PR:ää, käytäntö johtaa siihen, että koodia integroidaan päähaaraan kuitenkin suhteellisen usein.

5.1.5 Testaus

Testaukseen otetaan ohjeissa sen verran kantaa, että jos joutuu käyttämään muuta kuin foreach-silmukkaa, pitää testata silmukan toimivuus raja-arvoilla. Tämän lisäksi kehotetaan tekemään testejä ohjelmakoodille jo aikaisessa vaiheessa ennen integrointia päähaaraan. Tämä auttaa myös jatkuvan integroinnin ajaman testisetin pitämisessä luotettavana. Test driven development -ohjelmointitapa päätettiin jättää ottamatta käyttöön, koska ohjeiden tekoon mennessä testikattavuus oli koettu riittäväksi takaamaan laadun ja uusien tapojen omaksuminen vaatii kuitenkin aina jonkin verran työtä.

Käyttäjäkokemukseen tai käytettävyyteen ei otettu projektissa erikseen kantaa. Sisäisestä käyttöliittymästä teetettiin kuitenkin käyttäjätutkimus muutamaa kuukautta ennen projektin valmistumista, ja siinä esiin nousseita asioita otettiin kehityksessä huomioon. Näin vähennetään inhimillisten virheiden mahdollisuutta rahankäsittelyssä. Käyttöönottokoulutuksella ja ajoissa saadulla palautteella voidaan varmistaa, että järjestelmää käytetään tarkoitetulla tavalla eikä sinne jää pahasti harhaanjohtavia osioita.

Testaukseen ei ohjeissa tarkoituksella oteta yksikkötestejä laajemmin kantaa. Projektiryhmässä oli osa-aikaisena mukana yksi henkilö, joka testasi ohjelmaa määrittelyä vastaan. Lisäksi oli erillinen testaustiimi tekemässä sekä suunniteltua järjestelmätestausta että hieman tutkivaa testausta. Projektiryhmän testaaja piti aktiivisesti yhteyttä testaustiimiin. Yhteisenä työvälineenä virheiden raportoinnissa toimi Visual Studio Online ja siellä virhe-luokan tehtävät. Virheet merkittiin ensin tehtäviksi projektipäällikölle, joka siirsi ne eteenpäin kehittäjille, jotka olivat tehneet sen osan ohjelmasta, jota virhe koski. Virheet siirrettiin määrättyyn tilaan, kun niihin oli ratkaisu olemassa. Tällöin virhe myös siirrettiin testaajalle, joka oli sen tehnyt. Virheeseen laitettiin tiedoksi, milloin korjaus on testattavissa. Jos korjaus ei toiminut, tuli virhe takaisin kehittäjälle. Ja toisaalta, jos virhe ei ollut selkeä, se laitettiin suoraan takaisin testaajalle lisätietopyynnön kanssa. Virheiden kanssa toimimiseen oli siis oma tapansa, joka mietittiin yhdessä. Sen kanssa ei ollut ongelmia, joten sitä ei erikseen dokumentoitu.

5.1.6 Kommentointi

Kommentointiin ei juurikaan otettu ohjeissa kantaa. Sen verran, että esitettiin, että tehdään mahdollisimman hyvin itsensä kommentoivaa koodia eli selkeän nimisiä muuttujia, funktioita ja luokkia. Tämän seurauksena kommenteissa voidaan selittää sitä miksi asioita tehdään sen sijaan, että kuvailtaisiin mitä koodi tekee.

Lisäksi käytössä oli tapa laittaa kunkin funktion ja luokan yhteyteen kommentti, joka kertoo sen tarkoituksen. Visual Studio tukee tällaisten kommenttien luomista, ja myöhemmin kommentoitua luokkaa käyttäessään käyttäjä näkee kommentin sisällön. Kommentissa kuvataan lyhyesti toiminta ja funktiosta myös parametrit ja paluuarvo. Tätä tapaa ei dokumentoitu erikseen, sillä se oli hyvin vahvasti käytössä ja ohjeista ei kuitenkaan haluttu turhan pitkiä. Kommenttien puuttuminen on myös helppo huomata, joten uudelle kehittäjälle asia tulee viimeistään oman ensimmäisen PR:n yhteydessä ilmi, kun muut huomauttavat asiasta.

5.1.7 Refaktorointi

Refaktorointi oli vahvasti käytössä tilien osto-oikeuden tarkastusta tehdessä. Ensin tehtiin ensimmäinen versio, joka jo toimi kohtuullisesti. Sitten tehtiin seuraava, elegantimpi versio ja sitten nopeampi versio. Ohjeita tehtäessä näytti siltä, että tarkastuksia tullaan edelleen refaktoroimaan. Samoin käyttöliittymässä osa taustan tekniikasta vaihdetaan refaktoroiden.

Nämä ovat kuitenkin yksittäistapauksia, ja tekijät tietävät mitä tekevät. Erillisiä ohjeita asiaan liittyen ei siten nähty tarpeellisiksi.

5.2 Muodostettu ohjeistus

Ohjeet ovat liitteenä A. Ne koostuvat esipuheesta, jossa kehotetaan kehittämään ohjeita, motivoidaan ohjeiden noudattamiseen ja kerrotaan mihin ohjeet ottavat kantaa. Tämän jälkeen seuraa 22 kohdan muistilista, jossa on ohjeita sekä ohjelmoinnin yksityiskohtiin että laajemmin ohjelman tuottamiseen liittyviä lyhyitä ohjeita. Lisäksi ohjeisiin on liitetty muutama tietoturvaan liittyvä linkki. Lopuksi esitellään pariohjelmointia ja sen mahdollista hyödyntämistä projektissa, ja aivan viimeisenä ovat lähteet.

Ohjeiden kieli oli tarkoituksella lähellä puhekieltä. Tarkoitus on madaltaa ohjeiden muokkaamisen kynnystä. Sen sijaan, että painotettaisiin korrektia kieltä, jossa ei ole anglismeja tai ammattislangia, tärkeää on saada omat hyvät ideat helposti mukaan. Ohjeita lukivat myös projektiryhmän suomenkielestä kiinnostuneet jäsenet, jotka tekivät osansa ohjeiden kehittämiseksi osin myös selventämällä ja korjaamalla kieltä.

Pariohjelmointi on osana yleisiä ohjeita, vaikka se on relevantti lähinnä projektipäällikölle, jotta hän osaa esittää sen käyttöä, ja parille, kun he alkavat käyttää ohjelmointitapaa. Kuitenkin tiimi on täynnä ammattilaisia, joilla voi olla hetkittäin projektipäällikköä parempi näkyvyys tilanteisiin ja ohjeen lukemisen kautta motivaatio harkita pariohjelmointia. Pariohjelmointi oli myös herättelemässä ihmisiä ajattelemaan koodauskäytäntöjä laajemmin kuin pelkkänä muistilistana. Näin olisi helpompi tunnistaa käytössä olevia laajempia tapoja, joita voisi lisätä ohjeisiin.

Tietoturvalinkit ovat ohjeissa, koska tietoturvan ja ohjeiden suhdetta pohtiessa käytiin jonkin verran tietoturvamateriaalia läpi. Ensimmäinen linkki on käytettyyn tekniikkaan liittyvä, ja sen ajatuksena on, että kaikkien jotenkin kirjautumisen tai sisällön rajaamisen kanssa tekemisissä olevien olisi hyvä vilkaista se läpi, jotta tunnistavat tilanteet, joissa on syytä kiinnittää erityistä huomiota tietoturvaan. Toinen linkki on aivan tietoturvan perusteita, ja ajatuksena on tarjota perustietoa kokemattomimmille projektiryhmän jäsenille. Kolmas linkki on syventävää tietoa, jos joku kiinnostuu aiheesta.

Ohjeista suurin osa on muistilistan muodossa, koska se on nopea lukea ja siihen on helppo lisätä asioita. Ajatuksena oli, että ohjeiden on hyvä olla helppokäyttöisiä, jotta niitä tulee käytettyä ja laajennettua. Jotta muistilista pysyi tarpeeksi lyhyenä, joitain vahvasti käytössä olleita käytäntöjä myös jätettiin sen ulkopuolelle. Pidemmät perustelut muistilistan kohdille voi antaa linkkeinä tai kertoa sprintin lopetuspalaverissa. Muistilista sai vahvasti vaikutteita NASAn kymmenen kohdan muistilistasta [27]. Osa kohdista oli niin vahvasti käytettyyn ohjelmointikieleen liittyviä, etteivät ne sen takia päässeet listalle. Monia NASAlta lainattuja kohtia on lievennetty muistilistaan, sillä toisin kuin NASAn ohjelmistot tai muut turvallisuuskriittiset ohjelmistot, tämä ohjelmisto ei voi pahimmillaankaan aiheuttaa esimerkiksi ihmishenkien menetyksiä. Tällöin arvioitiin yhtä tiukkojen ohjeiden hidastavan kehitystä enemmän kuin tuovan lisäarvoa. Liian tiukoiksi mielletyt ohjeet voi-

vat myös johtaa niiden noudattamiseen vain nimellisesti, jolloin koodia selventämään tarkoitetusta ohjeesta olisikin seurauksena ajattelemattomasti ohjeiden mukaan muokattua koodia. Toinen ajatus lieventämisen takana on se, että miettimiskehotus saa enemmän aikaan kuin käsky, joka tuntuu siltä, ettei se varmaankaan päde juuri tekeillä olevaan koodiin. Seuraavaksi listataan kukin muistilistan kohta perusteineen.

1. *”Varo pyöristysvirheitä. Decimal-tyyppi toimii C#-koodissa ja kannassa, JavaScriptissä kokonaisluvut.”* Ohje on rahan käsittelyssä tärkeään pyöristämiseen ja tarkkuuteen liittyvä. Käyttöliittymässä ei käsitellä senttejä pienempiä rahan osia, joten ne voidaan esittää kokonaislukuina, ja muuten käytettiin C#:n decimal-tyyppiä. Osa summista tulee kolmen desimaalin tarkkuudella, mutta laskujen loppusummat ovat aina kahden desimaalin tarkkuudella. Decimal-tyypillä laskiessa pyöristys tapahtuu puolikkaaseen asti alaspäin ja sen jälkeen ylöspäin. Nämä numerointitavat olivat jo käytössä ennen ohjeistuksen luomista, mutta koska ne ovat erityisesti rahan kanssa tärkeitä, ne kirjattiin ylös.
2. *”Kommentoi koodia. Koska kaikki kirjoittavat selkeää ja ymmärrettävää koodia, voi kommenteissa keskittyä siihen miksi, ei niinkään miten.”* Koodin kommentointiin otettiin kantaa kehottamalla siihen, mutta niin, että koodi on mahdollisimman selkeää jo ilman kommentteja ja että kommentit kertovat siitä, miksi asioita tehdään. Jos jokainen rivi on kommentoitu, jokaista riviä muuttaessa pitää muuttaa kommentti. Jos niitä ei muuta, kommentit ja koodi ovat ristiriidassa, mikä häiritsee ymmärtämistä ja ohjelmointia.
3. *”Älä käytä rekursiota. Kyse on koodin selkeydestä, ja sitä kautta bugittomuudesta. Jos rekursion kiertäminen tuntuu vaikealta, kysy neuvoa. Jos vaihtoehtoinen ratkaisu on edelleen kovin monimutkainen, käytä rekursiota.”* Rekursiota kehotettiin välttämään siihen liittyvien vaikealukuisuusriskien takia. Mutta rekursion välttäminen ei ole itseisarvo, joten jos hyvää tapaa kiertää ei löydetä, käytetään rekursiota. Tämä ohje perustuu NASAn muistilistan 1. ohjeeseen [27].
4. *”Käytä mahdollisuuksien mukaan foreach-silmukkaa. Jos se ei sovi, testaa silmukan rajat.”* Ohi-indeksointi (off-indexing) on yksi yleisimpiä ohjelmointivirheitä. C# tarjoaa foreach-silmukan, jolla voi välttää mahdollisuuden mennä esimerkiksi taulukon pituuden yli. Tätä silmukkaa tulee suosia, ja jos sitä ei ole mahdollisuus käyttää, pitää silmukan rajat testata. Ohje on johdettu NASAn ohjeesta numero 2, jossa itse asiassa vaaditaan, että silmukoiden voidaan varmistaa pysyvän niiden ylärajan alla [27].
5. *”Jos funktio tai proseduuri on yli 80 riviä pitkä, käytä 5 min aikaa sen miettimiseen, eikö sitä saisi jotenkin selkeästi jaettua. Mieti samaa myös pitkien luokkien kohdalla.”* Lyhyet funktiot olivat myös NASAn listalla, ohjeena 4 [27]. Ohje lieventyi muotoon, jossa kehotetaan vain miettimään jakamista, jotta

saataisiin aikaan lyhyempiä ja selkeämpiä kokonaisuuksia. Samoin pituusraja siirtyi 60 rivistä 80 riviin, koska 60 riviä tuntui kehittäjistä liioitellun lyhyeltä.

6. *”Tee testejä ominaisuuksille jo aikaisessa vaiheessa, ennen lisäystä masteriin.”* Testejä kehotettiin tekemään jo kehittämisen aikana. Ehtona oli se, että ominaisuushaaroja ei lisätä osaksi päähaaraa ennen kuin niillä on testejä. Tämäkin ohje on tietyssä mielessä johdettu NASAn ohjeista: niissä kehotettiin laittamaan jokaiseen funktioon vähintään kaksi assertiota [27]. Näin taajan yksikkötestauksen vaatimista ei kuitenkaan pidetty järkevänä.
7. *”Kirjoita muuttujat koodiin niin, että niillä on mahdollisimman pieni näkyvyysalue.”* Muuttujien kirjoittaminen koodiin niin, että niillä on mahdollisimman pieni näkyvyysalue, auttaa muuttujien ymmärtämisessä, ja siinä, ettei muuttujaan voi vahingossa viitata tarkoittaessaan jotain toista. Samoin jos huomataan, että muuttujalla on väärä arvo, on pienempi määrä paikkoja joissa se on voitu asettaa. Tämä ohje on NASAn ohjeesta 6 johdettu. [27] Käytetyistä kielistä huomattavaa on, että JavaScriptissä muuttujien näkyvyysalue voi helposti olla hyvinkin suuri ja C#:ssa public-avainsanassa on sellainen piirre, että esimerkiksi sillä määriteltyä luokan muuttujaa voi käyttää jossain toisessa ohjelmassa, eikä voi tietää, onko niin tarkoitettu.
8. *”Kun kutsut funktioita, tarkasta aina paluuarvo. Tarkista myös syötteen oikeellisuus. Käsittele mahdolliset virheet.”* Tämä ohje koski funktioiden paluuarvojen tarkistamista aina, kun funktioita kutsutaan. Lisäksi mahdolliset poikkeukset pitää huomioida ja käsitellä asianmukaisesti. Ohje perustuu NASAn ohjeeseen 7. [27] Asiasta oli huomautettu projektissa muutaman PR:n yhteydessä jo ennen ohjeistusta. Lähinnä kielen valmiita funktioita ei aina käytetty huolellisesti.
9. *”Korjaa koodista kohdat, joista kääntäjä varoittaa.”* Kääntäjän varoitusten huomioiminen ja korjaaminen on melko selvä asia. Loppuvaiheessa projektin asetuksia muutettiin niin, että varoituksetkin estivät kääntymisen, ja näytettävien varoitusten määrää myös hieman kasvatettiin. NASAn ohje 10 kehottaa samaan. Lisäksi se kehottaa käyttämään lähdekoodin analysointityökaluja, joihin ReSharper kuuluu. [27]
10. ”*Kirjoita funktioita, joilla on paluuarvo ja jotka operoivat vain parametreillaan.”* Sellaisten funktioiden kirjoittamisesta, joilla on paluuarvo ja jotka operoivat vain parametreillaan, seuraa, että järjestelmää on helpompi testata ja kääntäjä saa jo kiinni virheitä.
11. *”Käytä kannasta lukemiseen näkymiä ja kantaan lisäämiseen proseduureja itsemuodostettujen kyselyjen sijaan.”* Näkymien käyttäminen kannasta lukemiseen ja proseduurien käyttö kantaan lisäämiseen itse muodostettujen kyselyjen sijaan antaa mahdollisuuden käyttää uudelleen jo tehtyjä ohjelman osia. Sovittu jako myös selventää tietokannan ja ohjelmakoodin vastuita pitämällä ne samanlaisina kunkin kehittäjän osalta.

12. *”Kirjoita ymmärrettävää koodia. Eli käytä ReSharperia. Vilkaise myös läpi MSDN C# Coding Conventions, itse olen ainakin kirjoittanut epäinformatiivisia LINQ-kyselymuuttujia https://msdn.microsoft.com/en-us/library/ff926074.aspx. Myös General Naming Conventions on lyhyt ja yksinkertainen sivu, mutta koodin ymmärrettävyyden kannalta hyvää asiaa.”* Yleisen koodityylin pitämiseksi yhtenäisenä ja hyvänä kaikki kehittäjät käyttivät ReSharper-työkalua. Työkalu valittiin ohjeissa suositeltavaksi, koska se oli jo laajassa käytössä projektiryhmässä. Tämän lisäksi ohjeessa annettiin linkki Microsoft Developer Networkin MSDN C# Coding Conventions -sivulle [43], jolla on listattuna kieleen liittyviä hyviä tapoja, sekä General Naming Conventions -sivulle [44], jolla oli yleispäteviä ohjeita nimeämiseen.
13. *”Käytä WebEssentialsia, jos teet webjuttuja.”* Tämä ohje oli lähinnä työkaluvinkki. Siinä missä ReSharper on paljolti koodin tyyliin liittyvä Visual Studio -laajennos, WebEssentials taas on kaikenlaista web-koodausta helpottamaan pyrkivä, projektiryhmässä hyväksi todettu laajennos.
14. *”Älkää laittako mitään masteriin ilman PR:ää.”* Jotta jokaisen koodi tulisi luettua läpi vähintään kerran, päätettiin kieltää koodin laittaminen päähaaraan ilman Pull Requestia. Tämä myös pakotettiin asetuksista, kun pieniä luisumisia alkoi esiintyä.
15. *”Pitäkää PR:t sen pituisina, että ne jaksaa lukea ajatuksella läpi. Jos tiedätte, että koodi koskettaa jotakuta toista, huomauttakaa että PR on luettavissa.”* Tämä sääntö liittyy edelliseen, jotta PR:t jaksaisi lukea ajatuksella läpi ja jotta ne toisaalta ehtisi lukea kerralla, niiden pitää olla suhteellisen lyhyitä. Ja työn jouhevoittamiseksi on vielä huomautus siitä, että tehdystä PR:stä kannattaa vinkata heille, joilla luulee asiaan olevan sanottavaa.
16. ”*Jos epäilet määrittelyä, ota asia puheeksi heti.”* Projektipäällikön ehdottama ohje käski ottaa määrittelystä heräävät epäilykset puheeksi heti. Tarkoituksena on muistuttaa, että määrittelyn selventämisessä menee aina jonkin verran aikaa, ja on hyvä, jos selvitys tehdään ennen kuin asia on työn alla. Ja on paljon parempi selvittää määrittelyssä olevat viat ainakin ennen kuin koodi on valmis, sillä mitä myöhemmin vika korjataan, sitä kalliimpaa korjaus on [2].
17. *”PR:llä voidaan hakea mielipiteitä jo ennen kuin koodi on ’valmis’.”* Tämä ohje oli suhteellisen irrallinen huomautus siitä, että epäilyksiä herättävään kohtaan voi hakea kommentteja myös PR:n keinoin jo ennen kuin koodi on valmis päähaaraan lisättäväksi.
18. *”Varmistakaa että haaranne kääntyy, ennen kuin laitatte koodia masteriin.”* ”Varmistakaa, että laitatte kääntyvää koodia päähaaraan” on yksinkertainen ohje, jolla on tarkoitus varmistaa, että päähaaran koodia ei jouduta turhaan korjailemaan. Ennen ohjeiden tekoa oli joitakin tapauksia, joissa ensin testattiin oma haara, otettiin päähaaran muutokset mukaan ja unohdettiin testata että kaikki toimii edelleen, ennen kuin koodi laitettiin päähaaraan.

19. *”Pidä työkalusi ajan tasalla.”* Erilaisten ohjelmistotyökalujen päivitys on luonnollisesti osa ohjelmoijan työtä. Tähän ohjeeseen liittyen arkkitehtimme ilmoitti aina, kun yleisessä käytössä olevista työkaluista tuli uusia versioita.
20. *”Logita harkitusti, äläkä yhdistä stringejä +:lla. (linkki ohjeeseen!)”* Lokiin kirjoittamiseen oli aluksi jokaisella ohjelmoijalla oma tapansa. Tässäkin tahdottiin yhtenäistää käytäntöjä, ja toisaalta käytetyn loki-kirjaston kanssa ei ollut suositeltavaa käyttää +-merkkejä merkkijonojen yhdistelemiseen. Tämä oli tullut esille hiljattain ennen ohjeiden kirjoittamista, joten se laitettiin tännekin muistiin, mutta linkki tarkemmin syistä kertovaan ohjeeseen on jäänyt lisäämättä. Ohjeessa oli myös kyse siitä, että haluttiin, että vain tarpeelliset asiat kirjoitettaisiin lokiin ja ne olisivat oikealla kriittisyystasolla merkittyjä sekä selvästi osoittaisivat sinne, mistä ne on tulostettu.
21. *”Muutokset tietokantaan tekee joko J.S. tai joku muu projektipäällikön luvalla.”* Projektissa oli niin paljon kehittäjiä, että sovittiin, että tietty kehittäjä tekee muutokset tietokantaan, ellei projektipäällikkö erikseen anna lupaa. Tällä pyrittiin hallitsemaan tietokannan tilaa.
22. *”Lukekaa README.md ja noudattakaa sen ohjeita.”* Projektilla oli myös README.md-tiedosto, jossa oli lisää ohjeita, jotka eivät enimmäkseen olleet kriittisiä vaan käteviä vinkkejä. Myös tämä tiedosto kehotettiin vilkaisemaan läpi, jotta sieltä osaisi etsiä myöhemmin tietoa, jos ajankohtaiseksi tulisi jokin asia, johon se otti kantaa.

5.3 Käyttöönottotapa

Ohjeista tehtiin ensin luonnos, minkä jälkeen se välitettiin kehittäjille sähköpostilla. Heillä oli viikko aikaa antaa parannusehdotuksia. Tarkennetut ohjeet perusteluineen esiteltiin erään sprintin retrospektiivissä helmikuun loppupuolella. Ne todettiin hyviksi: ne eivät olleet liian vaikeat, ja ne sopivat nykyisiin käytäntöihin. Joitain muotoseikkoja lisäksi vielä tarkennettiin.

Aikaa ohjeiden noudattamiseen oli vähän yli kolme kuukautta ennen projektin valmistumisaikaa, ja takana oli yli viisi kuukautta kehittämistä, joista tosin ensimmäiset kaksi kuukautta olivat vielä harvalla miehityksellä. Laskutuksen valmistumisaika oli kuukautta muuta ohjelmaa myöhemmin, joten sille aikaa oli enemmän.

Keinoja valvottiin jo käytössä olleen PR-käytännön yhteydessä niin, että aluksi erityisesti ohjeiden alullepanija katsoisi toisten PR:iä ohjeita silmällä pitäen ja niitä kommentoiden. Kaikki muutkin olivat toki vapaita ja velvollisia tekemään näin samoin kuin noudattamaan ohjeita parhaansa mukaan.

Ohjeet olivat muun dokumentoinnin kanssa samassa SharePoint-kansiossa, eli ne olivat kenen tahansa projektin jäsenen päivitettävissä ja kaikkien luettavissa uusimmassa muodossaan. Näin noudatettiin ISO 9001 -laatustandardin ohjeajattelun periaatetta siltä osin,

että ohjeet ovat kaikkien saatavilla ja vanhentuneita ohjeita ei päivitetä [1]. Ohjeet oli tarkoitettu päivitettäviksi, jos huomattaisiin niistä puuttuvan jotain tai jokin niiden osa todettaisiin turhaksi.

Koska scrum oli kehittäjille tuttu vähintään projektin ajalta, siitä ei laadittu erillistä ohjeistusta eikä sen käyttöönottoonkaan siten kiinnitetty erityistä huomiota. Oletuksena oli, että projektin uudet jäsenet tuntevat scrumin vähintään periaatetasolla. Tämän takia voitiin odottaa, että scrumin tapaamisissa scrumin käyttö tulee niin hyvin esille, että tapaamiset toimivat käyttöönottona, jos sille on tarvetta. Lisäksi projektiryhmän yhteishenki oli niin hyvä, että lisätietoja oli helppo kysyä.

6. ARVIOINTI

Tässä luvussa käsitellään ohjeistuksen käyttöönotto ja arvioidaan ohjeiden laatua. Lisäksi käydään läpi jatkokehitysajatuksia ohjeistukseen ja sen sisältöön liittyen.

Ohjeistuksen käyttöönotossa oli merkittäviä haasteita, jotka käydään läpi ja joihin ehdotetaan ratkaisuja. Haasteiden takia ohjeistuksen suoraa vaikutusta ohjelmiston laatuun voi epäillä vähäiseksi, mutta ohjelman laatua ennen ja jälkeen ohjeistuksen arvioidaan ja mietitään, mistä mahdolliset erot voisivat johtua. Jatkokehitysajatuksia heräsi jonkin verran ja ne kirjattiin ylös vastaisen varalle.

6.1 Käyttöönotto

Ohjeiden käyttöönotto alkoi siitä, kun ensimmäinen versio ohjeista laitettiin projektitiimin jaettuun kansioon ja ohjeiden olemassaolosta ja tarkoituksesta kerrottiin sähköpostitse. Samassa postissa myös pyydettiin tutustumaan ohjeisiin, ottamaan ne käyttöön ja antamaan niistä palautetta ja parannusehdotuksia. Ohjeet olivat jo lähes nykyisen muotoiset tässä vaiheessa. Hieman yli kolmasosa tiimin jäsenistä teki muutoksia dokumenttiin tai otti asiaan muuten kantaa. Tämän jälkeen eräässä sprintin aloituspalaverissa otettiin vielä kantaa siihen, että ohjeet ovat olemassa ja niitä tulisi kehittää ja noudattaa.

Alun ohjeiden esittelyn lisäksi asian eteen ei juuri nähty vaivaa. Suunnitelmana oli huomauttaa ohjeista lipsumisessa Pull Requestien yhteydessä, mutta tämä jäi toteuttamatta, koska huoli ominaisuuksien valmistumisesta ajallaan johti siihen, että muita tehtäviä priorisoitiin aikaa vievän tarkan tarkastelun sijaan. Ohjeista ei myöskään muistutettu ensimmäisten esittelyjen jälkeen, mikä johti siihen, että elokuussa asian tullessa puheeksi eräs projektitiimin jäsen ei edes muistanut, että ohjeet olivat olemassa.

Ohjeita olisi pitänyt tuoda selvästi päättäväisemmin esille alkuvaiheessa. Sen sijaan, että ohjeet otettiin esille sprintin aloituspalaverissa, niistä olisi voinut pitää erillisen lyhyen esittelytilaisuuden, johon valmistautumiseen olisi kuulunut ohjeiden lukeminen läpi. Tämä olisi antanut ohjeiden ajattelulle enemmän tilaa. Vaihtoehtoisesti ohjeita olisi voitu käydä vain tarkemmin läpi aloituspalaverissa.

Tämän lisäksi ohjeiden noudattamista olisi ehdottomasti pitänyt edes jossain määrin valvoa. Tämä työ voi vaikuttaa tuottamattomalta, mutta se auttaa ihmisiä ymmärtämään ohjeiden merkitystä konkreettisesti sekä muistuttaa niiden olemassaolosta motivoivammalla eri tavalla kuin esimerkiksi sähköposti. Valvonta on erityisen hyödyllistä käyttöönottovaiheessa – kun sitä on tehty jonkin aikaa ja ohjeiden noudattaminen ei enää unohdu, sitä voi vähentää. Lisäksi valvontaa voi yrittää jakaa. Tässä tilanteessa valvonta oli ohjeiden laatijan vastuulla, mikä on 22 ohjeen ja 10 kehittäjän projektissa jo merkittävä vastuu.

Ohjeita olisi voinut jakaa esimerkiksi kaksi kullekin kehittäjälle, ja jokainen olisi pitänyt omien ohjeidensa noudattamista silmällä lukiessaan muiden ohjelmakoodia.

Eräs keino ohjeiden käyttöönoton helpottamiseksi olisi myös voinut olla käyttöönoton tekeminen osissa. Jos ohjeet olisi jaettu esimerkiksi neljään eri ryhmään, jotka olisi esitelty yksi kerrallaan, olisi voinut olla helpompi sisäistää, mihin kaikkeen ohjeet ottavat kantaa. Tässä projektissa aikaa oli suhteellisen vähän, joten tällainen jako ei ollut mahdollinen, mutta myös pelkkä jaottelu ilman erikseen esittelemistä olisi voinut auttaa ohjeiden hahmottamisessa. Erillisistä esittelyistä on tietysti se hyöty, että ohjeet tulevat esille monta kertaa ilman että kuitenkaan toistetaan samaa vanhaa.

Käyttöönoton epäonnistumisesta kertoo myös se, että käyttöönottoa seuraavan sprintin lopussa projektiin tullut henkilö ei missään vaiheessa saanut tietää ohjeistuksen olemassaolosta. Hän toki teki ulkoista käyttöliittymää, jonka kautta ei voi tehdä muutoksia mihinkään tietoihin, joten siinä ohjeistus ei painotuksensa puolesta välttämättä ole niin relevantti. Tarkoitus kun oli ensisijaisesti varmistaa se, että rahaa ei käsitellä virheellisesti. Ohjeissa oli kuitenkin yleishyödyllisiä asioita, ja ne tehtiin myös yleisesti yhtenäistämään tiimin koodauskäytäntöjä.

Erääseen ohjeeseen liittyen tehtiin käyttöönottoa tukevaa työtä. Pull Requesteista tehtiin pakollinen välivaihe päähaaraan ominaisuuksia lisättäessä. Eräs ongelma oli edelleen se, että myös kehittäjän oma hyväksyntä mahdollisti PR:n yhdistämisen päähaaraan. Tämä mahdollisuus kuitenkin haluttiin säilyttää, sillä julkaisun tekevä kehittäjä joutui joskus muokkaamaan päähaaraa saadakseen julkaisun tehtyä. Ja koska kaikki PR:t, niiden hyväksynnät ja päähaaraan yhdistymiset näkyivät Visual Studio Onlinen virtuaalisessa keskusteluhuoneessa, oli helppo huomata, jos joku hyväksyi oman PR:nsä. Muutoksen jälkeen omien PR:ien hyväksymisestä alettiin huomauttaa ahkerasti ja ongelma saatiin nopeasti kuriin.

Ohjeiden käyttöönotosta on siis PR:ien onnistuneita kokemuksia. Tiimi koki asian tärkeäksi, ja siihen tartuttiin. Muuten käyttöönotto kärsi sellaisista ongelmista, joita nostettiin esille scrumin käyttöönottoa käsittelevässä tutkimuksessa ja joihin Haikala viittasi laatujärjestelmien käyttöönoton usein hiipuvan [1; 32].

6.2 Ohjeistuksen laatu

Ohjeistuksen laadun mittaaminen ei ole yksinkertaista. Koska ohjeistuksen tarkoitus oli parantaa ohjelman laatua, ohjelman laatua mittaamalla voi saada jotain selville ohjeiden laadusta. Valitut mittarit ovat raportoitujen ja myöhemmin korjattujen virheiden määrä ennen ja jälkeen ohjeistuksen käyttöönoton, sekä virhekorjauksia sisältävät Pull Requestit. Kumpaakin mittaria tarkkaillaan kehityksen alusta version 1.1 viimeiseen PR:ään asti, sillä siihen mennessä kaikki suuret toiminnallisuudet oli toteutettu ja ensimmäisen version ongelmia jo huomattu ja korjattu.

Raportoituja virheitä ennen ohjeiden käyttöönottoa oli 277 ja käyttöönoton jälkeen 283. Testauksen määrä vaihteli. Aivan projektin alussa oli vähemmän valmiita osia testattavana, ja toisaalta version 1.0 julkaisun jälkeen testaustahti rauhoittui. Virheitä ovat toki raportoineet myös kehittäjät työnsä ohessa eivätkä vain testaajat. PR:iä oli tehty ohjeiden käyttöönottoon mennessä 452 kpl, ja sen jälkeen tehtiin 667 kpl. Suhteessa raportoituja virheitä tehtyä PR:ää kohden oli ennen ohjeita noin 0,6 ja ohjeiden jälkeen noin 0,4. PR ei kuitenkaan ole täydellinen kehityksen määrän mittari, sillä PR:t voivat olla hyvinkin eri laajuisia, ja varsinkin juuri ennen käyttöönottoa monet PR:t olivat hyvin pieniä viilailuja. Toisaalta projektin alussa ei ollut pakko tehdä PR:ää ennen koodin laittamista päähaaraan, mutta näin päähaaraan päätyneen ohjelmakoodin määrä pitäisi olla marginaalinen, sillä päähaaraan laitettiin vain hyvin pieniä korjauksia ilman PR:ää.

Taulukko 6.1: Raportoitujen virheiden ja PR:ien suhde ennen ohjeistusta ja sen jälkeen.

	Raportoidut virheet	**Pull Requestit**	**Virheitä/PR**
Ennen ohjeistusta	277	452	0,61
Ohjeistuksen jälkeen	283	667	0,42

Jos vertaa raportoitujen virheiden määrää kehityksen määrään, vaikuttaa siltä, että laatu olisi parantunut ohjeiden käyttöönoton jälkeen. Koska ohjeiden käyttöönottoon ei kuitenkaan juurikaan nähty vaivaa, tälle johtopäätökselle ei ole perusteita. Toki voi olla, että pelkkä ohjeistuksen läpikäyminen on aiheuttanut pienen positiivisen asennemuutoksen, mutta tätä on vaikea mitata. Suhdelukuun voi vaikuttaa kehityksen määrän mittaamisen vaikeuden lisäksi se, että käyttöliittymä oli valmis jo melko aikaisessa vaiheessa, ja siihen voi liittyä lukuisia helposti näkyviä virheitä. Ja luonnollisesti projektin loppuvaiheessa aikaa käytettiin uusien ohjelman osien tekemisen lisäksi merkittävissä määrin vanhojen korjaamiseen. Valitettavasti korjaukset eivät ole omissa PR:issään vaan yleensä osana jotain suurempaa kokonaisuutta, joten uuden tekemisen ja vanhan korjaamisen välistä suhdetta on vaikea laskea. Toisaalta ohjeistuksen jälkeen raportoidut virheet voivat myös hyvin liittyä ohjelmakoodiin, joka on kirjoitettu jo ennen ohjeistuksen käyttöönottoa.

6.3 Jatkokehitysajatuksia

Jatkokehitysajatuksia tuli melko paljon. Yksittäisenä aihealueena kiinnostava on se, miten ohjeet voisivat erota tehdyistä, jos ne tehtäisiin heti projektin alussa. Jatkokehitysajatukset on esitetty oletetussa hyödyllisyysjärjestyksessä samankaltaiselle projektille, ensimmäisenä se, joka oletetaan hyödyllisimmäksi.

Pull Requesteihin liittyy etujen lisäksi ongelma. Joskus jokin niistä jäi pitkäksi aikaa odottamaan, kun kukaan ei lukenut sitä. Toisaalta joissain kiiretilanteissa niitä vain hyväksyttiin lukematta. Tämä on kuitenkin ongelma, joka ehkä olisi paremmin ratkaistavissa tehostetulla sisäisellä kommunikaatiolla ohjeen sijaan. Mahdollinen ohje voisi ehdottaa kiertävää pakkoa lukea PR. Näin kullakin PR:llä on ainakin yksi varma lukija tiedossa heti alussa. Tämä voisi myös laajentaa ihmisten ymmärrystä koodista, sillä on helppo lukea vain PR:iä aiheista, jotka ymmärtää jo.

Miettiessä ohjeita, joita projektissa olisi voinut olla vanhojen lisäksi, nousi esille eräs, joka oli käytössä, mutta jäi välillä noudattamatta. Tämä tapa oli Visual Studio Onlineen luotujen tehtävien numeroiden mainitseminen versionhallinnan kommitointiviesteissä, jotta kehityksestä jäisi enemmän dokumentaatiota. Jos tehtävät kirjattiin viesteihin, myös tehtäviin tuli linkki PR:ään, jonka yhteydessä siihen liittyviä asioita on päivitetty.

Ohjeita ei nyt päivitetty aivan alun jälkeen. Vaikka erilaiset jaetut kansiot tarjoavatkin muutoshistorian, olisi dokumentin aloittaminen muutoshistoriataulukolla hyvä ajatus. Näin muiden tekemät tarkennukset ohjeisiin eivät jää huomaamatta. Kunkin sprintin lopetuspalaverissa olisi myös voinut käydä ohjeet läpi, jos niihin olisi tullut muutoksia jonkin sprintin aikana heränneen ajatuksen johdosta.

Projektin kulusta kysellessä tuli esille, että projektiin tullessaan moni kehittäjistä koki, ettei perehdytystä juurikaan ollut. Jos jatkossa projektille laatii ohjeistusta, voisi harkita, että siihen tehdään myös osa, joka on tarkoitettu erityisesti projektiin uusina tuleville henkilöille. Tässä osassa voisi listata projektin erikoispiirteitä, siihen liittyviä tärkeimpiä muita dokumentteja ja käytäntöjä sekä projektissa työskenteleviä henkilöitä ja heidän erikoisosaamisiaan. Tällainen ohjeistus ei tietenkään korvaa perehdytystä, mutta se voisi toimia muistin tukena ensin perehdyttäjälle ja myöhemmin perehdytetylle.

Eräs yksittäinen asia, johon olisi voinut ottaa kantaa ohjeissa, oli Scrumin tehtävien koko ja nimeäminen. Asiasta oli lyhyt maininta projektin readme-tiedostossa, mutta vaikka tehtäviä käytiin yhdessä läpi sprinttejä päättäessä ja aloittaessa, tehtävien nimeämiseen tai kestoon ei juurikaan otettu kantaa. Pienemmät ja tarkemmin määritellyt tehtävät olisivat voineet johtaa pienempiin funktioihin ja PR:iin sekä ohjelman edistymisen helpompaan seurantaan.

Ohjeiden suunnittelu jo projektin alussa muuttaa tilannetta selvästi. Tällaisessa tilanteessa ei toisaalta ole vielä muodostunut projektille omia hyviä käytäntöjä, joten niiden dokumentoiminen ohjeisiin heti alussa ei ole mahdollista. Toisaalta, kun ohjeistus on jatkuvasti muuttuva dokumentti, tämä ei ole ongelma ja uudet hyvät käytännöt voidaan lisätä ohjeiksi sitä mukaa, kun niitä muodostuu.

Laadunvarmistamisen kannalta yksikkötestien automaattinen generointi vaikuttaa suhteellisen yksinkertaiselta. Sitä ei ehditty tämän projektin puitteissa testata, mutta keino vaikuttaa kokeilemisen arvoiselta.

Jatkuvan kehittämisen CMM-malliin liittyy monia käytäntöjä. Se tähtää yrityksen oman laatujärjestelmän luontiin ja kehittämiseen. [1, ss. 195–197] Laatujärjestelmä mahdollistaa ohjeiden toiminnan mittaamisen projektien välillä ja toisaalta mittarien kehittämisen ajan myötä. Tällaiseen järjestelmään voi kuulua projektikohtainen ohjeistus ja sitä varten olemassa oleva malli. Jos yrityksessä tehdään paljon laatukriittisiä projekteja, joissa halutaan käyttää ohjeistusta, kannattaa harkita omaa runkoa laatuohjeistukselle ja tapoja mitata ja kehittää sitä, esimerkiksi CMM-mallia mukaillen.

Jos ohjeistus olisi käytössä jo projektia aloittaessa, voisi se ottaa kantaa ohjelmointikielen valintaan. Laatuun liittyvän ohjeistuksen ollessa kyseessä, voi harkita erityisesti turvallisuuskriittiseen ohjelmointiin kehitetyn tai soveltuvan kielen käytön suosittelua. Tähän liittyy kuitenkin ongelmia: Yleisemmässä käytössä olevat kielet ovat useammin ohjelmoijille tuttuja, joten niihin liittyen on helpompi löytää tietoa ja neuvoja ja niihin liittyen on olemassa laajoja kirjastoja. Uuden kielen logiikan omaksuminen vie aikaa ja voi hidastaa työtä. Samoin alussa voi tehdä virheitä, koska ei ymmärrä kieltä. Tästä syystä ei ole syytä kirjoittaa edes ohjeistukseen joka on tarkoitettu hyvin korkeaa laatua vaativille projekteille, että projekti tulee tehdä turvallisuuskriittiseen ohjelmointiin kehitetyllä kielellä. Mutta ohjeistukseen voi laittaa maininnan siitä, että on olemassa erityisesti laadun varmistamista ajatellen kehitettyjä ohjelmointikieliä.

7. YHTEENVETO

Maksujärjestelmäprojekti oli luonteeltaan hyvin haastava. Projektilla oli tiukka aikataulu, joka oli projektin kokoluokkaan nähden lyhyt. Tehtävä ohjelma myös käsittelee rahaa, jolloin virheitä halutaan välttää. Asiakkaiden tyytyväisyys tuotteeseen on aina tärkeää tekijän maineen kannalta. Nyt tyytyväisyys oli erityisen tärkeää, sillä jos ostajat toteavat ohjelman hyväksi, on todennäköistä, että siihen tehdään jatkokehitystä. Jatkokehityksen mahdollisuus taas asettaa vaatimuksia ohjelman laadulle.

Kun kehittäjillä on erilaiset taustat ja heitä on suhteellisen paljon projektin kokoon nähden, ja on kiire, ohjelman laatu ja tyyli ovat uhattuina. Jos ohjelmisto päätyy tilaan, jossa se on niin sekava, että sitä on vaikea jatkokehittää ja virheitä on runsaasti, kehitystahti hidastuu merkittävästi. Tässä projektissa ei haluttu joustaa aikataulusta, joten ohjelma haluttiin pitää niin selkeänä, että kehittäjien on helppo tehdä muutoksia kaikkiin ohjelman osiin. Virheet ovat toki myös itsessään ongelma. Virheiden välttämiseksi ja muun laadun pitämiseksi korkeana käytiin läpi laadunvarmistuskäytäntöjä liittyen ohjelman tekoprosessiin ja itse ohjelmointiin.

Keinojen läpikäynti ei yksin riitä parantamaan laatua vaan ne pitää saada käyttöön, jotta niistä olisi hyötyä. Tämän takia päätettiin tehdä ohjeistus. Ohjeistusta muodostettaessa tavoiteltiin muokattavuutta, helppokäyttöisyyttä ja omaksumisen helppoutta. Ohjeisiin otettiin mukaan projektiryhmässä jo valmiiksi käytössä olleita laadunvarmistuskäytäntöjä sekä tutkituista laadunvarmistuskäytännöistä projektiin sopiviksi todettuja ja muokattuja osia. Myös ohjeiden käyttöönotto suunniteltiin, ja sen yhteydessä kerättiin projektiryhmän ajatuksia ohjeistuksesta ja muokattiin ohjeita niiden perusteella.

Käyttöönoton toteutus jäi vajavaiseksi, ja siksi on oletettava, että ohjeet eivät vahventaneet laadunvarmistusta. Ohjeiden mahdollista vaikutusta siihen, että laatuun kiinnitettiin enemmän huomiota vaikka kaikkia yksittäisiä ohjeita ei noudatettukaan, on vaikea arvioida. Osa ohjeista oli jo käytössä ennen ohjeistusta, ja niitä ei ainakaan lakattu noudattamasta ohjeiden takia, saattaa olla, että käyttö yleistyi projektiryhmän sisällä. Lopputuotteesta on saatu hyvää palautetta, ja projekti päättyi ajallaan, joten tämä lievästi puoltaa sitä, että käytössä olleet keinot olivat hyödyllisiä. Tarkkaa tietoa siitä, mitkä keinot olivat käytössä ja miten laajasti, ei kuitenkaan ole.

Tulevaa varten koottiin parannus- ja jatkokehitysajatuksia esimerkiksi sellaista tilannetta varten, jossa ohjeistusta pääsee tekemään heti projektin alussa. On vaikea yrittää parantaa sellaisia osia ohjeista, jotka eivät olleet käytössä. Tehtyä ohjeistusta voi kuitenkin käyttää pohjana uudelle laadunvarmistusohjeistukselle, sillä ainakin osa ohjeistuksen käytännöistä perustuu käytössä hyvältä vaikuttaneisiin tapoihin ja muut on valittu niin, ettei

niistä pitäisi ainakaan olla kehitystyölle suurta haittaa. Etenkin tulee muistaa, että jos ohjeistuksen haluaa ottaa käyttöön, on varmistettava, että joku on vastuussa siitä, että näin tapahtuu. Uuteen projektiin laadunvarmistuskeinoja etsiessä tai ohjeita suunnitellessa kannattaa tutustua tehdyn ohjeistuksen lisäksi käyttöönoton haasteisiin ja jatkokehitysajatuksiin.

LÄHTEET

[1] I. Haikala, J. Märijärvi, Ohjelmistotuotanto, 6. Painos ed. Suomen Atk-kustannus Oy, Gummerus Kirjapaino Oy, Jyväskylä, 1998, 389 p.

[2] B. Haskins, 8.4.2 Error Cost Escalation Through the Project Life Cycle, INCOSE International Symposium, Vol. 14, No. 1, 2004, pp. 1723-1737.

[3] Guide to Agile Practices, Agile Alliance, web page. Available (accessed 22.08.2015): http://guide.agilealliance.org/.

[4] The Scrum Guide, Scrum.Org, ScrumInc., web page. Available (accessed 09.10.2015): https://www.scrumalliance.org/why-scrum/scrum-guide.

[5] Guide to Agile Practices - Pair Programming, Agile Alliance, web page. Available (accessed 23.08.2015): http://guide.agilealliance.org/guide/pairing.html.

[6] Guide to Agile Practices - Quick Design Session, Agile Alliance, web page. Available (accessed 23.08.2015): http://guide.agilealliance.org/guide/quickdesign.html.

[7] J. Nosek, Case for collaborative programming, Communications of the ACM, Vol. 41, No. 3, 1998, pp. 105-108.

[8] M.L. Kim, The Effect of Pairs in Program Design Tasks, IEEE Transactions on Software Engineering, Vol. 34, No. 2, 2008, pp. 197-211.

[9] H. Hulkko, A multiple case study on the impact of pair programming on product quality, Proceedings, 2005, pp. 495-504.

[10] E. Arisholm, Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise, IEEE Transactions on Software Engineering, Vol. 33, No. 2, 2007, pp. 65-86.

[11] M. Fowler, Continuous Integration, Martin Fowler, web page. Available (accessed 22.08.2015): http://martinfowler.com/articles/continuousIntegration.html.

[12] D. Goodman, "It's Not the Pants, it's the People in the Pants" Learnings from the Gap Agile Transformation What Worked, How We Did it, and What Still Puzzles Us, Agile 2008 Conference, 2008, pp. 112-115.

[13] Guide to Agile Practices - Rules of Simplicity, Agile Alliance, web page. Available (accessed 22.08.2015): http://guide.agilealliance.org/guide/rules-of-simplicity.html.

[14] Guide to Agile Practices - Tdd, Agile Alliance, web page. Available (accessed 23.08.2015): http://guide.agilealliance.org/guide/tdd.html.

[15] H. Munir, Considering rigor and relevance when evaluating test driven development: A systematic review, Information & Software Technology, Vol. 56, No. 4, 2014, pp. 375-394.

[16] Guide to Agile Practices - Unit Testing, Agile Alliance, web page. Available (accessed 23.08.2015): http://guide.agilealliance.org/guide/unittest.html.

[17] Guide to Agile Practices - Exploratory Testing, Agile Alliance, web page. Available (accessed 23.08.2015): http://guide.agilealliance.org/guide/exploratory.html.

[18] T. Tenny, Program Readability: Procedures Versus Comments, IEEE Transactions on Software Engineering, Vol. 14, No. 9, 1988, pp. 1271-1279.

[19] J.L. Elshoff, Improving Computer Program Readability to Aid Modification, Communications of the ACM, Vol. 25, No. 8, 1982, pp. 512-521.

[20] D. Steidl, Quality analysis of source code comments, 2013 21st International Conference on Program Comprehension, 2013, pp. 83-92.

[21] F. Salviulo, Dealing with identifiers and comments in source code comprehension and maintenance: results from an ethnographically-informed study with students and professionals, Evaluation and Assessment in Software Engineering Proceedings of the 18th International Conference, 2014, pp. 1-10.

[22] E. Wong, CloCom: Mining existing source code for automatic comment generation, 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, 2015, pp. 380-389.

[23] M.E. Fagan, Advances in Software Inspections, IEEE Transactions on Software Engineering, Vol. 12, No. 7, 1986, pp. 744-751.

[24] Git, web page. Available (accessed 26.6.2015): https://git-scm.com/.

[25] G. Gousios, An exploratory study of the pull-based software development model, Software Engineering Proceedings of the 36th International Conference, 2014, pp. 345-355.

[26] G. Holzmann J., SCRUB: a tool for code reviews, Innovations In Systems And Software Engineering, Vol. 6, No. 4, 2010, pp. 311-318.

[27] The power of 10: Rules for developing safety-critical code, Computer, Vol. 39, No. 6, 2006, .

[28] M. Kim, A field study of refactoring challenges and benefits, Foundations of Software Engineering Proceedings of the ACM SIGSOFT 20th International Symposium, 2012, pp. 1-11.

[29] Q.D. Soetens, Studying the Effect of Refactorings: A Complexity Metrics Perspective, 2010 Seventh International Conference on the Quality of Information and Communications Technology, 2010, pp. 313-318.

[30] T. Latoza D., Maintaining mental models: a study of developer work habits, Software engineering Proceedings of the 28th international conference, 2006, pp. 492-501.

[31] X. Li, Effectively teaching coding standards in programming, Conference On Information Technology Education: Proceedings of the 6th conference on Information technology education; 20-22 Oct.2005, pp.239-244, 2005, pp. 239-22.

[32] H. Hassan, T. Al Shaima, Adopting Agile Software Development: Issues and Challenges, Vol. 2, No. 3, 2011, .

[33] K. Conboy, People over Process: Key Challenges in Agile Development, IEEE Software, Vol. 28, No. 4, 2011, pp. 48-57.

[34] J.C. Lusth, Language subsetting via reflection and overloading, 2009 39th IEEE Frontiers in Education Conference, 2009, pp. 1-6.

[35] Getting Started with the .NET Framework, Microsoft, web page. Available (accessed 26.6.2015): https://msdn.microsoft.com/library/hh425099%28v=vs.110%29.aspx.

[36] Visual C#, Microsoft, web page. Available (accessed 26.6.2015): https://msdn.microsoft.com/en-us/library/kx37x362.aspx.

[37] About JavaScript, Mozilla Developer Network, web page. Available (accessed 26.6.2015): https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.

[38] Microsoft SQL Server, Microsoft, web page. Available (accessed 26.6.2015): https://technet.microsoft.com/en-us/library/bb545450.aspx.

[39] What Is Visual Studio Online? Microsoft, web page. Available (accessed 26.6.2015): https://www.visualstudio.com/en-us/products/what-is-visual-studio-online-vs.aspx.

[40] Top 10 2013-Top 10, OWASP Foundation, web page. Available (accessed 14.07.2015): https://www.owasp.org/index.php/Top_10_2013-Top_10.

[41] K. Lui, A road map for implementing eXtreme Programming, Unifying The Software Process Spectrum, Vol. 3840, 2005, pp. 474-481.

[42] M. Meyer, Continuous Integration and Its Tools, IEEE Software, Vol. 31, No. 3, 2014, pp. 14-16.

[43] C# Coding Conventions (C# Programming Guide), Microsoft, web page. Available (accessed 09.03.2015): https://msdn.microsoft.com/en-us/library/ff926074.aspx.

[44] General Naming Conventions, Microsoft, web page. Available (accessed 09.03.2015): https://msdn.microsoft.com/en-us/library/ms229045%28v=vs.110%29.aspx.

LIITE A: MUODOSTETTU OHJEISTUS

Koodauskäytäntöjä

Tee parannuksia näihin ohjeisiin, ja lisää uusia. Koska kyseessä on osa dippatyötäni, arvostaisin tietoa muutoksista.

Rahan käsittelyssä ei ole syytä tehdä virheitä, vielä vähemmän kuin ohjelmakoodissa yleensä. Dokumentti on tarkoitettu koodauskäytäntöjen hiomiseen erityisesti suoraan rahaan liittyvässä koodissa. Myös tietoturvallisuus on yksi tärkeä osa rahan käsittelyä, mutta README.md-dokumentti käsittelee sitä enemmän.

Myös henkilö- ja pankkitietojen tietoturva on tärkeää, mutta sitä ei ole otettu erityisesti huomioon tässä ohjeessa.

Ensin olisi tarjolla checklist:

1. Varo pyöristysvirheitä. Decimal-tyyppi toimii C#-koodissa ja kannassa, JavaScriptissä kokonaisluvut.
2. Kommentoi koodia. Koska kaikki kirjoittavat selkeää ja ymmärrettävää koodia, voi kommenteissa keskittyä siihen miksi, ei niinkään miten.
3. Älä käytä rekursiota. Kyse on koodin selkeydestä, ja sitä kautta bugittomuudesta. Jos rekursion kiertäminen tuntuu vaikealta, kysy neuvoa. Jos vaihtoehtoinen ratkaisu on edelleen kovin monimutkainen, käytä rekursiota.
4. Käytä mahdollisuuksien mukaan foreach-silmukkaa. Jos se ei sovi, testaa silmukan rajat.
5. Jos funktio tai proseduuri on yli 80 riviä pitkä, käytä 5 min aikaa sen miettimiseen, eikö sitä saisi jotenkin selkeästi jaettua. Mieti samaa myös pitkien luokkien kohdalla.
6. Tee testejä ominaisuuksille jo aikaisessa vaiheessa, ennen lisäystä masteriin.
7. Kirjoita muuttujat koodiin niin, että niillä on mahdollisimman pieni näkyvyysalue.
8. Kun kutsut funktioita, tarkasta aina paluuarvo. Tarkista myös syötteen oikeellisuus. Käsittele mahdolliset virheet.
9. Korjaa koodista kohdat, joista kääntäjä varoittaa.
10. Kirjoita funktioita, joilla on paluuarvo ja jotka operoivat vain parametreillaan.
11. Käytä kannasta lukemiseen näkymiä ja kantaan lisäämiseen proseduureja itsemuodostettujen kyselyjen sijaan.
12. Kirjoita ymmärrettävää koodia. Eli käytä ReSharperia. Vilkaise myös läpi MSDN C# Coding Conventions, itse olen ainakin kirjoittanut epäinformatiivisia LINQ-kyselymuuttujia https://msdn.microsoft.com/en-us/library/ff926074.aspx. Myös

General Naming Conventions on lyhyt ja yksinkertainen sivu, mutta koodin ymmärrettävyyden kannalta hyvää asiaa.

13. Käytä WebEssentialsia, jos teet webjuttuja.
14. Älkää laittako mitään masteriin ilman PR:ää.
15. Pitäkää PR:t sen pituisina, että ne jaksaa lukea ajatuksella läpi. Jos tiedätte, että koodi koskettaa jotakuta toista, huomauttakaa että PR on luettavissa.
16. Jos epäilet määrittelyä, ota asia puheeksi heti.
17. PR:llä voidaan hakea mielipiteitä jo ennen kuin koodi on ”valmis”.
18. Varmistakaa että haaranne kääntyy, ennen kuin laitatte koodia masteriin.
19. Pidä työkalusi ajan tasalla.
20. Logita harkitusti, äläkä yhdistä stringejä +:lla. (linkki ohjeeseen!)
21. Muutokset tietokantaan tekee joko J.S. tai joku muu projektipäällikön luvalla.
22. Lukekaa README.md ja noudattakaa sen ohjeita.

Muutama tietoturva (OWASP) -linkki:

NET-security cheat sheet kannattaa vilkaista, ja ominaisuudesta riippuen myös jokin muu cheat sheeteistä. https://www.owasp.org/index.php/.NET_Security_Cheat_Sheet

OWASP:in top 10 lista kuuluu yleissivistykseen https://www.owasp.org/index.php/Top_10_2013-Top_10

Harkitkaa OWASP:n Developer guiden lukemista, tämä tosin on sitten pidempi teksti. Liittyy enemmän tietoturvaan kuin rahan käsittelyyn, mutta on maininnan arvoinen teos. https://www.owasp.org/index.php/OWASP_Guide_Project#tab=Main

Pariohjelmoinnista ja sen hyödyntämisestä projektissa:

Kun jossain vaiheessa alkaa olla vaikeaa käyttää enää näin montaa koodaria projektissa, voi resursseja joko siirtää toisiin projekteihin, tai sitten käyttää pariohjelmointia. Joko mahdollisia vielä jäljellä olevia ominaisuuksia, tai vaikeita virheitä voi olla hyvä ratkoa pareina. Pari koodaa keskimäärin virheettömämpää ja laadukkaampaa koodia kuin yksittäinen ihminen, ja myös nopeammin, jos koodi ei ole helposti jaettavissa. Virheiden ratkominen parina voi olla hankalampaa, koska toisella parista on enemmän kokemusta, mutta sitäkin voisi koittaa. Turvautukaa toki jo nyt työkavereihin vaikeissa kohdissa, ja lukekaa PR:iä huolella. Ehkä ihmisiä voisi PR:ien suhteen parittaa?

Lähteinä: Nasan checklist http://spinroot.com/gerard/pdf/P10.pdf, arkkitehtimme, projektin README.md -tiedosto, pariohjelmointiartikkeleja, OWASP:n sivut ja oma pää.