Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Tuomas Maanonen

Ohjelmistonkehitysprosessin kehittäminen Sarake Oy:lle

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

18.5.2020

Tekijä Otsikko Sivumäärä Aika	Tuomas Maanonen Ohjelmistonkehitysprosessin kehittäminen Sarake Oy:lle 47 sivua 18.5.2020
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	tieto- ja viestintätekniikka
Ammatillinen pääaine	ohjelmistotuotanto
Ohjaajat	konsultti Tuukka Taskinen lehtori Simo Silander

Insinöörityön tavoitteena oli käsitellä ohjelmistonkehityksen teoriaa, ohjelmistokehityksenprosessin kehittämisen tueksi. Tämä tehtiin käymällä läpi alan tutkimuksia ja kirjallisuutta. Työn lähtökohtana oli löytää toimivia menetelmiä varsinkin pitkän elinkaaren jatkuvien palveluiden kehittämiseen.

Työhön valikoitui seitsemän ohjelmistokehityksen eri vaihetta/osiota, jotka ovat vaatimusten hallinta, arkkitehtuurisuunnittelu, sovelluskehitys, testaus, versionhallinta, konfiguraationhallinta ja julkaisun hallinta. Näitä vaiheita pidettiin tärkeimpinä osina tilaajayrityksen ohjelmistonkehitystä. Työssä käytiin läpi jokaisen vaiheen oleellisia malleja ja toimintatapoja.

Kirjallisuuden avulla saatiin selvitettyä, mitä eri kuvatuilla vaiheilla tavoitellaan ja miten vaiheiden tavoitteet voitaisiin saavuttaa. Lisäksi jokaisen vaiheen osalta esitettiin kehitysehdotuksia tilaajayritykselle. Kehitysehdotuksien käytännön testaus jätettiin jatkokehitysprojektiin.

Avainsanat	ohjelmistotuotanto, ohjelmistokehitys, prosessit

Author Title Number of Pages Date	Tuomas Maanonen Developing Software Development Process 47 pages 18 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Simo Silander, Senior Lecturer Tuukka Taskinen, Consultant

The goal of this study was to go through and evaluate industry studies and literature to find a solid theoretical basis for future software development process advancements at Sarake Oy. The ability to support the development of continuous cloud services was a core requirement applied during the evaluation.

Software development was split into seven parts i.e. requirements management, architecture design, implementation, testing, version control, configuration management and release management. These steps were thought to be the most important for Sarake Oy. Models and methods supporting the goals of each step were presented.

Through the literature, the main goals of each step were successfully defined. Also, several proposals for software development at Sarake Oy were made. The practical testing and implementation of the proposals were left to be done in the follow-up project.

Keywords	software engineering, software development, processes

Sisällys

1 Johdanto 1

2 Ohjelmistonkehitysprosessi 3

3 Prosessin kehitysvaiheet 6

3.1 Vaatimusten hallinta 6

3.1.1 Vaatimuksen määritelmä 6

3.1.2 Vaatimusten hallinnan periaatteet 8

3.1.3 Vaatimusten kehittäminen 9

3.2 Arkkitehtuurisuunnittelu 14

3.2.1 Arkkitehtuurisuunnittelun mallit ja menetelmät 14

3.2.2 Arkkitehtuurisuunnittelun vaiheet 16

3.3 Sovelluskehitys 22

3.3.1 Etenemisen seuranta 22

3.3.2 Sovelluskehityksen menetelmä 24

3.4 Testaus 27

3.4.1 Yksikkö- ja integraatiotestaus 27

3.4.2 Regressiotestaus 28

3.4.3 Hyväksymistestaus 29

4 Prosessin tukivaiheet 32

4.1 Versionhallinta 32

4.1.1 Versionhallinnan mallit 32

4.1.2 Versionhallinnan toimenpiteet 35

4.2 Konfiguraationhallinta 37

4.2.1 Konfiguraationhallinnan määritelmä 37

4.2.2 Konfiguraationhallinnan toteutus 39

4.3 Julkaisun hallinta 41

5 Yhteenveto 43

Lähteet 45

1 Johdanto

Uusia ohjelmistokehitykseen liittyviä menetelmiä, ja työkaluja ilmestyy vuosittain. Kypsempiin aloihin verrattuna standardointi on alalla vähäistä, ja eri ammattilaiset tekevät usein samoja asioita hyvin eri menetelmillä. Parhaimmat menetelmät ja työkalut eivät ole ilmiselviä. Viime vuosikymmeninä yleinen trendi on ollut prosessien yksinkertaistaminen, varsinkin automaatiota hyödyntämällä. Lisäksi liiketoiminnallisesti ohjelmistoala – kuten monet muutkin alat – on kehittynyt poispäin tuotteiden toimittamisesta suoraan asiakkaille, jatkuvien palveluiden tuottamiseen. Tämän tapainen liiketoiminta on tuonut mukanaan sovelluksille huomattavasti aikaista pidemmän elinkaaren ja entistä nopeamman julkaisutahdin.

Edellä kuvaillun muutoksen keskellä on myös työn tilaaja, suomalainen ohjelmistoalan yritys Sarake Oy. Yrityksen erityisalaa on kaikenlainen tiedonhallinta, muun muassa dokumenttien, sopimusten ja ideoidenhallinta. Perinteisesti konsultointia tarjonnut yritys on siirtynyt nopeasti kehittämään ja tarjoamaan omia valmiita ratkaisukokonaisuuksia ja jatkuvasti kehitettäviä palveluita. Jatkuvasti kehitettävissä palveluissa pyritään yhdistämään standardisoidut ydinominaisuudet ja asiakkaan ilmeeseen ja prosesseihin mukautuminen. Näiden jatkuvien palvelujen kehittäminen vaatii myös jatkuvaa ohjelmistonkehityksenprosessin kehitystä.

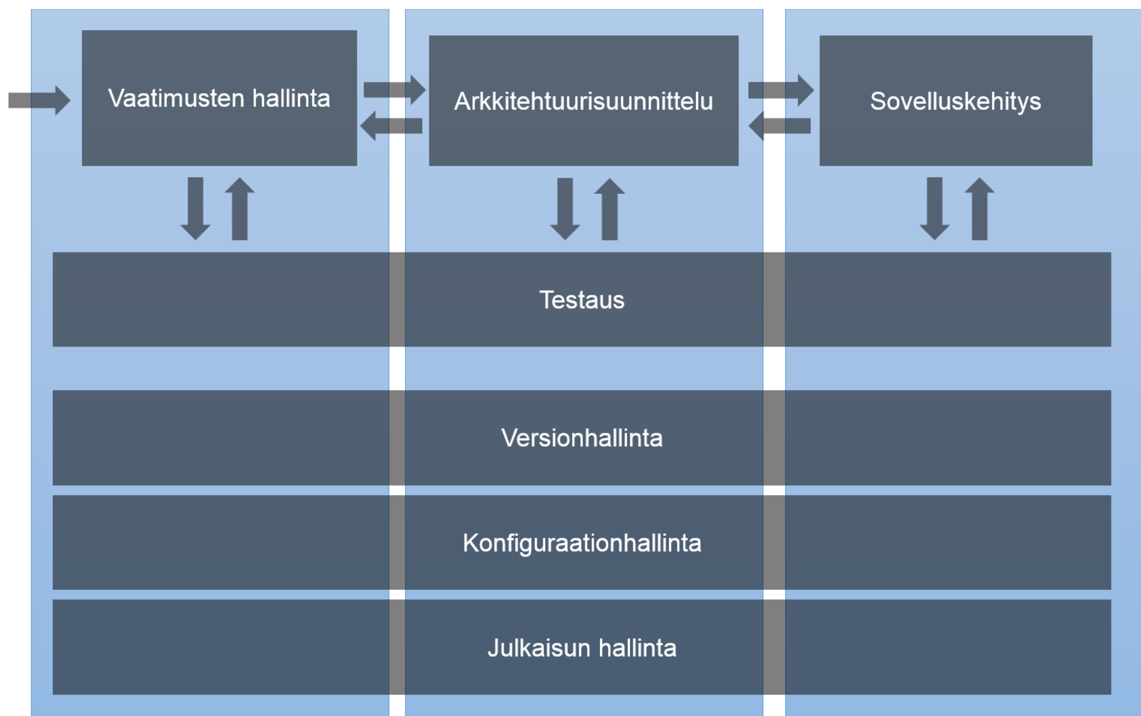
Tämä insinöörityö on osa tilaajayrityksen ohjelmistonkehityksenprosessin kehitystyötä. Tähän kehitystyöhön kuuluu tämä insinöörityö ja sitä seuraava jatkokehitysprojekti.

Tässä insinöörityössä pyritään lukemaan ja käsittelemään mahdollisimman paljon alan kirjallisuutta, jonka perusteella pyritään keräämään yhteen tarvittava teoria yrityksen nykyisten menetelmien kehittämiseen ja formaalin prosessin luomiseen. Lisäksi tässä insinöörityössä tehdään konkreettisia kehitysehdotuksia sovellettavaksi jatkokehitysprojektissa.

Insinöörityötä seuraavassa jatkokehitysprojektissa kehitysehdotuksia testataan käytännössä ja niistä valitaan toimivat osaksi lopullista ohjelmistonkehityksenprosessia. Tässä vaiheessa luodaan lopulliset kirjalliset kuvaukset kehitetyistä prosesseista.

2 Ohjelmistonkehitysprosessi

Tässä työssä kuvattavassa ohjelmistonkehitysprosessissa on seitsemän päävaihetta ja useita alivaiheita. Päävaiheet ovat vaatimusten hallinta, arkkitehtuurisuunnittelu, sovelluskehitys, testaus, versionhallinta, konfiguraationhallinta ja julkaisun hallinta. Jokainen vaihe kuvaa jotain asiaa yrityksen ohjelmistotuotannossa.

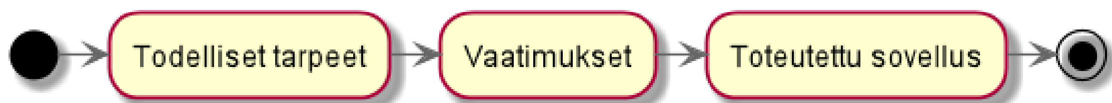


Kuva 1. Päävaiheiden suhteet.

Kuvassa 1 esitetään päävaiheiden suhteet. Kuvassa nuolet viittaavat siihen, että esitetyssä prosessissa liikutaan aina vaatimusten hallinnan, arkkitehtuurisuunnittelun, sovelluskehityksen ja testauksen välillä. Sen sijaan versionhallintaa, konfiguraationhallintaa ja julkaisun hallintaa suoritetaan jatkuvasti muun toiminnan rinnalla. Kuitenkin näissäkin hallintaa tehdään joko vaatimusten hallinnan, arkkitehtuurisuunnittelun tai sovelluskehityksen tuotosten näkökulmasta. Prosessiin tullaan sisään vaatimusten hallintaan, mutta sen jälkeen liikkuminen vaiheiden välillä päätetään ennen kaikkea kyseisen ohjelmistokehitysprojektin tarpeiden mukaan.

Metropolia

Kaikessa yksinkertaisuudessaan prosesseilla pyritään luomaan ja varmistamaan tie todellisista tarpeista vaatimuksiin ja toteutettuun ohjelmistoon kuvan 2 esittämässä järjestyksessä. Sitä, miten vaatimukset ja ohjelmisto loppujen lopuksi luodaan, ei pidetä yhtä tärkeänä kuin sitä, miten niiden vastaamista todellisiin tarpeisiin arvioidaan ja ylläpidetään. Lisäksi, koska eri ihmisillä on eri työskentelytavat ja prosessin käyttäjien arvioidaan olevan alansa ammattilaisia, ei pidetä järkevänä liian tarkkaa ohjeistusta siellä, missä eri työskentelytavoilla voidaan yhtä hyvin päästä samoihin lopputuloksiin. Tässä insinöörityössä kuitenkin mainitaan useita ehdotuksia.



Kuva 2. Tie todellisista tarpeista toteutettuun sovellukseen kulkee vaatimusten kautta.

Todellisten tarpeiden toteutumista ohjelmistossa pyritään varmistamaan valitulla joukolla käytäntöjä. Todelliset tarpeet on abstrakti konsepti, jonka sisällöstä ei voida koskaan olla täysin varmoja muuta kuin arvioimalla lopputulosta jälkikäteen. Tämä viittaa siihen, että vaikka sovellus tehtäisiin juuri asiakkaan pyytämällä tavalla, voi se jälkeenpäin osoittautua virheelliseksi, koska todellisista tarpeista ei ollut tarkkaa ymmärrystä. On hyödyllistä tunnistaa, että kerätyt vaatimukset eivät välttämättä vastaa todellisia tarpeita.

Tärkein prosesseja ohjaava käytäntö tulee Dave Thomasin 2014 kirjoittamasta artikkelista. Siinä Dave Thomas määrittelee ketterän ohjelmistotuotannon perusteet hyvin yksinkertaisesti. Hänen mielestään ketterän ohjelmistotuotannon askeleita ovat seuraavat:

- Arvioi, missä olet tällä hetkellä.
- Ota pieni askel kohti päämäärää.
- Säädä ymmärrystäsi oppimasi mukaisesti.
- Toista.

Dave Thomas myös selventää, että jos kohdataan kaksi tai useampi samanarvoista askelta, valitaan se, joka on tulevaisuudessa helpompi muuttaa. Nämä asiat ovat hänen mielestään koko ketterän ohjelmistotuotannon perusta. [Dave Thomas 2014.]

Dave Thomasin artikkelia vastaavia ajatuksia löytyy myös muualta, esimerkiksi ns. Spotifyn malli ja Lean-ajattelumalli. Näissä iteraatio koostuu suunnittelusta, rakennuksesta, julkaisusta ja muokkauksesta. Sovelluksesta yritetään luoda nopeasti toimiva versio, josta voidaan kerätä asiakaspalautetta seuraavaa versiota varten. Jokainen versio on pieni askel, jota käytetään säätämään ymmärrystä asiakkaiden todellisista tarpeista. Perusidea on sama: nämä mallit ovat vain huomattavasti kehittyneempiä ja yksityiskohtaisempia. [Bank 2014.]

3 Prosessin kehitysvaiheet

3.1 Vaatimusten hallinta

Vaatimusten hallinta on tässä kuvattavan prosessin ensimmäinen vaihe, mutta sen pääasiallinen tehtävä on pitää kerättyjä vaatimuksia mahdollisimman lähellä todellisia tarpeita koko ohjelmiston elinkaaren ajan ja reagoida muutoksiin. Vaatimusten hallinta ei siis ole tehtävä, joka suoritetaan vain projektin alussa, vaan jatkuvasti käytössä oleva prosessi [TOGAF 2018]. Kun uutta tietoa tulee saataville, on vaatimuksia usein syytä päivittää. Vaatimusten hallintaan kuuluu tässä tapauksessa myös vaatimusten kehittäminen, vaikka joissain määritelmissä näin ei olekaan.

3.1.1 Vaatimuksen määritelmä

Ohjelmistotuotannossa vaatimus voidaan määritellä monella tapaa. Vaatimuksella voidaan viitata mihin tahansa kriteeriin, joka ohjaa suunnittelua. Tarkemmin vaatimuksella voidaan viitata määrittelyyn siitä, mitä piirteitä ja ominaisuuksia järjestelmään tullaan toteuttamaan. Ne siis kuvaavat, miten järjestelmän tulisi toimia ja mitä piirteitä sen tulisi esittää. [Wiegers & Beatty 2013.]

Yleisen määritelmän lisäksi tässä insinöörityössä vaatimukset yhdistetään käsitteellisesti asiakkaan ja käyttäjän todellisiin tarpeisiin. Niillä siis pyritään kuvaamaan todelliset tarpeet sellaisella tasolla, että niitä vastaava ratkaisu voidaan teknisesti toteuttaa. Kuten kuvassa 2 ilmaistiin, ne toimivat siltana todellisista tarpeista toteutettuun ohjelmistoon. Teoriassa vaatimukset eivät itsessään tuo lisäarvoa, koska ne eivät suoraan mahdollista asiakasta tekemään niitä tehtäviä, jotka tuovat hänelle lisäarvoa. Kuitenkin käytännössä vaatimusta voidaan pitää tarpeellisena käsitteenä, koska se mahdollistaa niiden lähtökohtien ilmaisemisen, johon koko kehitystyö ja sen arvo perustuu.

Taulukossa 1 määritellään tässä insinöörityössä käsiteltäviä vaatimustyyppejä. Vaatimustyypit on valittu ”Software Requirements” -kirjasta [Wiegers & Beatty 2013]. Näistä vaatimustyypeistä toiminnallisia ja ei-toiminnallisia vaatimuksia pidetään teknisen näkökulman vaatimuksina, eli ne kuvaavat tarvittavan toteutuksen yksityiskohtia. Muut vaati-

mustyypit toimivat korkeammalla tasolla ja keskittyvät siihen, miksi järjestelmälle halutaan tiettyjä toiminnallisuuksia ja piirteitä. Ne toimivat myös toiminnallisten ja ei-toiminnallisten vaatimusten lähteinä ja tulevat yleensä suoraan käyttäjiltä, yritystoiminnan tarpeista tai muilta sidosryhmiltä. Testitapausta ei ajatella vaatimuksen tyyppinä vaan kehittyneempänä vaatimuksen esitysmuotona, joka sisältää tavan, jolla vaatimus voidaan validoida.

Taulukko 1. Erityyppisiä vaatimuksia [Wiegers & Beatty 2013: 7].

Vaatimus	**Määritelmä**
Liiketoiminnallinen vaatimus tai sääntö	Ohjelmistoa rakentavan tai sen tilanneen organisaation korkeantason tavoite, tehtävä tai sääntö (miksi ja mitä halutaan luoda)
Rajoite	Suunnittelua, rakennusta tai teknologiaa rajoittava asia, joka täytyy huomioida
Käyttäjän vaatimus	Tavoite tai tehtävä, jota käyttäjän täytyy pystyä suorittamaan järjestelmällä (mitä käyttäjä haluaa tehdä)
Toiminnallinen vaatimus	Kuvaus järjestelmän toiminnasta tietyissä olosuhteissa (järjestelmän käyttäytyminen)
Ei-toiminnallinen vaatimus (laadullinen vaatimus)	Kuvaus piirteestä, joka järjestelmän täytyy esittää (miten ja miten hyvin järjestelmä toimii)

Taulukossa 1 esitetyt vaatimustyypit kuvaavat eri vaatimusten lähteitä ja sitä, miten vaatimuksia kerättäessä tulee ottaa huomioon toimintojen lisäksi myös muut piirteet. Näiden pohjalta voidaan ajatella eri vaatimuksen kehitystapoja. Todellisuudessa eri vaatimustyyppejä on kuitenkin äärettömästi ja siitä syystä niihin ei ole syytä paneutua liian tarkkaan. Tärkeintä on luoda riittävästi käsitteitä, joita voidaan käyttää ajattelun apuna.

Vaatimuksia voidaan myös arvioida tuotelaadun näkökulmasta; tämä antaa varsinkin ei-toiminnallisille vaatimukset termille tarvittavaa kontekstia. Jos käytetään ISO/IEC 25010 [2011] -luokittelua, toiminnallinen sopivuus voidaan liittää toiminnallisiin vaatimuksiin. Tehokkuus, yhteensopivuus, käytettävyys, luotettavuus, turvallisuus, ylläpidettävyys ja siirrettävyys voidaan liittää ei-toiminnallisiin vaatimuksiin.

3.1.2 Vaatimusten hallinnan periaatteet

Vaatimusten hallinnassa on hyvä noudattaa yhden lähteen periaatetta; hyväksytyillä vaatimuksilla tulisi olla yksi ensisijainen tallennuspaikka, jonka sisältö kertoo vaatimusten todellisen tilan. Yhdellä pääasiallisella lähteellä vältetään erilaisten käsitysten sotkeutuminen, koska pääasiallista lähdettä voidaan käyttää ratkaisemaan ristiriidat. Yhden lähteen periaatteella voidaan vähentää ylläpitotaakkaa, jäljitettävyystaakkaa ja parantaa yhdenmukaisuutta [Ambler 2005]. Tämän pääasiallisen lähteen muuttamiselle on syytä määritellä prosessi siten, että muutos tapahtuu vain silloin, kun sen katsotaan lisäävän arvoa kokonaisuudessaan (huomioiden esimerkiksi siitä aiheutuva työmäärä).

Vaatimusten hallinnalla voidaan ajatella olevan järjestelmäkohtaisesti kolme mahdollista tilaa. Projekti alkaa yleensä (pois lukien tilanteet, joissa vaatimukset on asetettu etukäteen) vapaassa konseptointitilassa. Tällä tarkoitetaan vaatimusten hallinnan näkökulmasta sitä, että vaatimuksia ei ole vielä hyväksytty ja niitä voidaan vapaasti muuttaa. Vaikka myös konseptointitilassa tehdään prototyyppejä olettamuksien testaamiseen, tulee järjestelmän lopullinen toteutus aloittaa ennemmin tai myöhemmin. Toteutuksen aloittamiseen on hyödyllistä olla ensimmäinen hyväksytty versio vaatimuksista, jonka kaikki projektin jäsenet tiedostavat. Tämä hyväksytty versio voi toimia myös lupauksena asiakkaalle toimitettavista ominaisuuksista (versio saa muuttua, mutta myös muutokset tulee hyväksyä). Hyväksyttyä versiota pidetään vaatimusten perustasona [Wiegers & Beatty 2013: 459]. Lopuksi, kun järjestelmä lähestyy julkaisua, voidaan vaatimukset asettaa julkaisutilaan eli jäädyttää kyseistä versiota koskevat vaatimukset kieltämällä kaikki muutokset. Tämä tehdään siksi, että voidaan keskittyä julkaisun viimeistelyyn. Myös yksittäiset vaatimukset voidaan nähdä olevan jossain näistä tiloista. Lisäksi yksittäiset vaatimukset voidaan hylätä tai siirtää myöhempään julkaisuun. Näiden tilojen välillä liikkumiseen tarvitaan muutoksenhallintaprosessi.

Lopuksi vaatimusten hallintaan kuuluu myös vaatimusten tilan seuranta. Tulee olla paikka, josta voidaan seurata, onko vaatimus konseptoinnissa, hylätty, siirretty seuraavaan versioon, hyväksytty vai valmis. Näin voidaan helpommin arvioida työn edistymistä vaatimusten näkökulmasta. [Wiegers & Beatty 2013: 465.]

Yhteenvetona vaatimuksen hallinnan prosessissa tullaan toteuttamaan

- yksi, ensisijainen vaatimusten lähde
- vaatimusten perustaso ja muutoksen hallinta
- vaatimusten tilanseuranta.

3.1.3 Vaatimusten kehittäminen

Vaatimukset eivät usein tule valmiina asiakkaalta, vaan ne joudutaan kehittämään epäselvistä toiveista ja mainitsemattomista tottumuksista. Vaatimuksien keräämiseen asiakkaalta ja muilta sidosryhmiltä on olemassa useita menetelmiä [Wiegers & Beatty 2013: 121]. Näitä ei kuitenkaan tässä insinöörityössä käsitellä. Kun vaatimukset on kerätty, on niitä syytä kehittää toteutuksen kannalta hyödylliseen muotoon. Erityyppisiä vaatimuksia kehitetään eri menetelmillä.

Liiketoiminnalliset vaatimukset

Liiketoiminnalliset vaatimukset ohjaavat monesti muuta vaatimusten hallintaa. Voidaan ajatella, että tietotekniikka ja sitä kautta ohjelmistot ovat olemassa vain ja ainoastaan liiketoiminnallisten lähtökohtien tukemiseksi (todelliset tarpeet). Ohjelmistokehityksen näkökulmasta ne kertovat, liittykö esimerkiksi jokin toiminnallinen vaatimus oikeasti ratkottavaan ongelmaan ja tuoko se lisäarvoa rajoitetut resurssit huomioitaessa. Tavoitteiden ja tehtävien lisäksi liiketalouden puolelta vaikuttaa myös toimialan ja organisaation säännöt [Wiegers & Beatty 2013: 78]. On oleellista, että liiketoiminnan ääntä kuunnellaan myös ohjelmistotuotannossa ja että projekteissa on mukana myös liiketoiminnallisen puolen asiantuntija. Tässä insinöörityössä oletetaan, että liiketaloudelliset vaatimukset ovat olemassa, mutta niiden kehitysprosessia ei selitetä. Liiketoiminnallisten vaatimusten lisäksi voi olla muita rajoitteita [Wiegers & Beatty 2013: 89]. Rajoitteita voidaan pitää päätöksinä, jotka on tehty ja joita ei voi muuttaa.

Liiketoiminnalliset vaatimukset löytyvät usein visiodokumentista [Wiegers & Beatty 2013: 81]. Ne voivat olla taloudellisia esimerkiksi ”vähennä tukikustannuksia 50 %, 12 kuukaudessa” tai ei-taloudellisia esimerkiksi ”kehitä laajennettavissa oleva alusta, joka tukee useamman tuotteen perhettä.” Kuten on jo mainittu, ne voivat olla myös mitä tahansa muita rajoitteita, jotka syntyvät liikatoiminnallisista lähtökohdista esimerkiksi ”Järjestelmän tulee läpäistä kaikki tietoturvatestit ennen julkaisua.”

Käyttäjän vaatimukset

Käyttäjän vaatimukset antavat ensimmäisen korkeatasoisen kuvan siitä, mitä tehtäviä käyttäjä haluaa järjestelmällä suorittaa [Wiegers & Beatty 2013: 144]. Niiden kehittämisessä on syytä edetä iteratiivisesti pienin askelin. Ensimmäisenä voidaan käyttää liiketaloudellisia vaatimuksia ja käyttäjien mielipiteitä löytämään ne yksinkertaisimmat tehtävät, joilla määritellyt tavoitteet voitaisiin täyttää.

Sen jälkeen löydettyjä tehtäviä voidaan analysoida etsimällä vastausta kysymykseen ”onko tehtävä mahdollista teknisesti toteuttaa huomioiden budjetti, halutut laadulliset piirteet ja muut rajoitteet?” Jos vastaus on ei, viedään työ takaisin liiketaloudellisten vaatimuksien määrittelyyn tai heikennetään laadullisia vaatimuksia. Jos vastaus on ”emme tiedä”, siirrytään kuvaamaan ja suunnittelemaan näitä yksinkertaisia tehtäviä tarkemmin, jotta saataisiin kerättyä lisää tietoa. Jos aiemman kokemuksen perusteella vastaus on ”kyllä”, voidaan siirtyä joko tarkempaan kuvaukseen tai muiden vaatimuksien keräykseen tarpeen mukaan. Tällaisessa analyysissä pyritään selvittämään vaatimuksen toteutettavuus (requirement feasibility) [Wiegers & Beatty 2013: 50]. Yksinkertaisimmillaan tämä tarkoittaa käytetyn teknologian tuntevien henkilöiden mielipiteiden kuuntelemista.

Käyttäjän vaatimuksien kuvaukseen on kaksi yleistä tapaa: käyttötapaukset ja käyttäjätarinat [Wiegers & Beatty 2013: 143]. Käyttötapaukset koostuvat käyttäjäskenaarioista, eli yhdestä tai useammasta ketjusta toimenpiteitä käyttäjän ja järjestelmän välillä. Tässä insinöörityössä suositaan yhden polun epämuodollista käyttäjäskenaariota. Käyttötapauksien tarkempaa kuvausta tehdään vain välittömän tarpeen mukaan; tällä vähennetään tulevan muutoksen aiheuttamaa kustannusta. Kuvauksessa valitaan yksi ketju (yleensä tyypillinen työnkulku) ja kuvataan tämä luonnollisella kielellä (mitä toimijat tekevät). Ketjun kehityksessä on luontevaa käyttää hyödyksi konkreettisia esimerkkejä (mielikuvia oikeasta käyttäjästä tekemässä oikeaa tehtävää). Kun käyttäjäskenaarioon ollaan tyytyväisiä, voidaan se formalisoida: formalisointi tapahtuu esimerkiksi lisäämällä tunniste, luomalla prosessikaavio tai tekemällä käyttäjäskenaariosta täysi tilannekuvaus/käyttötapaus. Kun tämä ensimmäinen polku on kuvattu, voidaan tarvittaessa kuvata muita vaihtoehtoisia työnkulkuja samaan tapaan.

Epämuodollinen käyttäjäskenaario voi olla esimerkiksi ”Ostaja tilaa tuotteen. Järjestelmä tarkistaa, onko tuotetta saatavilla. Tuotetta on saatavilla, joten järjestelmä tallentaa tilauksen ja vähentää tuotteiden määrää tilatulla määrällä. Järjestelmä ilmoittaa ostajalle, että tilaus on tehty.” Siinä siis annetaan yksinkertaisesti kuvaus siitä, mitä eri toimijat tekevät tilanteessa, jossa kaikki etenee kuten pitääkin. Tästä vaihtoehtoinen työnkulku voisi olla esimerkiksi ” Ostaja tilaa tuotteen. Järjestelmää tarkistaa, onko tuotetta saatavilla. Tuotetta ei ole saatavilla, joten järjestelmä ilmoittaa ostajalle, että tilausta ei voi tehdä, koska tuote on loppu.” Muodollinen/formalisoitu käyttötapaus yhdistää nämä työnkulut, esimerkiksi taulukossa 2 esitetyllä tavalla.

Taulukko 2. Esimerkki käyttötapauksesta. Sisältöä ja muotoa muokattu [Wiegers & Beatty 2013: 150].

Id ja nimi	UC-4 Tilaa tuote
Luoja ja luontipäivä	Tuomas Maanonen 16.4.2020
Toimijat	Ostaja, järjestelmä
Laukaisu	Ostaja ilmoittaa, että haluaa ostaa tuotteen.
Esiehdot	1. Ostaja on kirjautunut sisään
Jälkiehdot	1. Ostajalle on ilmoitettu toimenpiteen lopputulos. 2. Tilaus on joko tallennettu tai jätetty täysin huomioimatta, ilman sivuvaikutuksia.
Yleinen työnkulku	**4.0 Osta tuote, jota on saatavilla** 1. Ostaja tilaa tuotteen 2. Järjestelmä tarkistaa, onko tuotetta saatavilla. 3. Järjestelmä tallentaa tilauksen ja vähentää tilattujen tuotteiden määrää tilatulla määrällä. 4. Järjestelmä ilmoittaa ostajalle, että tilaus on tehty.
Vaihtoehtoiset työnkulut	**4.1 Osta tuote, jota ei ole saatavilla** 1. Ostaja tilaa tuotteen 2. Järjestelmä tarkistaa, onko tuotetta saatavilla. 3. Järjestelmä ilmoittaa ostajalle, että tilausta ei voida tehdä, koska tuote on loppu.

Kun käyttötapaukset nimetään ja niiden osat numeroidaan kuten taulukossa 2, niin hyötynä on se, että eri käyttötapauksiin ja niiden työnkulkuihin on helppo viitata muualta. Lisäksi lisätyt metatiedot kertovat esimerkiksi kuka käyttötapauksen on tehnyt, vielä vuosia itse toteutuksen jälkeen. Tämän kaiken lisääminen on kuitenkin huomattavasti työläämpää.

Ei-toiminnalliset vaatimukset

Ei-toiminnallisia vaatimuksia voidaan luokitella moniin eri kategorioihin. Yksi näistä on laadulliset piirteet [Wiegers & Beatty 2013: 262; Bass ym. 2012]. Tässä insinöörityössä keskitytään juuri laadullisten piirteiden löytämiseen ja kehittämiseen. Kehittäminen aloitetaan listaamalla kaikki mieleen tulevat laadulliset piirteet. Apuna listaamisessa voidaan käyttää jo aiemminkin mainittua ISO/IEC 25010 [2011] -luokittelua tai muiden lähteiden listauksia [Wiegers & Beatty 2013: 263; Bass ym. 2012]. Listauksesta poistetaan sidosryhmien avustuksella ne, joita ei pidetä oleellisina kyseiselle järjestelmälle [Wiegers & Beatty 2013: 264]. Pitämällä mielessä järjestelmän liiketaloudelliset vaatimukset kysytään jokaisen kohdalla, tuoko se projektiin lisäarvoa eli onko se yleensäkään tarpeellinen.

Jos kyseessä olisi esimerkiksi verkkosovellus, voisi lista tässä vaiheessa sisältää piirteet saatavuus, koskemattomuus, yhteentoimivuus, tehokkuus, skaalautuvuus, turvallisuus ja käytettävyys [Wiegers & Beatty 2013: 263].

Tämän jälkeen jäljelle jäävät laadulliset piirteet on syytä priorisoida. Priorisointi tapahtuu asettamalla piirteet vastakkain; kumman piirteen puuttuminen aiheuttaisi isomman riskin ja kumpi toisi järjestelmälle enemmän lisäarvoa. Voidaan myös kysyä, aiheuttaako kyseinen laadullinen piirre esteitä muiden ominaisuuksien onnistuneelle toteutukselle. Priorisoinnin tarkoitus on antaa kuva siitä, mihin kannattaa ja mihin ei kannata kuluttaa resursseja. Sillä maksimoidaan keskittyminen juuri tärkeimpiin laadullisiin näkökulmiin. [Wiegers & Beatty 2013: 264.]

Lopuksi laadullisista ominaisuuksista rakennetaan tarkempia ei-toiminnallisia/arkkitehtuurillisia vaatimuksia. Myös näiden tulisi olla konkreettisia ja testattavissa ollakseen hyödyllisiä; tähän pyritään niiden tarkemmassa kuvauksessa. Todellisuudessa tämä toimenpide suoritetaan arkkitehtuurisuunnittelussa ja siihen löytyy ohjeistusta esimerkiksi ”Software Architecture in Practice” -kirjasta [Bass ym. 2012]. Kirjasta löytyy piirteiden huomiointia varten esimerkiksi tarkastuslistoja. Jokaiselle laadulliselle piirteelle tarvitaan oma kehitystapa, joten työ on melko vaativaa. Esimerkiksi pelkästään turvallisuudelle löytyy neljä eri kategoriaa, joista voidaan lisätä turvallisuuden kannalta oleellisille komponenteille vastuita. Kategoriat ovat havaitse hyökkäyksiä, vastusta hyökkäyksiä, reagoi hyök-

käyksiin ja palaudu hyökkäyksistä [Bass ym. 2012]. Näistä esimerkiksi vastusta hyökkäyksiä kategorian alle kuuluu toimijoiden tunnistamiseen, todentamiseen, valtuuttamiseen ja tiedon salaukseen liittyvät vaatimukset. Turvallisuuteen liittyviä konkreettisia vaatimuksia voisi siis yksinkertaisimmillaan olla käyttäjätietojen salaaminen turvalliseksi todetulla salausmenetelmällä tai tiettyjen toimintojen käytön estäminen ilman kirjautumista.

Toiminnalliset vaatimukset

Toiminnallisten vaatimusten sisältö vastaa samaa, mitä aikaisemmin mainituista vaatimustyypeistä pystyy päättelemään. Tässä insinöörityössä konseptin pääasiallinen tarkoitus on esittää, että tieto tulee kehittää ja ilmaista myös sellaisella tasolla, jolla kehittäjä sen pystyy käytännössä toteuttamaan. Toiminnallisia vaatimuksia kuvataan perinteisesti tyyliin ”[Toimija] pystyy [tekemään jotain] [asialle] [tilanteessa missä]” muodossa [Wiegers & Beatty 2013: 208]. Aiempaa verkkokauppa esimerkkiä hyödyntäen, toiminnallinen vaatimus voisi olla esimerkiksi ”Ostaja pystyy, tilaamaan uudelleen aiemmin tilaamansa tuotteen, valitsemalla sen tilaushistoriastaan.” Perinteistä vaatimusten esittämistapaa voi käyttää, jos jokin asia halutaan dokumentoida, eikä sitä pysty ilmeisesti päättelemään muun tyyppisistä vaatimuksista. Tällaisen esitystavan hyödyt tulevat esiin enimmäkseen silloin, kun järjestelmän vaatimukset tulee esittää ulkopuoliselle organisaatiolle. Toiminnalliset vaatimukset voidaan myös kuvata informaalisti arkkitehtuurisuunnitteluvaiheessa osana itse arkkitehtuuria [Bass ym. 2012].

Testitapaukset

Vaatimusten kehityksen piiriin voidaan lisätä myös testitapaukset. Testitapaukset ovat ylimääräinen kehityksen askel, jossa myös vaatimuksen validointi kuvataan. Testitapaukset ovat aina osa muita vaatimuksia. Testitapauksiin kuuluu yleisesti samoja asioita kuin käyttötapauksiin; oleellisimpana lisäyksenä ovat hyväksymiskriteerit, eli mitattavissa olevat asiat, joiden täytyy toteutua, kun käyttötapaus suoritetaan [Choudary 2019]. Testitapauksien määrittely aloitetaan kopioimalla tarvittava tieto vastaavasta käyttötapauksesta määrittelemällä hyväksymiskriteerit ja tarvittaessa esittämällä testin suoritus menetelmä. Muiden kuin käyttäjävaatimusten muuttamista testattaviksi ei käsitellä tässä insinöörityössä. Yleisesti tutkivassa testauksessa esitystapa on vapaa; esimerkiksi käytet-

tävyystestien ja tietoturvatestien esitykseen on luontevaa käyttää omaa formaattia. Lisäksi formaaleja määrittelykieliä voidaan käyttää. Oleellista on, että testausmenetelmä, sen puutteet ja testauksen lopputulos ovat selviä. Testitapauksia luotaessa tulee myös mietittyä, onko kyseisessä vaatimuksessa ylipäänsä mitään järkeä. Testitapauksia tai muuta sen tapaista dokumentaatiota ei tarvitse tehdä yksikkötesteille tai integraatiotesteille, jotka tullaan suorittamaan automaattisesti. Testitapaukset kuvaavat nimenomaan hyväksymistestauksessa suoritettavia testejä.

3.2 Arkkitehtuurisuunnittelu

Pelkästään vaatimusten kehittäminen eikä hallinta vielä takaa niiden toteutumista käytännön sovelluksessa; tähän pyritään arkkitehtuurisuunnittelulla. Arkkitehtuurisuunnittelun pääasiallinen tehtävä on kehittää vaatimukset toiminnallisiksi piirteiksi, luoda niitä tukeva sovellusrakenne ja ohjata sovelluskehitystä sellaisella tavalla, että vaatimukset otetaan siinä huomioon. Sovellusarkkitehtuuri on sovellusarkkitehtuurisuunnittelun lopputulos ja niinpä siihen kuuluu muun muassa sovelluksen rakenne (käytetyt arkkitehtuurimallit), halutut laadulliset piirteet, arkkitehtuurilliset päätökset (pakolliset rajoitteet) ja suunnittelu-/rakennusperiaatteet (ohjeet kehittäjille) [Ford & Richards 2020]. Se on siis sekä suunnitelma että kehitysohje samanaikaisesti.

3.2.1 Arkkitehtuurisuunnittelun mallit ja menetelmät

Tämän insinöörityön arkkitehtuurisuunnittelu on yhdistelmä ADD-menetelmää, Simon Brownin C4-mallia ja C4-mallin inspiraationa toiminutta Philippe Kruchtenin 4 + 1 -arkkitehtuurimallia [Bass ym. 2012; Brown 2019; Kruchten 1995].

ADD-menetelmä tulee sanoista attribute-driven design method, joka kuvaa suunnitelman pohjautumista laadullisiin piirteisiin tai attribuutteihin. ADD-menetelmä sisältää viisi vaihetta, joihin tämän insinöörityön arkkitehtuurisuunnittelun vaiheet perustuvat. Nämä vaiheet ovat seuraavat:

1. Valitse suunniteltava elementti.
2. Tunnista arkkitehtuurillisesti merkittävät vaatimukset.
3. Luo suunnitelma kyseiselle elementille.

4. Kerää jäljelle jääneet vaatimukset.
5. Valitse oleelliset vaatimukset seuraavaan iteraatioon ja toista.

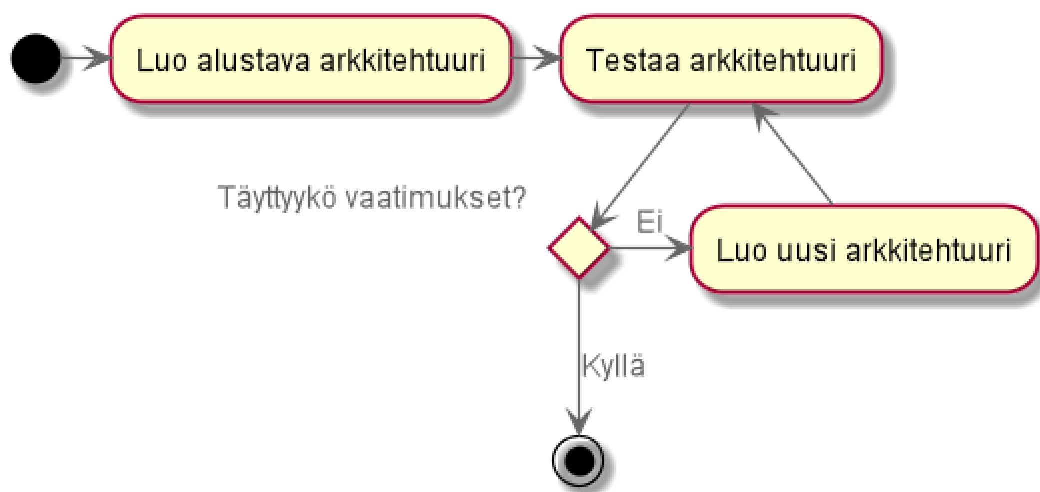
Ideana menetelmässä on luoda järjestelmän yhdelle osiolle/elementille toimiva arkkitehtuuri nopeasti, jotta se voidaan antaa heti sitä tarvitseville kehittäjille. Tämän saavuttamiseksi järjestelmä jaetaan ensimmäisenä useampiin erillisiin palasiin. Lisäksi menetelmässä painotetaan luo ja testaa -menetelmää, jonka pohjana toimivat arkkitehtuurillisesti merkittävät vaatimukset. [Bass ym. 2012.]

Philippe Kruchtenin 4 + 1 -arkkitehtuurimalli keskittyy siihen, miten arkkitehtuuria tulisi kuvata. Siinä arkkitehtuuria kuvataan neljästä eri näkökulmasta, jotka ovat kehittämisnäkökulma, looginen näkökulma, prosessinäkökulma ja fyysinen näkökulma. Lisäksi näitä neljää näkökulmaa valaistaan käyttöskenaarioilla, joihin näkökulmien kuvaukset myös osittain perustuvat. Täten mallin nimessä (4 + 1) neljä viittaa näkökulmiin ja yksi käyttöskenaarioihin. Kyseinen useamman näkökulman idea löytyy tavalla tai toisella myös muista malleista ja menetelmistä. [Kruchten 1995.]

C4-mallin keskiössä on tapa kuvata kehittämisnäkökulmaa. Kehittämisnäkökulmassa muun muassa jaetaan järjestelmä useampiin erillisiin palasiin, joka tukee esimerkiksi rinnakkaiskehitystä. C4-malli sopii molempiin aikaisemmin kuvattuihin malleihin, mutta kuvaustavan yksityiskohdat eroavat esimerkiksi 4 + 1 -mallista. C4-mallin nimi tulee siitä, että kehittämisnäkökulmaa kuvataan neljältä eri tasolta, joiden nimet alkavat englanniksi C-kirjaimella. Suomeksi nimet ovat konteksti (context), säiliöt (containers), komponentit (components) ja koodi (code). C4-mallin ideana on se, että jokaisessa kuvaustasossa katsotaan järjestelmää hieman lähempää ja yksityiskohtaisemmin kuin edellisellä tasossa, käyttäen samoja abstraktioita/symboleita. Ylemmän tason kuvaksista jätetään pois teknologian yksityiskohdat, jotta ne olisivat helpommin kaikkien ymmärrettävissä. Mallissa myös esitetään tapa kuvata nämä abstraktiot, vaikka kuvaustavan käyttöä ei velvoitetakaan. Insinöörityön kuvan 4, 5 ja 6 kaavioissa on hyödynnetty vastaavaa kuvastapaa. Kuvaustavassa jokaisella abstraktiolla/symbolilla on nimi, kuvausteksti ja sulkeissa käytetyt teknologiat. [Brown 2019.]

3.2.2 Arkkitehtuurisuunnittelun vaiheet

Tässä insinöörityössä arkkitehtuurisuunnittelu jaetaan neljään vaiheeseen: tunnista, valitse, luo ja validoi. Prosessissa noudatetaan luo ja testaa -periaatetta, jossa luotu arkkitehtuuri toimii vaatimukset toteuttavana hypoteesina, jota yritetään todeta vääräksi keksimällä ja testaamalla ongelmakohtia [Bass ym. 2012]. Tämä käytännössä tarkoittaa sitä, että arkkitehtuurisuunnittelu koostuu pääpirtein arkkitehtuurin luonnin ja validoinnin iteraatioista kuvan 3 esittämällä tavalla.



Kuva 3. Arkkitehtuurisuunnittelussa käytetty luo ja testaa -iteraatio.

Luonnollisesti arkkitehtuurisuunnittelu tulee aloittaa valitsemalla suunniteltava komponentti ja tunnistamalla sitä koskevat vaatimukset. Tässä insinöörityössä tätä kutsutaan tunnistusvaiheeksi. Komponenttia koskevia vaatimuksia kutsutaan arkkitehtuurillisesti merkittäviksi vaatimuksiksi [Bass ym. 2012]. On hyödyllistä mainita, että ensimmäinen suunniteltava ”komponentti” on koko järjestelmä [Bass ym. 2012]. Käytännössä tämä viittaa järjestelmän luurangon suunnitteluun. Vasta kun luuranko on suunniteltu, voidaan sen sisällöstä valita yksittäinen säiliö tai komponentti suunniteltavaksi. Kuten luontivaiheessa tullaan näkemään, C4-mallin konteksti ja säiliötasot keskittyvät juuri koko järjestelmän kuvaamiseen. Vasta komponentti- ja kooditasoilla joudutaan valitsemaan, mikä säiliö tai komponentti halutaan kuvata ensin. Tähän valintaan vaikuttaa muun muassa säiliöön tai komponenttiin liittyvä riski ja epävarmuus [Bass ym. 2012].

Valintavaihe

Valintavaihe on luonnin ja validoinnin sykliä tukeva vaihe, jossa valitaan arkkitehtuurisuunnittelussa käytettävät resurssit. Komponenttiin vaikuttavien vaatimusten perusteella valitaan rakenteessa noudatettavia arkkitehtuurimalleja ja luodaan kehitystyössä noudatettavia rajoitteita, joiden uskotaan tukevan ei-toiminnallisten vaatimusten toteutumista [TOGAF 2018]. Kehitystyön ja arkkitehtuurin rajoitteet tulevat osittain projektin ja organisaation yleisistä rajoitteista (esimerkiksi etukäteen valitut teknologiat). Lisäksi on tarpeen miettiä, kuka arkkitehtuuria tarvitsee, ja valita, millä tavalla arkkitehtuuria on järkevä esittää. Arkkitehtuuri kuvataan aina kehittämisnäkökulmasta, mutta joskus C4-mallin kooditasoa tehtäessä, nähdään tarpeellisena esittää arkkitehtuuri myös muista näkökulmista.

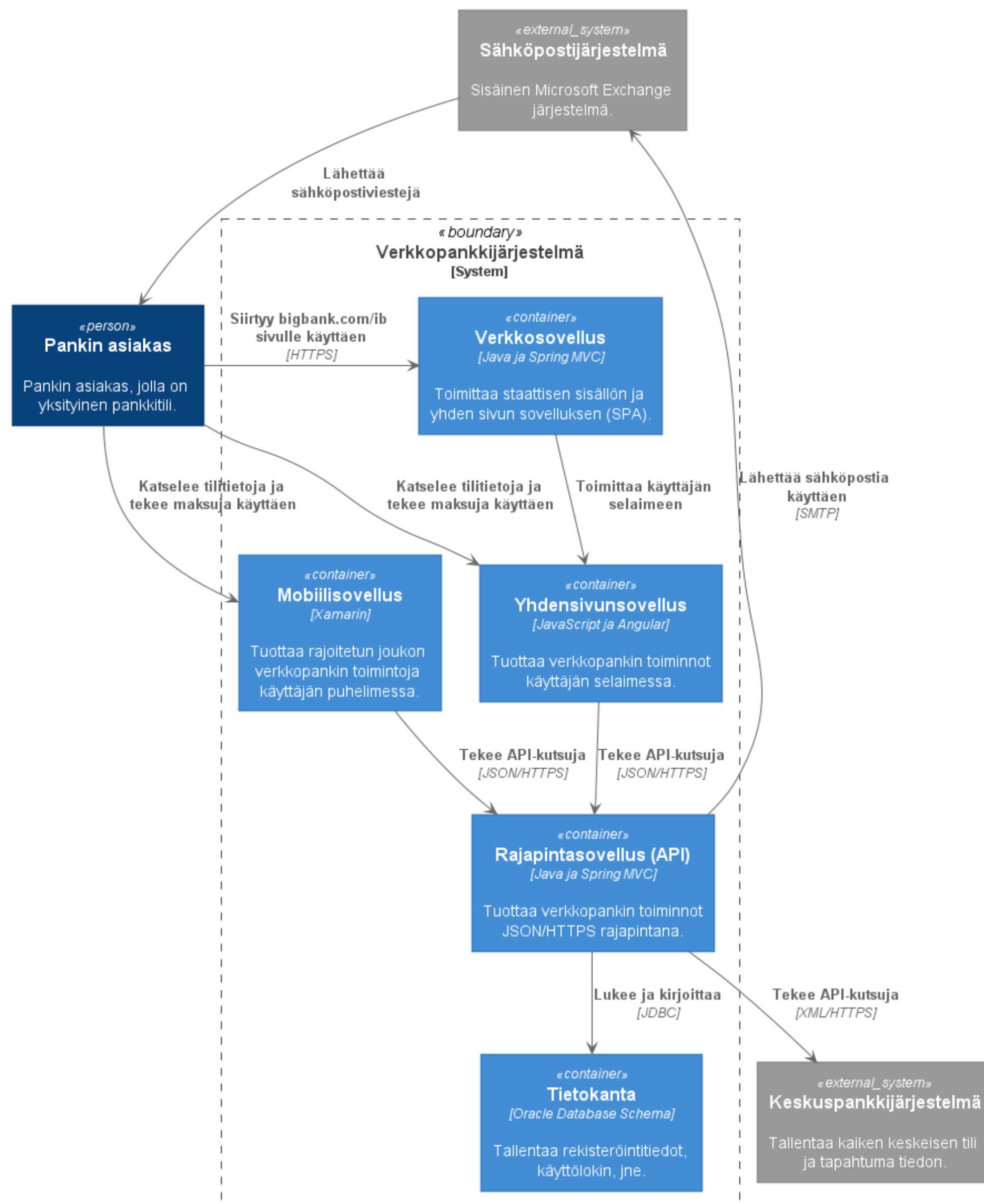
Kaikki arkkitehtuurisuunnitteluun liittyvät resurssit kerätään arkkitehtuurisäiliöön [TOGAF 2018]. Arkkitehtuurisäiliö on yksinkertaisesti ennalta määritelty tallennuspaikka näille resursseille; sitä voidaan hyödyntää melko vapaasti. Oleellista arkkitehtuurisäiliössä on se, että organisaation arkkitehtuurisuunnitteluun liittyvät ohjeet, käytännöt, standardit, vanhat arkkitehtuurit ja tällä hetkellä käytössä olevat arkkitehtuurit, ovat kaikki tallennettu ja saatavilla samasta paikasta. Arkkitehtuurisäiliön ja valintavaiheen avulla organisaation arkkitehtuurillinen osaaminen kasvaa, vaikka arkkitehdit vaihtelisivat. Lisäksi uuden arkkitehtuurin rakentamista ei koskaan tarvitse aloittaa tyhjästä.

Luontivaihe

Luontivaiheen keskiössä on 4 + 1 -mallin kehittämisnäkökulma [Kruchten 1995]. Kehittämisnäkökulman kuvaustapa otetaan C4-mallista [Brown 2019]. Tässä insinöörityössä keskitytään yksittäisen järjestelmän suunnitteluun, joten jokaisen kaavion keskiössä on sama järjestelmä tai järjestelmän osio.

Kuva 4. Esimerkki C4-mallin järjestelmän kontekstin/ympäristön kuvauksesta [Brown 2019].

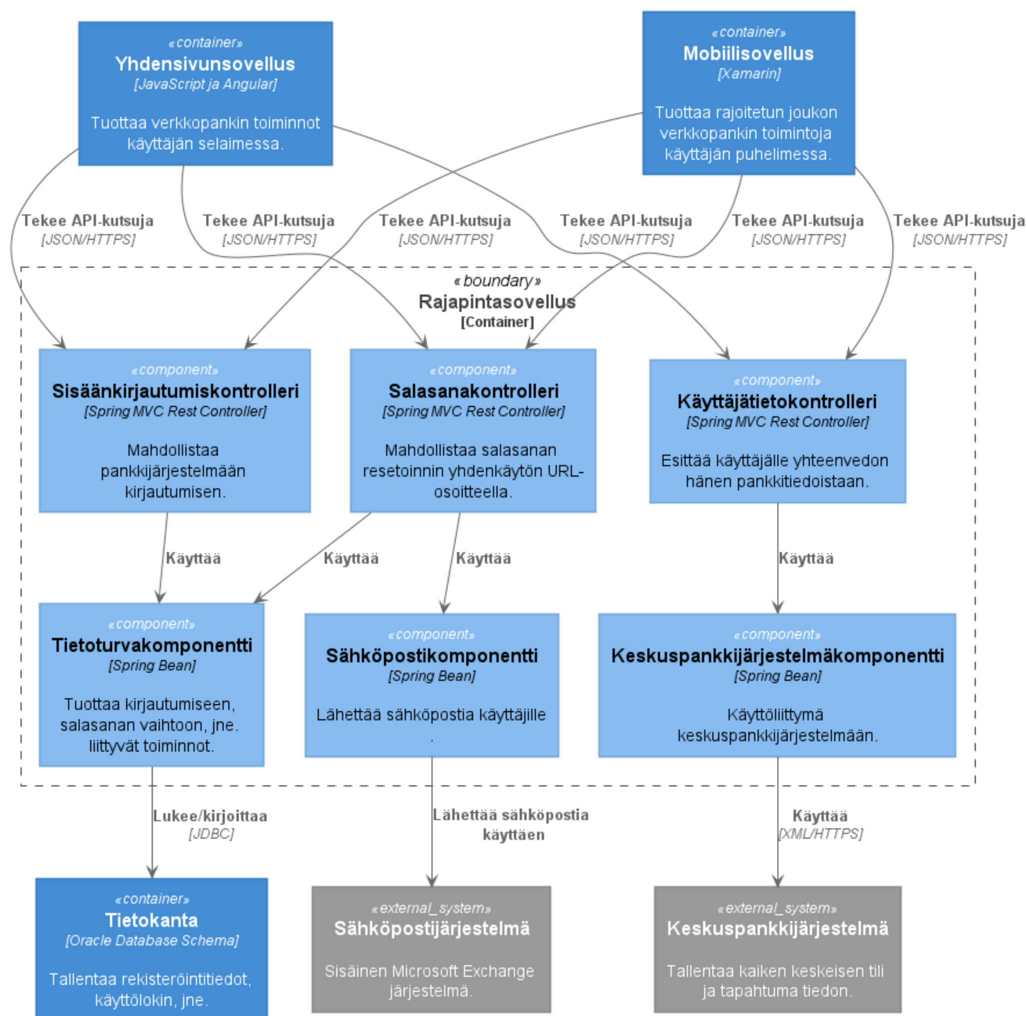
Kuva 4 esittää tavan kuvata järjestelmän toimintaympäristö/konteksti. Tämä on ensimmäinen kuvaustaso, jota käytetään, kun aletaan suunnittelemaan uutta järjestelmää. Kontekstissa kuvataan järjestelmän käyttäjät ja järjestelmän tärkeimmät riippuvuudet (ulkoiset järjestelmät). Lisäksi kuvataan näiden toimijoiden suhteet ja tieto/toiminto, jota järjestelmä antaa tai saa toimijoilta. Tällä kuvastasolla vältetään määrittelemästä teknologisia yksityiskohtia. [Brown 2019.]



Kuva 5. Esimerkki C4-mallin järjestelmän sisäisten säiliöiden ja niiden suhteiden kuvauksesta [Brown 2019].

Kuvassa 5 pankkijärjestelmä on jaettu säiliöihin. Kuvassa on vielä mukana järjestelmän konteksti ja se miten nämä ulkoiset tekijät toimivat säiliöiden kanssa. Säiliöt ovat kokonaisuuksia, jotka ajetaan samalla fyysisellä laitteella samassa prosessissa tai ryhmässä prosesseja. Esimerkiksi mobiilisovellus ja tietokanta ovat eri säiliöitä. Tällä kuvaustasolla voidaan antaa teknologiaan liittyvää ohjeistusta. Esimerkiksi jos käytössä on teknologia, jota on pakko käyttää, voidaan se mainita oleellisessa säiliössä. [Brown 2019.]

Säiliökäsite löytyy nykyisin myös toteutustasolla käytettäessä esimerkiksi Docker-alustaa [What is a Container 2020]. Tällaisilla säiliöintiteknologioilla järjestelmän jakaminen säiliöihin onnistuukin hyvin luontevasti.



Kuva 6. Esimerkki C4-mallin säiliön komponenttien kuvauksesta [Brown 2019].

Kuvassa 6 pankkijärjestelmän rajapintasovellussäiliö on jaettu komponentteihin. Tämä vaihe on luontevaa tehdä kehittäjien kanssa siinä vaiheessa, kun säiliötä aletaan kehittämään. Komponentit ovat kokonaisuuksia, jotka kuuluvat yhteen ja joilla on yhteinen rajapinta. Komponenttien kuvauksessa esitetään komponenttien vastuut [Bass ym. 2012]. Mikäli mahdollista vastuut kirjataan toiminnallisina vaatimuksina, jotka arkkitehti päättelee ja kehittää säiliön arkkitehtuurillisista vaatimuksista. Komponentteja myös arvioidaan sen mukaan, mitkä laadulliset piirteet niitä erityisesti koskevat. Komponentit

voivat esimerkiksi toteuttaa roolia, jossa juuri suorituskykyyn liittyvät vaatimukset ovat erityisen oleellisia, tai ne voivat sisältää suoran asiakasrajapinnan, jolla on käytettävyyteen liittyviä vaatimuksia. Laadullisia piirteitä ja ei-toiminnallisia vaatimuksia voidaan tässä vaiheessa käyttää luomaan kehittäjien kanssa rajoituksia ja ohjeistusta, jolla pyritään takaamaan niiden toteutuminen alhaisella tasolla. Jos komponenttien määrälle ei löydy muita ohjaavia perusteita, voidaan järjestelmä jakaa komponentteihin sen mukaan, kuinka monta kehittäjää tai kehittäjäryhmää sitä rakentaa. Komponenttikarttaa voidaan käyttää jakamaan kehitystyö sujuvasti kehittäjäryhmien välillä, koska komponentit ovat pienimpiä mahdollisia osioita, joita voidaan pitää erillisinä kokonaisuuksina. Useamman kehittäjäryhmän rakentamat komponentit istuvat sulavasti kokonaisuuteen arkkitehtuurisuunnitelman kehystämänä.

Järjestelmän komponenttikuvauksen lisäksi voidaan myös luoda erikseen käyttöönottokuvaus, prosessikuvaus, datamalleja tai muita UML-kaavioita. Nämä kuuluvat C4-mallin kooditason kuvauksiin, eli niissä kuvataan toteutuksen yksityiskohtia. Näiden kaavioiden luonti riippuu tarpeesta ja niiden kuvaustapaa ei ole erikseen määritelty. UML-kaavioihin perustuva suunnittelu ja kehitys on parhaimmillaan kriittisissä järjestelmissä, koska sen notaatio on tarkemmin määritelty. Ainakin yhdessä tutkimuksessa on todettu, että UML (tietyllä tavalla käytettynä) parantaa esimerkiksi suunnittelua, mutta sen käyttö on kalliimpaa kuin epäformaalimpien menetelmien käyttö [Anda 2006].

Validointivaihe

Arkkitehtuurin tai sen osan valmistuttua siirrytään validointivaiheeseen. Arkkitehtuurin validointi on hyvin oleellinen mutta vaikea vaihe. Validoinnin tarve riippuu arkkitehtuurin ”uutuudesta” eli siitä, kuinka monta uutta, testaamatonta olettamusta arkkitehtuuri sisältää. Esimerkiksi jos arkkitehtuurissa on käytetty arkkitehtuurimallia, jonka tiedetään parantavan tiettyjä laadullisia piirteitä, voidaan tätä pitää riittävänä perusteena valinnalle ilman erillistä validointia. Samaan tapaan voidaan käyttää kokemuksia aikaisemmista järjestelmistä. Sen sijaan, jos arkkitehtuuri sisältää malleja tai päätöksiä, joiden ei tiedetä täyttävän niille asetettuja vaatimuksia, on niitä syytä pyrkiä analysoimaan ja validoimaan. Arkkitehtuurin analyysi on liian monimutkainen aihe esittää tässä insinöörityössä, mutta siihen löytyy lukuisia keinoja lähdeaineistosta [Bass ym. 2012]. Uudelle arkkitehtuurille pitää saada vertaisarvioita, eli analyysejä myös muilta kuin arkkitehdiltä itseltään [Bass

ym. 2012]. Jos analyysissä ei päästä riittävään varmuuteen siitä, että arkkitehtuuri toteuttaa sille asetetut vaatimukset, on syytä edetä sovelluskehitykseen mahdollisimman pienin askelin ja kerätä tarvittavaa tietoa arkkitehtuurin arvioimiseksi ja ymmärryksen säätämiseksi.

3.3 Sovelluskehitys

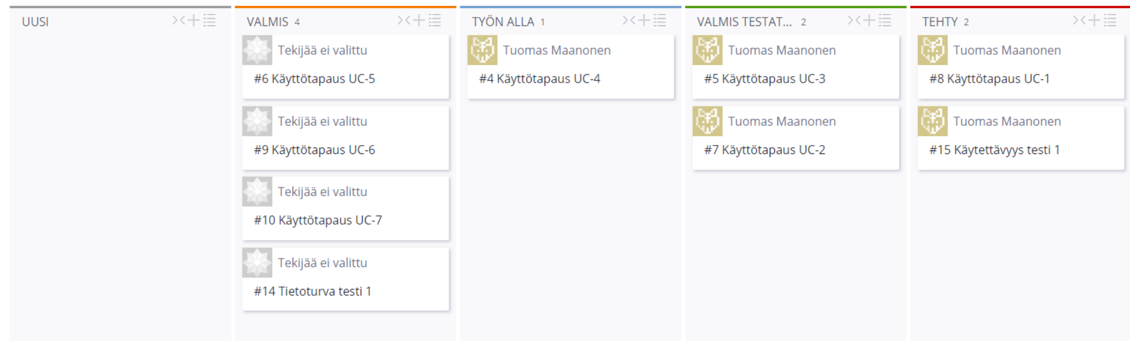
Sen jälkeen, kun arkkitehtuurisuunnittelussa saadaan valmiiksi suunnitelma ensimmäiselle säiliölle, voidaan siirtyä luomaan käytännön toteutusta. Prosessin tavoitteena on kuvata tapa toteuttaa sovellus siten, että lopputulos vastaa mahdollisimman tarkasti arkkitehtuurisuunnitelmaa, täyttää sille asetetut vaatimukset ja minimoi toteutusvirheiden määrän. Prosessissa keskitytään pääasiassa määrittelemään yksittäisen komponentin kehityssykli ja yleinen toimintatapa. Lisäksi sovelluskehityksessä – kuten arkkitehtuurisuunnittelussa – pyritään lisäämään tiedon määrää organisaatiossa tallentamalla ratkaisuja ja valitsemalla hyödyllisimmät komponentit, jotka voidaan muuttaa uudelleen käytettäviksi kirjastoiksi.

Ennen itse sykliin paneutumista mainitaan muutama asia sovelluskehityksen kokonaisuudesta. Aikaisemmin vaatimusten hallinnan prosessissa viitattiin siihen todellisuuteen, että vaatimukset voivat muuttua, ja vaatimusten hallinta on siksi dynaaminen prosessi, jossa muutokset pyritään ottamaan huomioon. Tämän tapaisen vaatimusten hallinnan prosessin rinnalla sovelluskehityksessä on syytä kerätä mahdollisimman paljon palautetta ja tietoa jo toteutetuista kokonaisuuksista syötettäväksi vaatimusten hallintaan ja arkkitehtuurisuunnitteluun. Sovelluskehityksessä siis pyritään luomaan toimivia väliversioita ketterän sovelluskehityksen käytäntöjen mukaisesti.

3.3.1 Etenemisen seuranta

Tässä insinöörityössä suositellaan vaatimusten seurantaa järjestelmälle luodusta Kanban-taulusta [Albarqi & Qureshi 2018]. Tässä Kanban-taulussa yhdistyy käyttötapaukset ja muut vaatimusten varmentamiseen liittyvät tehtävät (esimerkiksi ei-toiminnallinen testaus). Kuvassa 7 esitetään tästä esimerkki ja kuvassa 9 esitetään taulun käyttötapauksen sisältö. Tässä insinöörityössä sovelluskehitystä tukevat toimenpiteet ovat yhdistelmä

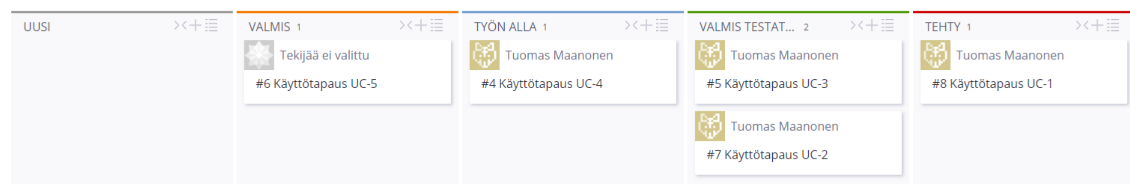
Scrum- ja Lean-malleja (esimerkiksi kehityksen aikaiset tapaamiset, työn priorisointi, kehityksen seuranta ja kommunikaatio asiakkaan kanssa) [Albarqi & Qureshi 2018].



Kuva 7. Koko järjestelmän Kanban-taulu.

Kuvassa 7 esitetyssä taulussa on viisi vaihetta. Näistä valmis tarkoittaa valmista käyttötapausta tai tehtävää, jota ei kuitenkaan ole vielä toteutettu. Työn alla tarkoittaa siirtymistä toteutukseen ja valmis testattavaksi tarkoittaa siirtymistä testaus/validointivaiheeseen.

Väliversioiden kehitykseen suositellaan Scrum-tyyppisiä muutaman viikon jaksoja (sprint). Näitä jaksoja varten järjestelmän Kanban-taulusta valitaan kyseisellä jaksolla toteutettavat vaatimukset ja muut tehtävät [Albarqi & Qureshi 2018]. Kuvassa 8 esitetään jaksoa varten suodatettu Kanban-taulu.



Kuva 8. Suodatettu taulu tällä hetkellä suorituksessa olevalle jaksolle (sprint).

Koska sovelluskehitystä on helpompaa ajatella komponenttien toteutuksena (suoraan vaatimusten toteutuksen sijaan), lisätään käyttötapauksien alle niiden toteutumiseen

vaadittavat tehtävät. Tämä esitetään kuvassa 9, jossa näkyy käyttötapauksen yksityiskohdat ja alitehtävät. Alitehtäviä voidaan luoda vapaasti, mutta niissä on mahdollista esimerkiksi puhua arkkitehtuurisuunnitelman komponenttien toteuttamisesta (kuten kuvassa on tehty). Tällä tavalla kehitystoiminnassa voidaan edetä yksiselitteisellä tavalla, joka on kuitenkin sidoksissa vaatimuksiin.

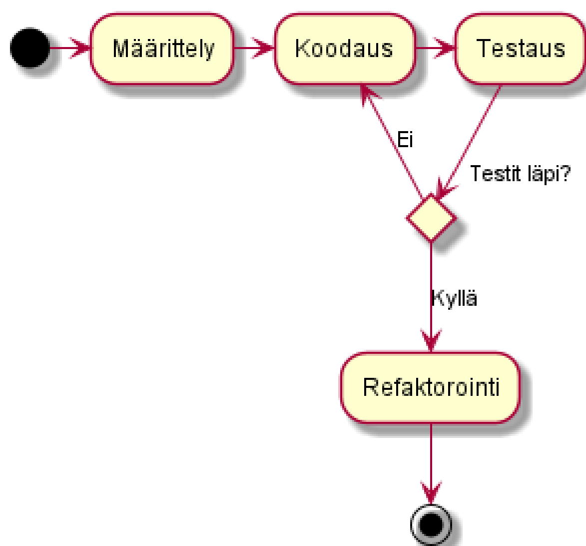
Kuva 9. Käyttötapaus ja sen suoritukseen tarvittavat tehtävät.

Sovelluskehityksen työ jaetaan arkkitehtuurisuunnitelman komponenttien mukaisesti. Koodi voidaan esimerkiksi jakaa komponentteja vastaaviin paketteihin. Tällä tavalla arkkitehtuuri toimii myös koodin karttana.

3.3.2 Sovelluskehityksen menetelmä

Komponenttien sovelluskehitys tehdään iteratiivisesti määrittelyn, koodauksen, testauksen ja refaktoroinnin sykleissä kuvan 10 esittämällä tavalla. Testeistä pyritään tekemään pieniä ja niiden rakenne määritellään jo määrittelyvaiheessa, ennen itse toteutuksen koodausta. Prosessi pohjustetaan siis test-driven development (TDD) -menetelmään, koska ne tutkimukset, joita on tarjolla, osoittavat, että käytäntö parantaa sovelluslaatua ja vähentää toteutusvirheiden määrää (verrattuna muihin toimintatapoihin) [Nagappan ym. 2008; Bhat & Nagappan 2006; Canfora ym. 2006]. Prosessissa kuitenkin yritetään minimoida turhan vaativa ja vähäisen hyödyn testaaminen, koska menetelmä hidastaa prosessia eikä sitä välttämättä ole järkevää hyödyntää kaikkeen. Esimerkiksi käyttöliittymän kehitystä TDD-menetelmällä pidetään yleisesti kömpelönä [Hellmann ym. 2010]. Tässä

insinöörityössä on yritetty löytää kultainen keskitie. TDD-menetelmä vastaa myös insinöörityön alussa mainittuja Dave Thomasin periaatteita. Seuraavaksi esitellään prosessin vaiheet yksityiskohtaisemmin.



Kuva 10. Sovelluskehityksen sykli.

Prosessin lopputuloksena ei tarvitse olla 100 %:n testikattavuus; riittää, että komponentin oleellisten vastuiden toteutuminen on validoitu ja prosessia on noudatettu. Tätä prosessia seuraamalla saadaan ideaali tilanteessa, arkkitehtuuria vastaava koodin rakenne, läpi mietityt rajapinnat, ei-toiminnalliset vaatimukset toteuttava koodi ja ryhmä testejä regressiotestaukseen.

Määrittelyvaihe

Kuten jo aiemmin mainittiin, määrittelyvaiheessa määritellään toteutettavan komponentin testi ja sitä kautta se, miten komponentin halutaan toimivan. Tätä ei tule pitää pelkästään validoinnin keinona, vaan se on myös määrittelyä ja dokumentointia. Testistä on siis ideaalitilanteessa helppo selvittää, mitä komponentin halutaan tekevän ja mahdollisesti, mihin vaatimuksiin tai arkkitehtuurisuunnitelman osaan se perustuu. Tällä automaattisesti parannetaan suunnittelua: kun testit luodaan etukäteen, tulee mietittyä, miten jokaista yksikköä käytetään. Yksi huomioitava yksikkötestausmenetelmä on data-driven

testing (DDT). Jos komponenttia pidetään hyvin monimutkaisena, voidaan määrittely tarvittaessa suorittaa myös UML-kaavioilla, matemaattisesti tai muilla formaaleilla menetelmillä. Näiden hyödyllisyys arvioidaan erikseen, eikä niiden käyttöä velvoiteta prosessissa. Määrittely voidaan tehdä myös ilman testejä esimerkiksi kommenttiin silloin, kun ympäristössä ei ole tarpeeksi sujuvaa tukea automaattiselle testaukselle. Oleellisinta on, että komponentilta haluttu toiminta on ainakin alustavasti suunniteltu ennen koodausta. Määrittelyä ei tarvitse tehdä loppuun ennen koodausta, vaan pelkästään tarpeellinen määrä koodauksen tueksi TDD-käytännön mukaisesti.

Koodaus-, testaus- ja refaktorointivaihe

Toinen vaihe on itse koodaus, jossa toteutetaan aikaisemmin määritelty komponentti. Koodauksen yksityiskohdista ei puhuta sen enempää, vaan työ jätetään yksittäisille kehittäjille. Koodauksessa noudatetaan arkkitehtuurisuunnittelun aikana määriteltyjä rajoituksia ja ohjeistusta. Tätä kautta varmistetaan alhaisella tasolla, että ei-toiminnalliset vaatimukset tulee huomioitua. Koodaus lopetetaan siinä vaiheessa, kun sen uskotaan toteuttavan tähän asti määritellyt toiminnot.

Kolmas vaihe on testaus. Tässä viitataan yksinkertaisesti jo määriteltyjen testien ajamiseen. Jos automaattisia testejä ei ole, tehdään testaus manuaalisesti. Siinä vaiheessa, kun testit menevät läpi, koodi refaktoroidaan ja prosessissa siirrytään seuraavaan iteraatioon. Mikäli testit eivät mene läpi, siirrytään takaisin koodausaskeleeseen.

Syklin viimeinen vaihe on refaktorointi. Refaktorointi vaihe edellyttää, että komponentin kaikki testit menevät läpi. Refaktoroinnissa arvioidaan, täyttyykö Dave Thomasin mainitsema päätöksenteon kriteeri, eli onko toteutuksessa valitut rakenteet tulevaisuudessa mahdollisimman helppo muuttaa. Tämä saavutetaan esimerkiksi poistamalla haisevia rakenteita (code smells). Refaktoroinnissa voidaan hyödyntää staattisen analyysin työkaluja sekä tietysti kehittäjän omaa harkintaa.

Lisäksi kun komponentti valmistuu, tulee se viedä vertaisarvioon (code review). Vertaisarvioi ei sinänsä kuulu refaktorointi askeleeseen, mutta siinä voidaan vielä suorittaa refaktorointia, ennen komponentin integraatiota muuhun järjestelmään.

3.4 Testaus

Vaikka arkkitehtuurisuunnittelu ja sovelluskehitys olisi tehty hyvin kurinalaisesti, ei lopputuloksen vaatimuksenmukaisuuden varmistamisessa kannata luottaa silkkaan tuuriin. Sovelluksen vaatimuksenmukaisuutta on tärkeää pyrkiä validoimaan. Näitä validointitoimenpiteitä kutsutaan testaukseksi. Myös toimenpiteet, joilla arvioidaan, vastaavatko kehitetyt vaatimukset sidosryhmien todellisia tarpeita, ovat testausta. Testausta tehdään koko ajan vaatimusten kehittämisen, arkkitehtuurisuunnittelun ja sovelluskehityksen rinnalla. Tässä prosessissa käsitellään vain toteutuksen validointia, ei sitä edeltävää konseptitason validointia. Tässä insinöörityössä kuvataan lyhyesti neljä eri sovellustestauksen muotoa. Ne ovat yksikkötestaus, integraatiotestaus, regressiotestaus ja hyväksymistestaus. Testausta on luontevaa pitää varmentamisen paitsi myös dokumentaation välineenä. Tämä on yksi syy suosia automaattista testaamista, jossa testit korvaavat suuren osan järjestelmädokumentaatiosta.

Testausta tehdään usein liian vähän ja liian myöhään. Iso osa virheistä tapahtuu yksikkötasolla ja aikaisin kehitysprosessissa; mitä aikaisemmin nämä virheet huomataan, sitä halvemmalla ne voidaan korjata. Tämä on nykyään yleistä tietoa ohjelmistotestauksessa ja myös tutkimukset tukevat näkemystä. [Woodward & Hennell 2005.]

3.4.1 Yksikkö- ja integraatiotestaus

Yksikkötestauksessa keskitytään yksikön rajapinnan validointiin ja yksikkötestit on siksi järkevää määritellä rajapinnan suunnittelun jälkeen, mutta ennen toteutusta. Yksikkötestien tulisi olla pieniä ja keskittyä tarkasti tiettyyn asiaan [Hauer 2019]. Testejä on luontevaa tehdä normaaleilla sekä poikkeavilla arvoilla. Myös yleiset ongelmakohdat kuten raja-arvot kannattaa testata. Yksikkötestit ovat ensisijaisesti sovelluskehityksen työkalu; niiden avulla toteutuksen lisäksi myös rajapintojen selkeys ja testattavuus tulee mietittyä. Lisäksi kun testi määritellään ennen toteutusta, on kehityksen lähtökohtana aina vaatimukset. Yksikkötestauksen yksityiskohdat riippuvat käytetystä ympäristöstä. Usein yksikkötestauksessa suljetaan pois ulkoiset tekijät, jotta ainoastaan yksikkö itse voi aiheuttaa ongelmia. Olio-ohjelmoinnissa tämä tehdään käyttämällä ympäristön jäljittely (mock) -oliokirjastoa.

Integraatiotestit eroavat yksikkötesteistä vain käsitteellisesti. Integraatiotesteissä tarkoituksena ei ole erottaa yksikköjä tai komponentteja toisistaan, vaan testata kokonaisuuksia. Integraatiotestit suoritetaan usein samoilla työkaluilla kuin yksikkötestitkin. Ne kohdistuvat pääasiassa komponenttien rajapintoihin ja niiden avulla voidaan testata komponentille asetetut vastuut. Myös komponenttien ei-toiminnalliset vaatimukset olisi hyvä testata, mutta tämä kaatuu usein testauksen vaikeuteen ja siihen, että helppoja työkaluja ei ole olemassa. Ei-toiminnalliset vaatimukset pidetään kuitenkin mielessä testauksen tuloksia arvioitaessa.

Oleellisimmat automaattisen testauksen työkalut ovat testinajaja (test-runner) ja testilauseet (asserts). Testilauseet ovat loogisia kyllä- tai ei-väittämiä, jotka testisuorituksen tulee läpäistä. Java-ympäristössä yksikkö- ja integraatiotestit ajetaan JUnit 5 -työkalulla. Itse testilauseisiin voidaan myös hyödyntää Junit-työkalua, mutta kattavampi ratkaisu on JAssert-työkalu; JAssert-työkalun käyttö lyhentää testikoodia. [Hauer 2019.]

Yksikkötestaus ja integraatiotestaus ovat niin sanottua funktionaalista testausta eli niissä testataan funktioiden/metodien tai luokkien toimintaa. Tämän lisäksi voidaan tehdä rakenteellista testausta. Rakenteellinen testaus tarkoittaa koodin rakenteen analysointia yleensä automaattisesti staattisen analyysin työkaluilla [Woodward & Hennell 2005]. Esimerkiksi testikattavuuden laskeminen on rakenteellista analyysiä; sillä voidaan löytää reikiä testauksessa tai turhaa koodia. Staattisen analyysin työkaluja on useita, eikä niihin tässä insinöörityössä paneuduta sen enempää.

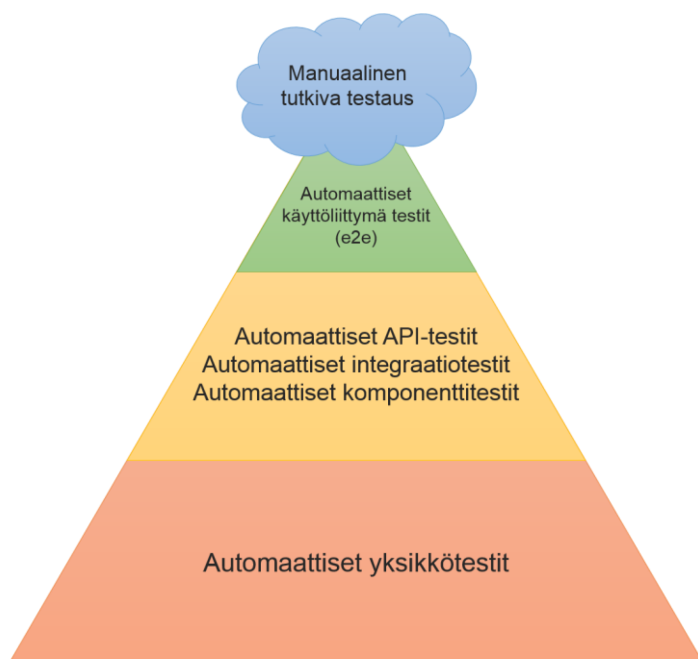
3.4.2 Regressiotestaus

Regressiotestaus viittaa yksinkertaisesti olemassa olevien testien säännölliseen ajamiseen. Yksikkö- ja integraatiotestien ajamisen lisäksi regressiotestauksessa voidaan ajaa myös staattisen analyysin työkaluja. Regressiotestauksessa pyritään löytämään muutosten aiheuttamat odottamattomat ongelmat ja sivuvaikutukset. Tällaisia muutosten aiheuttamia sivuvaikutuksia kutsutaan regressioksi; regression todennäköisyys nousee jatkuvasti kehityksen edetessä. Pitämällä muutosten määrä testauskertojen välillä vähäisenä, on mahdollisia uusia ongelmia helpompi löytää. Jatkuvassa integraatiossa regressiotestit ajetaan säännöllisesti automaatiopalvelimella. Tällä tavalla testaus tulee myös automaattisesti dokumentoitua, ja kaikki projektin jäsenet voivat tarkistaa järjestelmän

koodin ajankohtaisen tilan. Lisäksi kehittäjän ei tarvitse odotella, että kaikki testit ajetaan hänen omalla työasemallaan. Automaatiopalvelimista lisää konfiguraationhallinnassa. Jos regressiotestaus ei mene läpi, tulee asia korjata ennen uusien ominaisuuksien vientiä versionhallintaan. Tämän tapaisten käytäntöjen arvo nousee varsinkin silloin, kun kehittäjät eivät työskentele samassa fyysisessä tilassa ja koodin määrä on suuri.

3.4.3 Hyväksymistestaus

Hyväksymistestaus on koko järjestelmän testausta; siinä ei keskitytä mihinkään yksittäiseen palikkaan vaan koko järjestelmän vaatimuksiin. Hyväksymistestaus on usein testaajien, omistajan tai muiden sidosryhmien manuaalista tutkivaa testausta. Hyväksymistestausta ei välttämättä tehdä projektin alussa, mutta heti kun saadaan valmiiksi jotain käytettävää, on sitä luontevaa alkaa hyväksymistestaamaan säännöllisesti. Esimerkiksi jokaisen sovelluskehitysjakson väliversio tullaan hyväksymistestaamaan. Jatkuvaa hyväksymistestausta helpottaa sovelluksen uusimman version automaattinen julkaisu testiympäristöön. Hyväksymistestaukseen lasketaan myös lopullinen ei-toiminnallisten vaatimusten testaus, esimerkiksi tietoturva- ja käytettävyystestaus. Hyväksymistestauksen monet muodot ovat liian laaja aihe tässä insinöörityössä käsiteltäväksi. Myös hyväksymistestausta voidaan automatisoida, mutta työkalut ovat usein kömpelöitä ja monen tyyppiselle hyväksymistestaukselle automatisointi on usein mahdotonta. Hyväksymistestejä ei myöskään suoriteta yhtä usein kuin yksikkö- ja integraatiotestejä. Näistä syistä hyväksymistestien automatisoinnissa todennäköisesti on isommat riskit ja pienemmät hyödyt. Tämä kaikki tietysti muuttuu, jos automatisoinnin saavuttamiseen löydetään hyvä ja nopea menetelmä.



Kuva 11. Ideaali testimäärän jakauma pyramidissa, jossa helposti automatisoitavat alimpana [Scott 2018].

Lopuksi on tärkeää mainita, milloin eri testausmenetelmiä on järkevää käyttää ja miten paljon. Hyvää testauksen määrän jakaumaa kuvataan usein kuvan 11 tapaisella pyramidilla. Pyramidin rakennetta – jossa yksikkötestit ovat alimpana – perustellaan sillä, että ”korkeamman tason” testit johtavat saman koodin moninkertaiseen testaukseen, joka vaikeuttaa koodin muuttamista [When to Use Test Driven Development 2016]. Kuitenkin osa alan ammattilaisista suosii integraatiotestausta, vaikka se suoritetaan korkeammalla tasolla kuin yksikkötestaus. Tätä perustellaan sillä, että varsinkin jäljittelyyn (mock) perustuva yksikkötestaus vaikeuttaa refaktorointia, on työlästä eikä oikeastaan testaa sitä, miten järjestelmää todellisuudessa käytetään [Hauer 2019]. Tässäkin asiassa loppujen lopuksi kehittäjät ja testaajat päättävät, miten on järkevää toimia. Tarpeetonta päällekkäistä testausta kannattaa kuitenkin välttää. Sovelluskehityksen aikana syntyy luonnollisesti joitain testejä. Testauksen tarvetta niiden lisäksi voidaan arvioida esimerkiksi testikattavuuden työkaluilla. Työn määrä on syytä priorisoida; ei ole järkevää käyttää kymmenkertaista kehitysaikaa oikeaoppisen yksikkötestin luomiseen alhaisen prioriteetin yksikölle. Varsinkin yksinkertaisille komponenteille muutama integraatiotesti voi riittää. Toisaalta kriittiset yksiköt ja komponentit on syytä testata tarkkaan. Yleisesti sovelluskehityksen produktiivisuutta pidetään hyvin vaikeana tai jopa mahdottomana arvioida, mikä

vaikeuttaa yksinkertaisten ohjeiden antamista [When to Use Test Driven Development 2016].

4 Prosessin tukivaiheet

4.1 Versionhallinta

Ennemmin tai myöhemmin sovelluskehityksessä syntyy tilanne, jossa tärkeitä tiedostoja ja dokumentteja on lukuisia ja useita näistä muutetaan samanaikaisesti. Tiedon kehittämisen aikana saattaa tapahtua vahinkoja, jotka johtavat joko väärään muutokseen tai koko tiedoston häviämiseen. Versionhallinnassa pyritään varmistamaan, että mitään tietoa ei koskaan häviä. Sillä mahdollistetaan, että ongelmatilanteissa tieto voidaan aina palauttaa haluttuun versioon. Lisäksi sovelluskehityksen versionhallinnassa pyritään tukemaan rinnakkaiskehitystä ja esimerkiksi mahdollistamaan vanhojen julkaisujen säilytys.

Versionhallinnan prosessi pätee kaikkiin tiedostoihin, ei pelkästään lähdekoodiin. Lähdekoodi ja muut tiedostot voidaan laittaa samaan versionhallinnan työkaluun, tai niille voidaan käyttää eri työkaluja. Tässä insinöörityössä suositaan periaatetta, jossa lähdekoodi ja siihen liittyvä dokumentaatio laitetaan samaan versionhallintaan ja muut tiedostot eri versionhallintaan.

Projektin alussa luodaan projektia varten versionhallintaan kansiot. Kansiorakenteiden monimutkaisuudessa huomioidaan projektin suuruus; muutamalle tiedostolle on turha luoda alikansioita. Yleensä projektissa on ainakin vaatimuksiin, suunnitteluun ja lähdekoodiin liittyviä tiedostoja; ne voidaan jakaa omiin kansioihinsa.

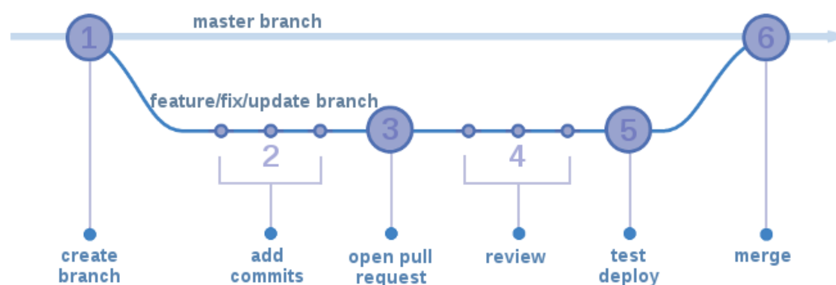
Versiohallinta on hyvin työkalupohjaista toimintaa. Se, mitä voi tehdä ja mitä ei voi tehdä, riippuu työkalusta. Tässä insinöörityössä lähdekoodin hallintaan käytetään Git-versionhallintaa ja muihin dokumentteihin yrityksen omaa dokumentinhallintajärjestelmää. Tässä insinöörityössä keskitytään lähdekoodin versionhallintaan.

4.1.1 Versionhallinnan mallit

Lähdekoodin versionhallinta jaetaan käsitteellisesti haaroihin ja julkaisuihin. Haaroitukseen on olemassa eri malleja; näistä yleisesti käytettynä on esimerkiksi git-flow-niminen

malli [Driessen 2010]. Git-flow-mallin luoja Vincent Driessen ei kuitenkaan pidä sitä soveltuvaksi esimerkiksi jatkuvasti kehitettäville verkkosovelluksille. Tähän hän suosittelee kirjoituksensa päivityksessä Github-flow-mallia [Driessen 2010]. Versionhallinnan prosessissa otetaan käyttöön joko git-flow tai github-flow-malli, järjestelmän tyypin mukaan. Seuraavaksi kuvaillaan mallien oleelliset piirteet.

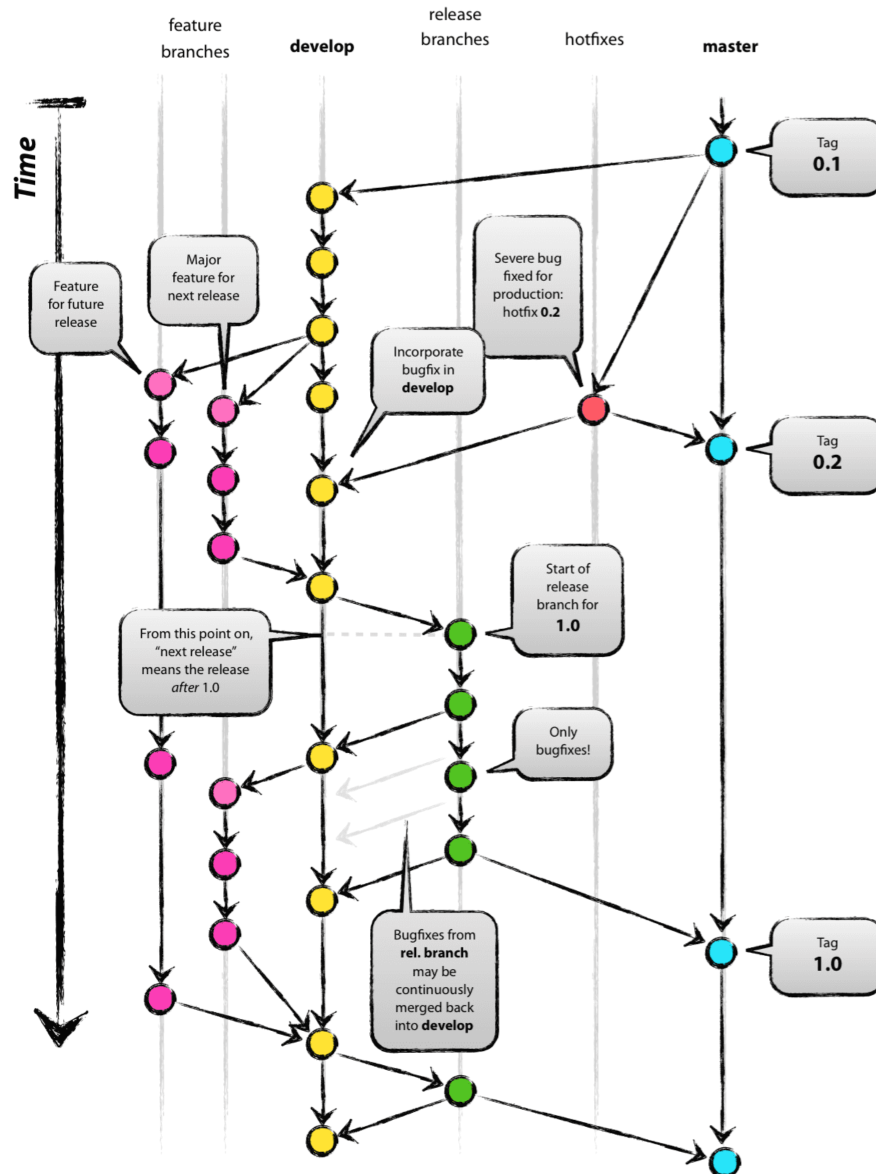
Git-flow ja Github-flow eroavat siinä piirteessä, että git-flow-mallissa on erillisiä julkaisuja, kun taas Github-flow-mallissa master-haara on ainut jatkuvasti päivitettävä julkaisu [Driessen 2010; Understanding the GitHub flow 2017]. Tämä piirre tekee Github-flow-mallista yksinkertaisemman ja mahdollistaa saman yksinkertaisen prosessin käytön kaikille muutoksille [Git Workflow 2018].



Kuva 12. Github-flow-malli [Git Workflow 2018].

Kuvassa 12 esitetty Github-flow-mallissa on vain kahdentapaisia haaroja: master-haara ja muutoshaara (ominaisuus, korjaus, päivitys). Kaikki muutokset tehdään aina muutoshaaraan, josta ne viedään vertaisarvioin ja testauksen kautta takaisin master-haaraan. Tällä tavalla master-haara pidetään aina hyväksyttynä, julkaisuvalmiina versiona. Tästä myös huomataan, että Github-flow-malli on suunniteltu toimimaan yhdessä vertaisarvioinnin työkalun kanssa (jollainen löytyy esimerkiksi juuri GitHub-palvelusta).

Metropolia



Kuva 13. Git-flow-malli [Driessen 2010].

Kuvassa 13 esitetyssä git-flow-mallissa on kaksi päähaaraa ja useita sivuhaaroja. Ideana on se, että kaikki uusi kehitystyö tehdään develop (kehitys) -haaraan tai sen vasemmalla esitettyihin feature (toiminto) -haaroihin. Kun julkaisun valmistelu aloitetaan, luodaan julkaisulle oma release (julkaisu) -haara. Tähän julkaisuhaaraan lisätään vain testauksessa ilmenneiden virheiden korjauksia. Lopuksi julkaisu viedään master-haaraan, jossa se merkataan (tag) versionumerolla. Julkaisuhaarasta kehityshaaraan voidaan viedä vain tehdyt virheenkorjaukset. Kuten kuvasta 13 huomataan, git-flow-malli on huomattavasti monimutkaisempi kuin Github-flow-malli. Siinä julkaisun konsepti on

keskiössä [Git Workflow 2018]. Kuitenkin git-flow-mallilla saavutetaan tuki useammalla erilliselle julkaisulle, koska jokaiselle niistä luodaan oma haara, jota voidaan tarvittaessa päivittää, vaikka kehityshaarassa olisikin jo edetty seuraavan version luontiin.

4.1.2 Versionhallinnan toimenpiteet

Keskustellaan yleisten mallien lisäksi hieman eri toimenpiteistä. Versionhallinnan toimenpiteistä tärkeimpinä voidaan pitää muutoksien hakemista, muutosten viemistä ja muutosten yhdistämistä.

Muutosten hakeminen ja vieminen

Muutoksien hakeminen on toiminnoista yksinkertaisin; siinä päivitetään omaan kehitysympäristön kopioon muiden luomat muutokset. Muutoksien hakemista kannattaa tehdä mahdollisimman usein, esimerkiksi kerran työpäivässä; tällä tavalla iso osa konflikteista vältetään ja konfliktit eivät kerkeä ajautua kovin suuriksi.

Aina kun sovelluskehityksessä valmistuu uusi sovellusyksikkö, tulee se viedä johonkin versionhallinnan haaraan. Viennin (commit) kommentista tulee selvitä muutoksen tarkoitus; tämä lähinnä siksi, että niitä voidaan jälkeenpäin käyttää muutosluetteloa varten. Kommentista selviää siis esimerkiksi lisätyt ominaisuudet ja korjatut ongelmat. Tuotannon alkuvaiheissa kommentteja voi olla vaikea keksiä eikä niitä silloin välttämättä myöskään tarvitse; tärkeimmässä asemassa ne ovat jo tuotannossa olevan tuotteen päivityksen yhteydessä. Lähtökohtaisesti ohjelmiston käännettyä versiota tai käytettyjä kirjastoja ei viedä versionhallintaan. Kehittäjien tulee myös pyrkiä siivoamaan tarpeettomat tiedostot ennen vientiä. Lähtökohtaisesti keskussäiliöön viedään valmiita, testattuja, puhdistettuja ja kommentoituja komponentteja. Git-työkalua käytettäessä paikalliseen säiliöön voidaan viedä myös keskeneräisiä komponentteja.

Muutosten yhdistäminen

Muutosten yhdistäminen on toimenpiteistä monimutkaisin. Usein yhdistäminen onnistuu automaattisesti, mutta joskus siinä syntyy ongelmia. Git-työkalulla on useita tapoja käsi-

tellä muutoksien yhdistämistä. Muutoksien yhdistäminen pätee paikallissäiliöstä keskussäiliöön (remote) vientiin ja kahden eri haaran yhdistämiseen. Seuraavissa kappaleissa asiaa käsitellään haarojen yhdistämisen näkökulmasta, mutta idea on sama.

Yksinkertaisimmat tilanteet ovat ne, joissa yhdistettävät haarat joko ovat identtisiä (eli niissä on vain ja ainoastaan samoja muutoksia) tai toinen haaroista sisältää jo ennalta toisen haaran kaikki muutokset, mutta siinä on myös uusia muutoksia. Identtisten haarojen tilanteessa Git yksinkertaisesti ilmoittaa, että haara johon muutoksia yritetään yhdistää, on jo ajan tasalla. Jälkimmäisessä tilanteessa (haara sisältää jo kaikki muutokset ja enemmänkin) tuodaan yhdistettävään haaraan siitä puuttuvat muutokset. Kummassakaan tilanteessa ei oikeasti luoda uutta revisiota; tällä tavalla Git tekee operaatioista täysin turvalliset ja suojelee käyttäjää virheiltä. Lisäksi Git-työkalulla on useita eri tapoja yhdistää haaroja, jotka eroavat toisistaan. Käytännössä näissä kaikissa tavoissa Git ensin etsii haarojen yhteisen juuren, eli revision, jossa ne ensimmäisen kerran eriytyivät toisistaan. Näistä yksinkertaisimmassa tilanteessa Git ottaa toisesta haarasta juuren jälkeen tehdyt muutokset ja lisää ne toisen haaran muutosten päälle ja luo tästä uuden revision. Git hallitsee myös monimutkaisempia tilanteita, mutta niitä ei tässä insinöörityössä käsitellä. Mikäli mikään näistä yhdistystavoista ei onnistu, koska kehittäjät ovat muuttaneet samanaikaisesti samoja asioita, tulee yhdistys toteuttaa manuaalisesti kertomalla Git-työkalulle, mitkä muutokset halutaan säilyttää. Tässä auttaa esimerkiksi git diff -komento, joka esittää eri haarojen eroavaisuudet. [Loeliger & McCullough 2012.]

Julkaisuversioiden hallinta

Jossain vaiheessa sovelluskehitystä lähestytään julkaisua. Git-flow-mallia käytettäessä julkaisua varten luodaan oma haara [Driessen 2010]. Tähän haaraan voidaan viedä testauksessa ilmeneviä korjauksia, konfiguraatiota tai muita julkaisuun tarvittavia muutoksia. Uusia toiminnallisuuksia siihen ei lähtökohtaisesti lisätä, vaan julkaisuhaara tulee luoda vain silloin, kun kaikki julkaisuun tulevat toiminnallisuudet on jo lisätty. Ideana julkaisuhaarassa on lähtökohtaisesti korjata ongelmia ja parantaa julkaisun laatua ilman koodin lisäämistä. Se on siis julkaisun viimeistelyyn tarkoitettu haara. Kun lopulta ollaan valmiina julkaisuun, git-flow-mallissa julkaisuhaara viedään master-haaraan ja versio merkataan julkaisun tunnuksella (esimerkiksi 0.1 tai 1.0) [Driessen 2010]. Tällä tavalla

jokainen ”maailmalla” oleva julkaisu pystytään tarvittaessa löytämään. Github-flow-mallissa, versionhallinnan näkökulmasta, muutosten vieminen master-haaraan on aina julkaisu; todellisuudessa master-haara on vain julkaistava versio ja lopullisesta julkaisusta päätetään erikseen. Github-flow-mallissa ei ollenkaan käytetä julkaisukäsitettä.

Julkaisujen versiointi noudattaa yksinkertaista käytäntöä, jossa julkaisun tunnus muodostuu maksimissaan kolmesta numerosta, jotka ovat version numero, revision numero ja rakennuksen (build) numero. Version numero alkaa nollasta ja sitä muutetaan vain, kun järjestelmään on tehty merkittäviä muutoksia (esimerkiksi git-flow mallissa ensimmäinen todellinen julkaisu on versio 1). Revision numero tulee käyttöön ensimmäisen todellisen julkaisun jälkeen ja sitä korotetaan, kun järjestelmään tehdään pieniä muutoksia, esimerkiksi kun korjataan virheitä tai tietoturvahaavoittuvuuksia. Rakennuksen numero kasvaa joka kerta, kun järjestelmä rakennetaan automaatiopalvelimella. Lopputuloksena on versiointitunnus, jossa esimerkiksi ensimmäisen version, kolmas revisio, rakennus 555 merkataan 1.3 b555.

4.2 Konfiguraationhallinta

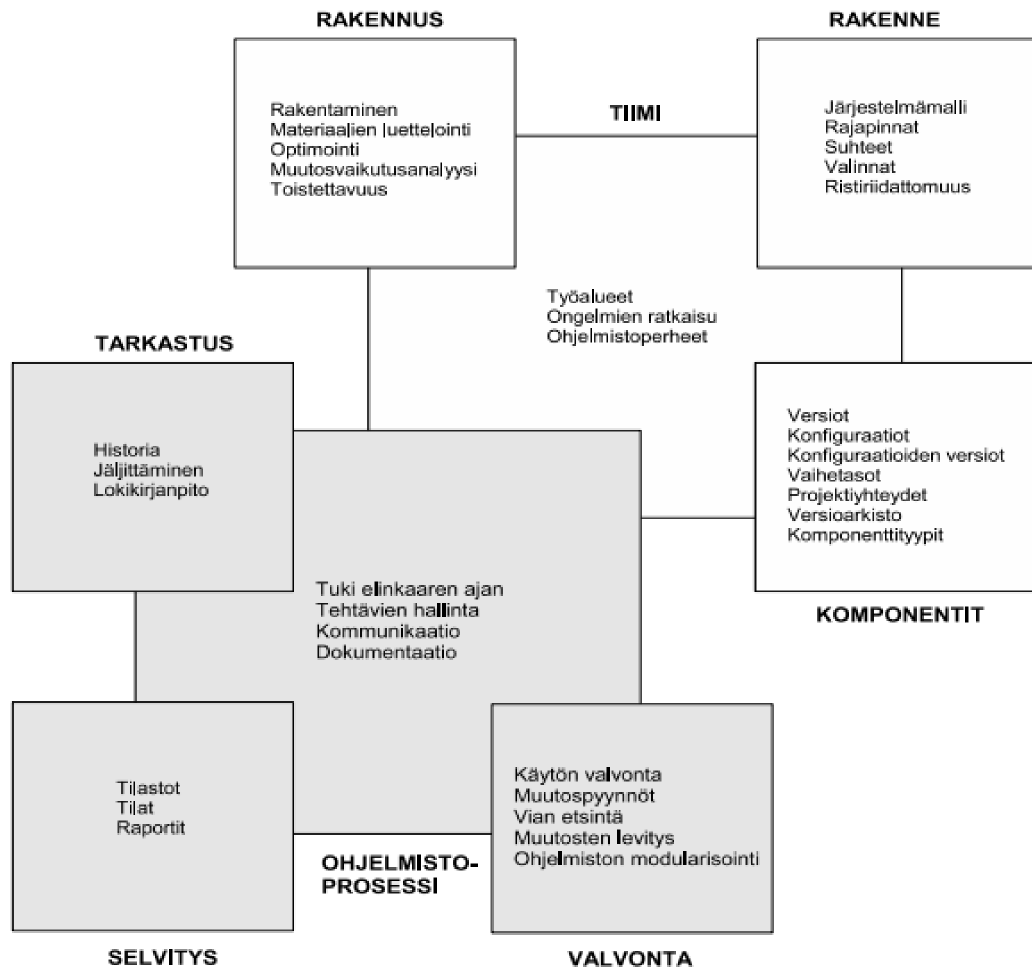
Kuten insinöörityön versiohallinnan luvussa mainittiin, ohjelmistokehityksen edetessä hallittavan tiedon määrä kasvaa usein hyvin suureksi. Tämä tieto muuttuu ja kehittyy jatkuvasti, mikä aiheuttaa usein monenlaista sekaannusta. Tämän sekaannuksen minimointiin ja hallintaan pyritään konfiguraationhallinnalla [Jouni Niemelä 2006: 1].

4.2.1 Konfiguraationhallinnan määritelmä

Konfiguraationhallinta viittaa yleisesti ohjelmiston, sen elinkaaren ja muutosten kurinalaiseen hallintaan [Jouni Niemelä 2006: 1]. Konfiguraationhallintaan sisältyy myös monilta osin esimerkiksi versionhallinta. Tässä prosessissa kuvataan muiden vaiheiden ulkopuolelle osuvia konfiguraationhallinnan osia.

Usein konfiguraationhallinnan osiksi mielletään aineiston tunnistaminen, muutoksen hallinta, tilanseuranta ja konfiguraation tarkastus. Lisäksi konfiguraationhallintaa voidaan

tutkia eri näkökulmista, joita kutsutaan toiminnallisiksi alueiksi. Toiminnalliset alueet esitetään kuvassa 14. Kuvasta näkyy, mitä eri toiminnallisia alueita on Dartin mukaan olemassa ja mitä niihin kuuluu. Jokaisessa toiminnallisessa alueessa on asioita, jotka sisällytetään osaksi konfiguraationhallintaa. [Dart 1991.]



Kuva 14. Dartin toiminnalliset alueet ja niiden suhteet [Dart 1991; Jouni Niemelä 2006: 2].

Lisäksi konfiguraationhallintaan kuuluu variaatiokäsite. Variaatiot ovat sovelluksen aikaisempia versiota tai tietylle asiakkaalle/asiakasryhmälle muokattuja versiota, jotka eroavat pääversiosta (baseline). Konfiguraationhallinnassa hallitaan myös variaatiota ja niiden luontia.

4.2.2 Konfiguraationhallinnan toteutus

Keskustellaan seuraavaksi, miten eri konfiguraationhallinnan osa-alueet pystytään käytännössä toteuttamaan.

Komponenttien, rakennuksen ja rakenteen alueet

Jos toiminnallisia alueita ja muita ajatuksia konfiguraationhallinnasta yritetään yhdistää, voidaan ajatella, että aineiston tunnistaminen onnistuu osana komponenttien ja rakennuksen toiminnallisia alueita. Näihin kuuluvat esimerkiksi versiot, konfiguraatiot ja rakentaminen.

Versionhallinnassa saavutamme jo komponenttien tallennuksen ja versioinnin. Kykenemme versionhallinnasta tunnistamaan konfiguraatioon kuuluvan tarkan lähdekoodin esimerkiksi leimaamalla kyseisen konfiguraatioon liitetyn revision. Tällöin, jos konfiguraatiolla tarkoitetaan version 1.0 julkaisun konfiguraatiota, löytyy siihen liittyvä lähdekoodi leimalla 1.0. Pystymme siis tunnistamaan lähdekoodiin liittyvän aineiston.

Rakentamalla sovelluspakettimme automaattisesti jollain rakennusohjelmalla kykenemme yhdistämään koontiohjeet konfiguraatioon ja sitä kautta tunnistamaan kaikki käytetyt riippuvuudet ja muut elementit, joita sovelluspaketin luomiseen tarvitaan. Manuaaliset vaiheet rakennusprosessissa tulee minimoida, jotta rakentaminen olisi toistettavissa ja koontiohjeista pystyttäisiin selvittämään kaikki tarvittavat elementit. Rakennuksessa ei siis tulisi käyttää palikoita, joita ei ole mainittu koontiohjeissa tai vähintään jossain muussa dokumentaatiossa. Jos näin toimitaan, koontiohjeissa toteutuu yleensä loput aineiston tunnistamisesta.

Huomioidaan vielä tilanne, jossa yksittäinen versionhallinnan säiliö ja sen koontiohje eivät sisällä kaikkia konfiguraatioon tarvittavia komponentteja. Tällainen tilanne esiintyy esimerkiksi silloin, kun useampi erillinen järjestelmä muodostavat isomman järjestelmän tai järjestelmien yhdistelmän. Tällaisessa tapauksessa myös arkkitehtuurisuunnitelma voi sisältää konfiguraatiotietoa ja on syytä pitää ajan tasalla, jotta siitä voidaan tarkastaa konfiguraatioon kuuluvat erilliset järjestelmät ja komponentit. Tällaiset suuret konfiguraatiot ovat usein tilanteita, joissa todelliset konfiguraationhallinnan työkalut (kuten Ansible),

tulevat hyödyllisiksi. Näillä työkaluilla voidaan automatisoida useamman eri järjestelmän ja niiden infrastruktuurin luonti. Tällöin myös monimutkaisen konfiguraation kaikki osiot pystytään tunnistamaan. Tässä tilanteessa mukaan tulee oleellisesti myös rakenteen toiminnallinen alue.

Tarkastuksen, selvityksen ja valvonnan alueet

Muutoksen hallinta, tilanseuranta ja konfiguraation tarkistus liittyvät oleellisesti tarkastuksen, selvityksen ja valvonnan toiminnallisiin alueisiin.

Muutoksen hallintaa käsiteltiin jo vaatimusten hallinnan osiossa. Yleisesti ottaen muutos tapahtuu aina ensin vaatimusten tasolla, pois lukien tilanteet, joissa muutoksella korjataan toteutusvirhe. Muutos voi siis tapahtua joko muutoksen hallinnan prosessin kautta muuttamalla itse vaatimuksia tai vikailmoituksen kautta. Konfiguraationhallinnan näkökulmasta on oleellista, että vaatimuksia muutettaessa tehdään muutosvaikutusanalyysi, eli etsitään kaikki konfiguraation osiot, joihin muutos vaikuttaa.

Tilanseurantaan ja konfiguraation tarkistukseen kuuluu molempiin osioita valvonnan, tarkastuksen ja selvityksen toiminnallisista alueista. Tilanseuranta koostuu vaatimusten tilan seurannasta, vikojen seurannasta, testauksen raportoinnista ja rakennuksen historiasta. Konfiguraation tarkistus voidaan suorittaa tilanseurannan tarjoaman tiedon avulla. Nämä asiat hoituvat yleensä jollain työkalulla; vaatimusten hallinnan työkalut sisältävät yleensä tilanseurannan, versionhallinnan työkaluissa on nykyisin usein vikojen seurannan mahdollistavat työkalut, testauksen raportointi ja rakennuksen historia löytyvät liitännäisinä eri sovellustenpakettien rakennustyökaluihin.

Konfiguraationhallinnan työkalut

Tässä vaiheessa on hyödyllistä siirtyä tässä insinöörityössä käytettyihin konfiguraationhallinnan työkaluihin. Konfiguraationhallinnassa on hyödyllistä käyttää automaatiopalvelinta eli CI/CD-palvelinta (esimerkiksi Jenkins), sovelluspakettien rakentamiseen ja testaamiseen. Lisäksi tarvitaan työkalut vaatimusten ja vikojen hallintaan.

Automaatiopalvelimen kurinalaisella käytöllä mahdollistetaan monta asiaa. Ensiksi, automaatiopalvelin mahdollistaa koko rakennusprosessin ja sen historian dokumentoinnin. Jokaisen automaatiopalvelimella suoritetusta koonnista säilytetään lokitiedot. Näihin lokitietoihin voidaan sisällyttää testauksen, testikattavuuden ja staattisen analyysin raportointi. Lisäksi automaatiopalvelimelle voidaan tallentaa useita erillisiä koontiketjuja, jolla mahdollistetaan kokonaisvaltainen variaatioiden hallinta. Koska jokaisen variaation koonneista tallennetaan historia, voidaan automaatiopalvelimelta seurata niiden tilaa ja sen kehittymistä. Automaatiopalvelin myös automaattisesti numeroi koonnit ja sitä kautta niistä syntyneet paketit; tätä kautta voidaan identifioida paketti ja esimerkiksi viitata siihen testiraporteissa. Kaikki tämä helpottaa konfiguraation tarkistusta uuden julkaisun yhteydessä. Lisäksi automaatiopalvelimella voidaan nopeuttaa testauksen sykliä automatisoimalla paketin vienti testiympäristöön.

Vian ja vaatimusten tilan seurantaan automaatiopalvelin ei kykene. Kuten aikaisemmin mainittiin, tähän tarvitaan erillinen työkalu. Työkaluja vaatimusten tilan seurantaan on jo aikaisemmin mainittu. Esimerkiksi Kanban-taulu tukee juuri tätä tarvetta. Vikojen seurantaan löytyy myös työkaluja ja kuten aikaisemmin mainittiin, vikojen seuranta löytyy myös monista versionhallinnan työkaluista. Vikojen seurannan sisältäviä versionhallinnan työkaluja on esimerkiksi GitHub, GitLab ja Gitea. Vian seurannan työkaluun voidaan ilmoittaa viasta ja asettaa vialle tila esimerkiksi auki tai kiinni. Tässä auki tarkoittaa käsittelemätöntä vikaa ja kiinni korjattua tai muuten käsiteltyä vikaa.

4.3 Julkaisun hallinta

Kehitetty sovellus todetaan lopulta riittävän valmiiksi toimitettavaksi asiakkaiden käyttöön. Vaikka sovellus olisi kehitetty hyvin, voidaan toimituksessa epäonnistua. Tämä epäonnistuminen näkyy välittömästi asiakkaalle ja sillä voi olla suuretkin seuraamukset. Tästä syystä toimitustilanne täytyy hoitaa kurinalaisesti. Toimitusta/julkaisua tukevat käytännöt kuuluvat tässä insinöörityössä julkaisun hallintaan.

Vaikka julkaisun hallinnalla käsitellään pääasiassa julkaisua/toimitusta, ei julkaisun hallinta ole pelkästään ohjelmistokehityksen lopussa suoritettava toimenpide. Julkaisua valmistellaan koko ohjelmistokehityksen ajan. Tässä insinöörityössä keskitytään julkaisun

hallinnan kahteen toimenpiteeseen: dokumentaation luontiin ja pakkauksen hallittuun vientiin tuotantoympäristöön [OpenUP 2012].

Dokumentaatiolla ei julkaisun hallinnassa viitata tekniseen dokumentaatioon vaan käyttäjää avustavaan tuote- ja tukidokumentaatioon. Tuote- ja tukidokumentaatioon kuuluu esimerkiksi toimintojen dokumentaatio, käyttöohjeet, usein kysytyt kysymykset ja muu vastaava. Lisäksi tukidokumentaatioon kuuluu tukihenkilöstöä avustava dokumentaatio; tämä voi ohjeistaa esimerkiksi järjestelmän hallintaan ja yleisten ongelmien korjaamiseen. [OpenUP 2012.]

Julkaisun vientiin tuotantoympäristöön kuuluu käyttöönottosuunnitelma, käyttöönoton peruutussuunnitelma ja julkaisusta kommunikointi käyttäjille/asiakkaille [OpenUP 2012]. Julkaisun vienti edellyttää, että järjestelmän ympäristö ja sille tarpeellinen infrastruktuuri on jo määritelty. Tämä voidaan suorittaa jo sovelluskehityksen vaiheessa luomalla järjestelmästä esimerkiksi Docker-säiliö ja määrittämällä kaikkien tarvittavien järjestelmien infrastruktuuri esimerkiksi Ansible-konfiguraationhallintatyökalulla. Jos käyttöönottosuunnitelmassa hyödynnetään automatisointityökaluja, on esimerkiksi käyttöönoton peruutus huomattavasti helpompaa.

5 Yhteenveto

Tämän insinöörityön tavoitteena oli käsitellä alan kirjallisuutta ja kerätä yhteen tarvittava teoria yrityksen nykyisten menetelmien kehittämiseen ja formaalin prosessin luomiseen. Tässä insinöörityössä kuvailtu teoria ja siinä esitellyt menetelmät valikoituivat työhön tekijän oman harkinnan mukaisesti. Kokonaisuudessa tässä insinöörityössä onnistuttiin määrittelemään eri vaiheiden tavoitteet ja kuvaamaan yksi mahdollinen tapa, miten ne voitaisiin saavuttaa. Esiteltyjen menetelmien joukosta vain osa tulisi yritykseen uusina.

Vaatimusten hallinnan osalta tässä insinöörityössä suositellaan vaatimusten ensisijaisen tallennuspaikan ja hallintatyökalun tarkempaa määrittelyä. Ideaalisti tämän tulisi tukea kuvattua vaatimusten tilanseurantaa ja muutoksen hallintaa. Lisäksi tässä insinöörityössä kuvattua ei-toiminnallisten vaatimusten kehitysmenetelmää suositellaan. Siinä keskitytään kehittämään konkreettisia ja testattavia vaatimuksia, jotka perustuvat kehitettävälle järjestelmälle valittuihin tärkeimpiin laadullisiin piirteisiin (esimerkiksi tietoturva, käytettävyys, saatavuus).

Arkkitehtuurisuunnittelun osalta suositellaan ottamaan käyttöön esitetty valinnan, luonnin ja validoinnin sykli. Lisäksi C4-mallin käyttöä kehittämisnäkökulman luonnissa suositellaan, koska sitä pidetään helposti ymmärrettävänä, mutta toimivana. Kuvatussa mallissa ei kuitenkaan kielletä esimerkiksi UML-kaavioiden hyödyntämistä.

Sovelluskehityksen osalta kuvatun TDD-käytäntöön perustuvan mallin tarkempaa soveltamista suositellaan. Tähän liittyen myös automaattisen testauksen lisäystä varsinkin yksikkö- ja integraatiotestauksen osalta suositellaan.

Versionhallinta yrityksessä on insinöörityötä tehtäessä vaihdettu SVN-versionhallinnasta tässä insinöörityössäkin kuvattuun Git-versionhallintaan. Koska käyttöönotettu työkalu tukee tässä insinöörityössä kuvattuja versionhallinnan malleja (git-flow ja Github-flow), suositellaan niiden soveltamista tulevaisuuden ohjelmistokehityksenprosesseissa. Näistä suositaan Github-flow-mallia, paitsi jos kyseisessä kehitystyössä on oleellista varmistaa tuki useamman julkaisun ylläpidolle samanaikaisesti.

Myös konfiguraationhallinnassa automaatiota voitaisiin huomattavasti lisätä. Uusien jatkuvien palvelujen osalta automaatiopalvelimen laajaa käyttöä esimerkiksi variaatioidenhallinnassa, testauksen raportoinnissa, tilanseurannantyökaluna ja toistettavassa paketin rakentamisessa suositellaan. Lisäksi varsinkin jatkuvien palveluiden osalta suositellaan infrastruktuurin luomisen ja hallinnan automatisointiin kykeneviä työkalujen (esimerkiksi Ansible-työkalu) harkintaa. Julkaisunhallinnan osalta suositukset ovat samat kuin konfiguraationhallinnassa. Varsinkin uusien jatkuvien palvelujen osalta julkaisun toimituksen automatisointia entistä laajemmin suositellaan. Sovellusten kehittämistä alusta alkaen esimerkiksi Docker-säiliöihin tulisi harkita itse toimitustilanteen yksinkertaistamiseksi.

Tässä insinöörityössä nähdään kaksi tilaisuutta jatkokehitykselle. Ensimmäinen ilmenee siitä, että annettuja ehdotuksia ei ole ehditty toteuttamaan ja testaamaan käytännössä. Ehdotuksien käytännön toteutus jätettiinkin jo ajanpuutteen vuoksi pois insinöörityöstä jatkokehitysprojektiin. Lisäksi tässä insinöörityössä menetelmien ja lähteiden vertailu on vähäistä; kuvattujen menetelmien ei voida todeta olevan parhaita tai sopivimpia tilaajayritykselle. Tämä oli odotettavissa aihealueen laajuus ja ajan rajallisuus huomioitaessa. Edellä mainitun jatkokehitysprojektin lisäksi tulee uusien menetelmien tutkimista jatkaa myös tulevaisuudessa.

Lähteet

Albarqi, Aysha Abdullah & Qureshi, Rizwan. 2018. The Proposed L-Scrumban Methodology to Improve the Efficiency of Agile Software Development. International Journal of Information Engineering and Electronic Business. Vol 11, s. 23-25.

Ambler, W. Scott. 2005. Single Source Information: An Agile Core Practice for Effective Documentation. Verkkoaineisto. <http://www.agilemodeling.com/essays/singleSourceInformation.htm>. Luettu 17.3.2020.

Anda, Bente; Hansen, Kai; Gullesen, Ingolf & Thorsen, Hanne Kristin. 2006. Experiences from introducing UML-based development in a large safety-critical project. Empirical Software Engineering. Vol 11, s. 555-581. Berlin: Springer Science+Business Media.

Bank, Chris. 2014. Building Minimum Viable Products at Spotify. Verkkoaineisto <https://speckyboy.com/building-minimum-viable-products-spotify>. 18.9.2014. Luettu 27.3.2020.

Bass, Len; Clements, Paul & Kazman, Rick. 2012. Software Architecture in Practice, Third Edition. E-kirja. Addison-Wesley Professional.

Bhat, Thirumalesh & Nagappan, Nachiappan. 2006. Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies. Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering (ISESE ’06), s. 356-363. New York: Association for Computing Machinery.

Brown, Simon. 2020. The C4 model for visualising software architecture. Verkkoaineisto. <https://c4model.com>. Luettu 17.3.2020.

Canfora, Gerardo; Cimitile, Aniello; Garcia, Felix; Piattini, Mario & Visaggio, Corrado Aaron. 2006. Evaluating Advantages of Test Driven Development: a Controlled Experiment with Professionals. Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering (ISESE ’06), s. 364-371. New York: Association for Computing Machinery.

Choudary, Archana. 2019. Test Case in Software Testing. Verkkoaineisto. <https://www.edureka.co/blog/test-case-in-software-testing>. Päivitetty 28.11.2019. Luettu 27.4.2020.

Dart, Susan. 1991. Concepts in Configuration Management Systems. Proceedings of the 3rd international workshop on Software configuration management (SCM ’91), s. 1-18. New York: Association for Computing Machinery.

Driessen, Vincent. 2010. A successful Git branching model. Verkkoaineisto. <http://nvie.com/posts/a-successful-git-branching-model>. 5.1.2010. Päivitetty 5.3.2020. Luettu 21.3.2020.

Ford, Neal & Richards, Mark. 2020. Fundamentals of Software Architecture. E-kirja. O'Reilly Media, Inc.

Git Workflow. 2018. Verkkoaineisto. University of Wyoming. <https://arccwiki.uwyo.edu/index.php/Git_Workflow>. Päivitetty 7.2.2018. Luettu 16.4.2020.

Hauer, Philipp. 2019. Modern Best Practices for Testing in Java. Verkkoaineisto. <https://phauer.com/2019/modern-best-practices-testing-java>. Päivitetty 5.2.2020. Luettu 29.3.2020.

Hauer, Philipp. 2019. Focus on Integration Tests Instead of Mock-Based Tests. Verkkoaineisto. <https://phauer.com/2019/focus-integration-tests-mock-based-tests>. Päivitetty 16.11.2019. Luettu 29.3.2020.

Hellmann, Theodore D; Hosseini-Khayat, Ali & Maurer, Frank. 2010. Agile Interaction Design and Test-Driven Development of User Interfaces – A Literature Review. Agile Software Development, s.185-201. Berlin: Springer.

ISO/IEC 25010. 2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. International Organization for Standardization.

Karl, Wiegers. Joe, Betty. 2013. Software Requirements, Third Edition. E-kirja.

Kruchten, Philippe. 1995. Architectural Blueprints—The “4+1” View Model of Software Architecture. IEEE Software. Vol 12, no. 6, s. 42-50.

Loeliger, Jon & McCullough, Matthew. 2012. Version Control with Git, 2nd Edition. E-kirja. O'Reilly Media, Inc.

Nagappan, Nachiappan; Maximilien, E. Michael; Bhat, Thirumalesh & Williams, Laurie. 2008. Realizing quality improvement through test driven development: results and experiences of four industrial teams. Empirical Software Engineering. Vol. 13, no. 3, s. 289-302. Berlin: Springer Science+Business Media.

Niemelä, Jouni. 2006. Ohjelmistojen konfiguraationhallinta. Verkkoaineisto. <http://www.mit.jyu.fi/opetus/kurssit/jot/2006/luennot/SCM_luento.pdf>. 12.12.2006. Luettu 21.3.2020.

OpenUP. 2012. Verkkoaineisto. Eclipse Foundation, Inc. <https://www.eclipse.org/downloads/download.php?file=/technology/epf/OpenUP/published/openup_published_1.5.1.5_20121212.zip>. 1.6.2012. Luettu 27.4.2020.

Scott, Alister. 2018. Ideal Software Testing Pyramid. https://watirmelon.files.wordpress.com/2018/02/ideal-automated-testing-pyramid.jpg. Luettu 29.3.2020.

Thomas, Dave. 2014. Agile is Dead (Long Live Agility). Verkkoaineisto. <https://pragdave.me/blog/2014/03/04/time-to-kill-agile.html>. Päivitetty 3.11.2014. Luettu 17.3.2020.

The TOGAF® Standard, Version 9.2. 2018. Verkkoaineisto. The Open Group Standard. <https://pubs.opengroup.org/architecture/togaf9-doc/arch>. Luettu 27.4.2020.

Understanding the GitHub flow. 2017. Verkkoaineisto. GitHub, Inc. <https://guides.github.com/introduction/flow>. 30.11.2017. Luettu 21.3.2020.

What is a Container. 2020. Verkkoaineisto. <https://www.docker.com/resources/what-container>. Luettu 26.4.2020.

When to Use Test Driven Development. 2016. Verkkoaineisto. <https://www.jrebel.com/blog/when-to-use-test-driven-development>. 26.4.2016. Luettu 29.3.2020.

Woodward, Martin R. & Hennell, Michael A. 2005. Strategic benefits of software test management: a case study. Journal of Engineering and Technology Management. Vol. 22, no 1-2, s. 113-140. Elsevier.