Tampereen yliopisto

Pietari Pakarinen

MENETELMIÄ KESTÄVIEN OHJELMISTOJEN TUOTTAMISEEN

Tarkastelua artefaktien avulla

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Huhtikuu 2021

TIIVISTELMÄ

Pietari Pakarinen: Menetelmiä kestävien ohjelmistojen tuottamiseen
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Huhtikuu 2021

Ohjelmistojen ylläpito on ohjelmistokehityksen kallein vaihe. Kun ohjelmistoista tehdään alusta asti kestäviä, ylläpitovaiheessa säästyy resursseja. Tutkielmassa selvitän, mitä tiedeyhteisö on kirjoittanut ohjelmistojen kehittämisestä siten, että niiden ylläpito on mahdollisimman vähän työllistävää ja halpaa. Lähteitä hain tietojenkäsittelytieteen alan keskeisistä tietokannoista, ohjelmistoja kehittävien organisaatioiden sivuilta ja eri teknologioita ja toimintamalleja kehittäneiden tahojen sivuilta.

Tutkielman alussa esittelen joitakin keskeisiä käsitteitä. Sen jälkeen esittelen artefaktit. Artefaktien rooli kestävien ohjelmistojen tuottamisessa on keskeinen. Selvitin, että oikeaoppisella artefaktien hallinnalla on mahdollista purkaa ohjelmistokehitysprosessiin osallistuvien eri tahojen siiloja. Artefaktit ovatkin keskeinen kommunikaatioväline ohjelmistojen kehityksessä.

Seuraavissa kolmessa luvussa tarkastelen ohjelmistokehityksen elinkaaren eri vaiheita. Käsittelen erikseen vaiheen ennen ohjelmakoodin kirjoittamista, ohjelmakoodin kirjoitusvaiheen ja ohjelman ylläpitovaiheen. Kävi ilmi, että aikaisessakin vaiheessa luotuja artefakteja tarvitaan aina ylläpitovaiheessa asti, eikä silloin välttämättä ole enää mahdollista kommunikoida suoraan artefaktin kirjoittaneen tahon kanssa. Siksi artefaktit tulee säilöä huolellisesti.

Ohjelmistokehityksen elinkaarta käsittelevissä kappaleissa tutustuin myös teknologioihin ja ajatusmalleihin, joista on hyötyä kestäviä ohjelmistoja tuottaessa. Niitä on jatkuvasti kasvava määrä, eikä yhden työntekijän ole mahdollista hallita niitä kaikkia. Se johtaa haasteisiin, joihin tietojenkäsittelyn koulutuksen ja alan jo työskentelevien tulee vastata.

Ohjelmistokehitysala kehittyy nopeasti, ja sitä tutkitaan paljon. Tietoa löytyy runsaasti, joskin aivan uusimmista aiheista saattaa olla parempi etsiä tietoa epätieteellisistä julkaisuista, esimerkiksi teknologian tai toimintamallin kehittäneen tahon julkaisemasta dokumentaatiosta.

Avainsanat: Ohjelmistotuotanto, Kirjallisuuskatsaus, Artefaktit, Kestävä ohjelmistokehitys, Agile, DevOps

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

Sisällysluettelo

1 Johdanto **1**

2 Käsitteiden määrittelyä **1**

3 Artefaktit ohjelmistokehitysprojektin kommunikaatiossa **3**

4 Ohjelmistoprojektin valmistelusta ja suunnittelusta **8**

5 Kestäviä ohjelmistoja modernein, ketterin menetelmin **11**

6 Kestävää ohjelmiston ylläpitoa **13**

7 Keskustelu **14**

8 Yhteenveto **15**

Lähdeluettelo **16**

1 Johdanto

Ylläpito on ohjelmistokehitysprojektin kallein vaihe, se vie yhä enemmän ohjelmistokehittäjien aikaa ja sen vaikutukset ovat merkittävämmät kuin koskaan (Jin, 2005). Eri tapoja suunnitella ohjelmistojen kehittämistä on tutkittu paljon ja malleja ohjelmistojen kehittämiseen on runsaasti. Eri tapoja toteuttaa sovelluksia on paljon, joista nykyään ketterät menetelmät ovat hallitsevia. Ohjelmistojen ylläpito nähdään merkittävänä kulueränä, mutta yleispätevää mallia sen toteuttamiseen ei oikeastaan ole. Yksi tapa vähentää kustannuksia ohjelmistojen ylläpitovaiheessa onkin minimoida tarvittavien muutosten määrää ja kokoa jo aiemmissa vaiheissa tehtävillä ratkaisuilla.

Tutkielmassa tutkitaan, kuinka tehdä sellaisia ohjelmistoja, joiden muutos- ja ylläpitokustannukset pysyvät pitkällä tähtäimellä matalina. Sellaisia ohjelmistoja kuvaillaan kestäviksi ohjelmistoiksi. Tekemällä ohjelmistoista kestäviä, on mahdollista pitää niiden ylläpitokustannukset matalina. Tutkielmassa tutkin ohjelmistokehitysprosessia ja sen aikana tehtävien ratkaisujen vaikutusta lopputuloksena syntyvän ohjelmiston kestävyyteen.

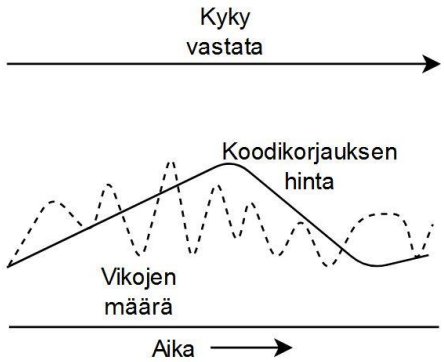
Johdannon jälkeisessä kappaleessa määrittelen joitakin tutkielmassa keskeisiä käsitteitä, joita en muualla tekstissä määrittele. Sen jälkeen käsittelen artefakteja. Selitän, mitä artefaktit ovat ja kerron niiden merkityksestä ohjelmistokehityksessä. Jatkan kertomalla ohjelmistokehityksen elinkaaren ja artefaktien avulla ohjelmistokehityksen eri vaiheista, ja kuinka niissä jokaisessa voi vaikuttaa prosessissa syntyvän ohjelmiston kestävyyteen. Etenen elinkaaren mukaisessa järjestyksessä.

Tekstin varsinainen sisältö on jaoteltu ohjelmistokehitysprosessin eri vaiheisiin. Eri vaiheiden kautta tarkastelen, mitä asioita voi tehdä ohjelmistokehitysprosessin aikana, jotta ohjelmiston julkaisunjälkeinen ylläpidon tarve vähenisi, eli ohjelmistosta tulisi kestävämpi. Jaottelen prosessin kolmeen vaiheeseen. Ennen ohjelmoinnin aloittamista, ohjelman rakentamisen aikana ja julkaisun jälkeen. Jakamalla tutkielman vaiheisiin, pystyn yhdistämään tietyt ohjelmiston kestävyyttä parantavat toimet niitä vastaaviin ohjelmistokehitysprosessin vaiheisiin. Tutkielman lopussa on keskustelua aiheesta ja yhteenveto tutkielman keskeisimmästä sisällöstä.

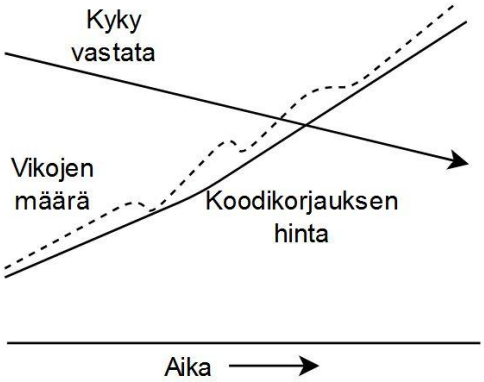
2 Käsitteiden määrittelyä

Tate (2006) määrittelee kestävän ohjelmistokehityksen ohjelmistokehitysmalliksi, jonka tuloksena syntyy ohjelmistoja, joiden ylläpitokustannukset ovat mahdollisimman pieniä. Itse ohjelmistokehitysmalli koostuu käytännöistä ja periaatteista, jotka mahdollistavat optimaalisen ohjelmistokehitystahdin jatkuvasti. Kestävä ohjelmistokehitys mahdollistaa

ohjelmiston vaatimien muutosten kustannusten ja määrän pitämisen matalana, samalla kun kyky korjata niitä pysyy vakiona, kuten kuvasta 1 näkyy. Kuvassa 2 on toisaalta tilanne, jossa tiimin kyky vastata tarpeisiin heikkenee ajan myötä, ja muutosten hinta ja määrä kasvaa.



Kuva 1 Koodimuutosten hinnan muutos vikojen pysyessä matalana sekä vastekyky (Tate, 2006)



Kuva 2 Koodimuutosten hinnan muutos vikojen kasaantuessa sekä vastekyky (Tate, 2006)

Ohjelmistokehityksen elinkaari (Software development life cycle, SDLC) on Ruparelian (2010) mukaan käsitteellinen viitekehys tai prosessi, joka huomioi sovelluksen kehityksen vaiheiden rakenteen sen varhaisesta toteutuskelpoisuudesta julkaisuun ja ylläpitoon asti. SDLC-prosessin kuvaamiseen on useita malleja, joista Ruparelia kuvailee joitakin.

Agile Alliance järjestön mukaan (Agile Manifesti, 17.4.2021) Agile on kattotermi joukolle viitekehyksiä ja menetelmiä, jotka pohjautuvat Agile manifestiin ja sen taustalla olevaan kahteentoista periaatteeseen. Agile menetelmiä kutsutaan ketteriksi menetelmiksi.

Käsittelen tutkielmassa ketteriä menetelmiä. Niistä esimerkki on Scrum. Schwaber & Sutherlandin (Scrum guide, 2020) mukaan, Scrum on kevyt viitekehys, joka auttaa ihmisiä, tiimejä ja organisaatioita tuottamaan arvoa ratkaisemalla kompleksisia ongelmia joustavasti. Tiivistetysti sanottuna Scrum tarvitsee Scrum Masterin luomaan ympäristön, jossa:

1. Tuoteomistaja järjestää kompleksisen ongelman ratkaisemiseen tarvittavan työn tuotteen kehitysjonoon.
2. Scrum-tiimi tuottaa sprinttiin valitsemastaan työstä arvoa tuottavan inkrementin.
3. Scrum-tiimi ja sen sidosryhmät tarkastelevat tuloksia ja mukauttavat sen pohjalta seuraavan sprintin toimintaa.
4. Toistetaan kohtia 1–3.

Virallista määritelmää täydentääkseni, Scrumin kehitysjakso eli sprintti on ajanjakso, jonka aikana tehdään tiettyjä Scrumiin kuuluvia vaiheita. Inkrementti taas on konkreettinen askel kohti tuotteen tavoitetta.

3 Artefaktit ohjelmistokehitysprojektin kommunikaatiossa

Ennen kuin tarkastellaan itse ohjelmistokehitysprosessia, on syytä mainita artefaktit. Artefaktien tunnistaminen, luominen, hallinta ja jakaminen ovat keskeisiä osia kestävää ohjelmistokehitysprosessia. Tässä luvussa esitellään väitteelle perusteet.

Ohjelmistokehitysprosessin aikana syntyy paljon muutakin kuin pelkkää ohjelmakoodia. Tällaisia ohjelmistokehityksen sivutuotteita kutsutaan artefakteiksi. Correria (2013) esittelee artefaktit ohjelmistokehittäjien ohjelmistokehityksen aikana keräämänä tietona. Hänen mukaansa artefakteihin kuuluu kaikki lähdekoodista malleihin ja dokumentaatioon. Artefaktit ovat hänen mukaansa paitsi keskeinen osa rakenteilla olevaa ohjelmistoa, myös kriittisiä tiimien kommunikaatiolle ja tiedon säilyttämiselle tulevaa käyttöä varten.

Hänen mukaansa näitä artefakteja syntyy ja ne kehittyvät SDLC:n aikana keskenään eriaikaisesti. Hän huomauttaakin, että tiimit usein keskustelevat ideoista, joista syntyy vapaamuotoisia sisältöjä ja vasta sitten muodostuu konkreettisia artefakteja.

Downey & Power (2007) sanovat artefakteista, että jos selvitettäisiin, mitä tietoja, taitoja ja kykyjä tarvitaan tietyn artefaktin tuottamiseen, olisi mahdollista rekrytoida osaajia, joiden tiedot, taidot ja kyvyt vastaavat nykyistä paremmin tarpeita. Sittemmin onkin syntynyt useita uusia rooleja henkilöille, jotka osallistuvat ohjelmistojen kehittämiseen, mutta keskittyvät nimenomaan joihinkin artefakteihin. Esimerkkejä tällaisista uusista rooleista on muun muassa DevOps-insinööri ja Cloud-insinööri.

Downey & Power kertovat artefaktien merkityksestä ohjelmistokehitysprosessissa tarkemmin. He esittävät idean, että ohjelmistokehitys on erinäisten artefaktien, mukaan lukien itse ohjelman, tuottamista. Kun ohjelmistokehitystä lähestytään tästä näkökulmasta,

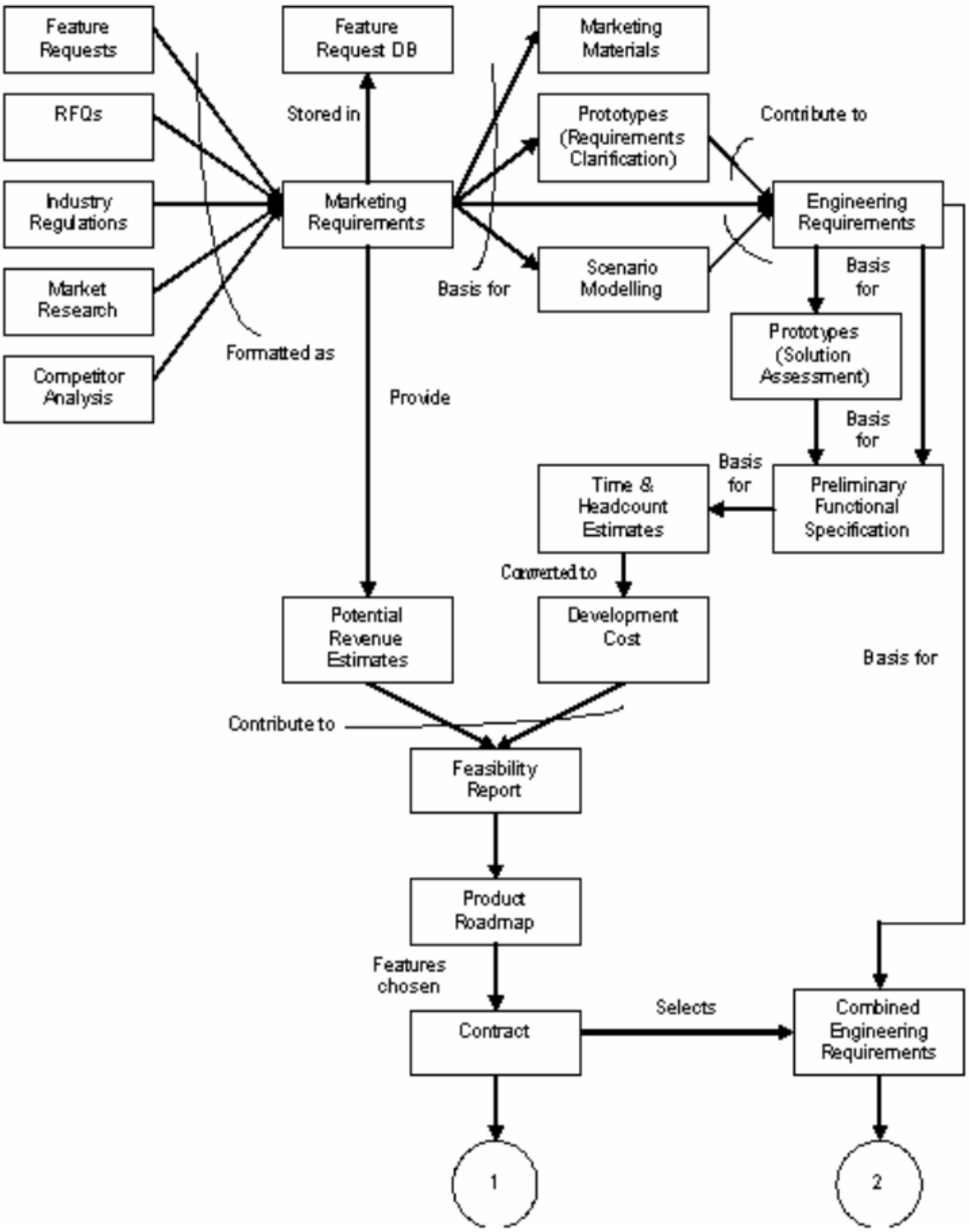
saavutetaan parempi ymmärrys ohjelmistokehitykseen liittyvistä tehtävistä ja sitä kautta vaadittavista tiedoista, taidoista ja kyvyistä. He myös toteavat, että nykyisellään ohjelmistokehitykseen liittyvät roolit vaihtelevat paljon eri yritysten ja jopa eri projektien välillä. Siksi ohjelmistokehitysrooleja ei pystytty määrittelemään yleisesti sovellettavalla tavalla.

Mambrey & Robinson (1997) määrittelevät artefaktit reunaolioiksi (boundary object). Nämä oliot “kansoittavat useita toisiaan leikkaavia sosiaalisia maailmoja, ja tyydyttävät jokaisen informationaaliset tarpeet”. Downeyn & Powerin artikkeli käsittelee juuri tällaisia artefakteja, jotka tarjoavat rajapintoja kehitystiimin ja organisaation muiden osastojen ja jopa muiden organisaatioiden välille. Artefaktit ovatkin tärkeitä juuri siksi, että niiden avulla on mahdollista kommunikoida ideoita eri osastojen välillä. Tällainen kommunikaatio on tärkeä osa ohjelmistokehitysprosessia. Jos organisaatio on siiloutunut, eli tiimit, osastot tai organisaatiot eivät keskustele keskenään, on havaittu, että tehokkuus heikkenee.

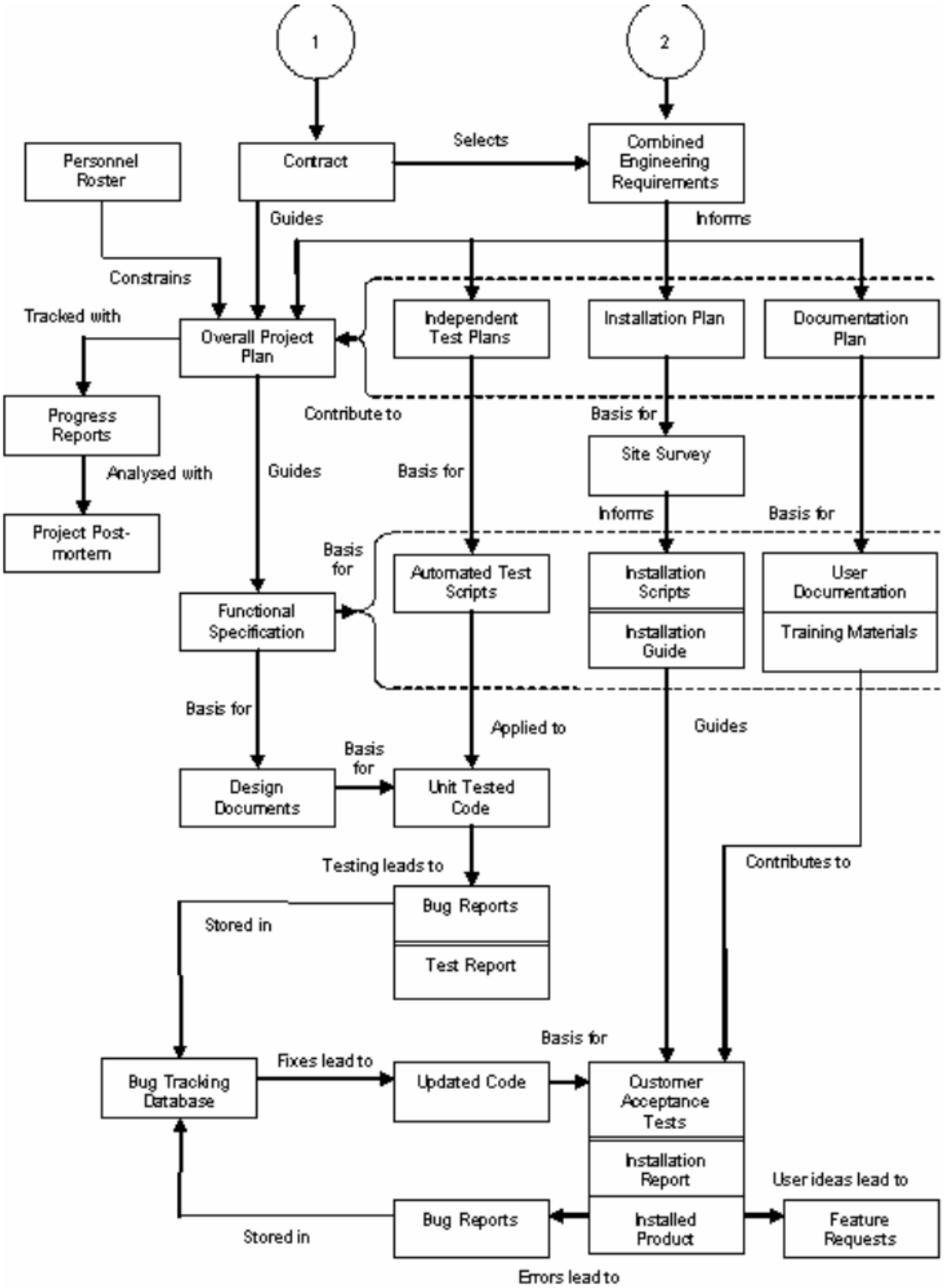
Esimerkkinä siiloutumisesta johtuneista tuottavuusongelmista, ohjelmistokonsulttitalo Eficode esittelee case-raportissaan (Eficoden blogi, 2020), kuinka autonvalmistaja Volvon ohjelmistokehitystiimit olivat siiloutuneet seitsemään eri siiloon. Siilot käyttivät erilaisia työkaluja samojen asioiden tekemiseen, mikä hidasti tuotantoa ja johti ohjelmistokehitysprosessin merkittävään hidastumiseen. Artefaktien luominen, säilöminen ja jakaminen on tällaisten tilanteiden välttämiseksi tärkeä osa ohjelmistokehitysprosessia.

Kommunikaation tärkeys korostuu myös siinä, ettei Downeyn & Powerin mukaan kukaan yksi kehitystiimin jäsen osallistu kaikkien artefaktien luomiseen. Kaikkien tietyn ohjelmiston osan kanssa työskentelevien on kuitenkin oltava perillä siihen liittyvistä artefakteista, joten on järkevää, että kaikilla kehittäjillä on pääsy artefaktirepositorioon, missä niiden tulisi olla jäsennelty ymmärrettävällä tavalla.

Downey & Power tunnistavat, että artefakteja on enemmän, kuin mitä kukaan yksi ihminen pystyy edes käsittelemään. He tekivät niistä diagrammin (kuvat 3 ja 4). Kuvasta näemme, kuinka artefakteja on paljon ja monenlaisia, vaikka esimerkiksi kaikki suunnitteludokumentit on kuitattu laatikolla, jossa lukee "Design documents", vaikka senkin voisi vielä eritellä lukuisiksi eri artefakteiksi.



Kuva 3 Ensimmäinen puolikas artefaktidiagrammista (Downey & Power, 2007)



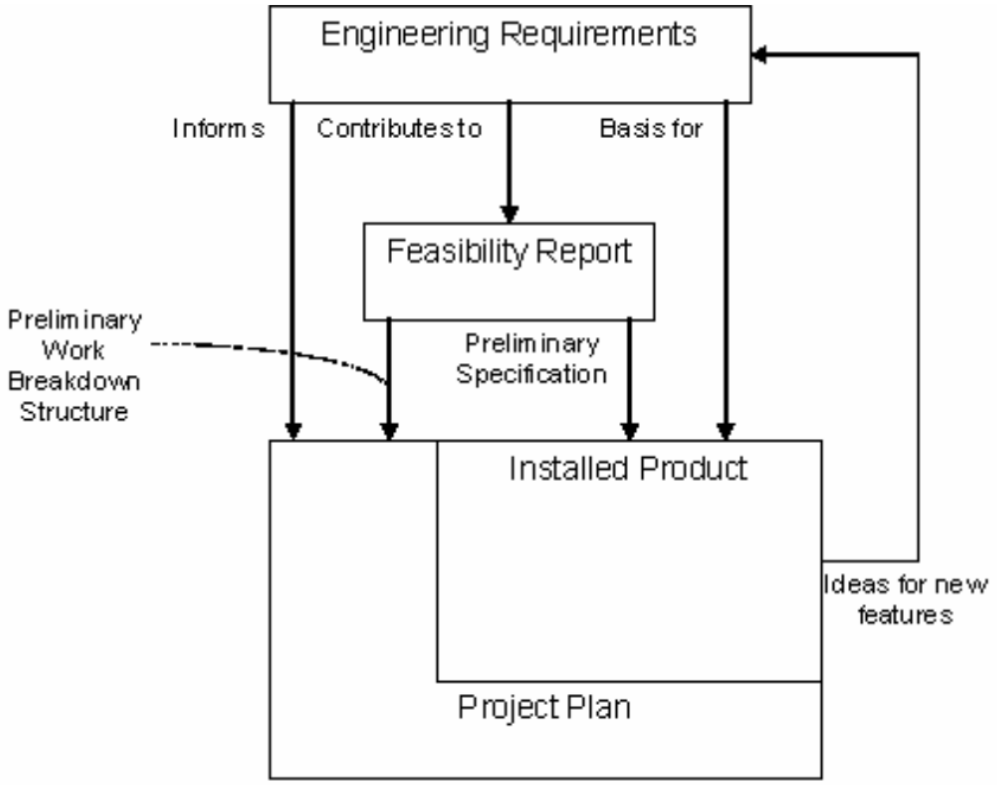
Kuva 4 Artefaktidiagrammin toinen puolikas (Downey & Power, 2007)

Downeyn & Powerin diagrammi on erittäin monimutkainen, mutta he myös jakoivat ohjelmistokehityksen neljään eri vaiheeseen, jotta olisi mahdollista luoda korkeamman ta-

son kuva artefakteista. Ne ovat määrittely, valinta, hallinta ja rakentaminen. Heidän mukaansa projektin alussa käytetään runsaasti aikaa, jotta saadaan tunnistettua, millaista kysyntää markkinoilla on. Määrittelyvaiheen lopputuloksena syntyy vaatimusmäärittelydokumentti.

Valintavaiheessa valitaan, mitä vaatimusmäärittelydokumentissa eritellyistä ominaisuuksista kehitetään. Valinta perustuu ominaisuuksien rahalliseen tuottavuuteen ja asiakastyytyväisyyden maksimoimiseen. Myös ominaisuuksien tekninen toteutettavuus huomioidaan. Kaikki nämä eri kriteerit esiintyvät toteutuskelpoisuusraporteissa (feasibility reports). Näissä raporteissa on sisältöä esimerkiksi myynti ja markkinointi tiimeiltä sekä teknisiltä tiimeiltä. Kun toteutuskelpoisuus raportissa todetaan ominaisuuden olevan toteutuskelpoinen, se asetetaan tuoteaikajanalle.

Hallintavaiheessa projektin julkaisu on määritelty ja ominaisuudet on hyväksytty. Sitten sen rakennusta ohjaa projektisuunnitelma. Lopulta rakennusvaiheessa syntyy artefakteja, jotka liittyvät ohjelmiston tuottamiseen ja lopputuotteen asentamiseen. Kuvassa 5 kuvataan näihin vaiheisiin liittyvät artefaktit korkealla tasolla.



Kuva 5 korkean tason luokittelu ohjelmistokehitysprojektin artefakteille (Downey & Power, 2007)

Downey & Power huomauttavat vielä, että kuvan 5 artefakteista on huomioitava, kuinka moni eri rooli liittyi niiden syntyyn. He toteavatkin, että artefaktit ovat kommunikaatioväline, joka mahdollistaa yhteistyön eri osapuolten välillä ja tukee päätöksentekoprosessia, ja siten ne täyttävät Mambreyn & Robinsonin artefaktin määritelmän.

4 Ohjelmistoprojektin valmistelusta ja suunnittelusta

Edellisessä kappaleessa kuvasin Downeyn & Powerin neljä artefaktien luomiseen liittyvää työvaihetta. Näistä kolme - määrittelyvaihe, valintavaihe ja hallintavaihe - tapahtuvat ennen varsinaisen ohjelmoinnin aloittamista.

Kun kestävää ohjelmistokehitysprojektia aloittaa, tulee määritellä tarkasti, mitä lähdetään rakentamaan. Jos määrittelyvaiheessa ei synny riittävän kattavaa vaatimusmäärittelydokumenttia, joudutaan korjaukset tehdä usein vasta julkaisun jälkeen, jolloin niiden kustannukset ovat huomattavasti korkeammat. Tilanne on ilmentymä teknisestä velasta.

Ramasubbu & Kermer (2019) toteavat, että tekninen velka juontaa suunnittelun, kehityksen tai julkaisun aikana valituista oikopoluista, ja se maksetaan korkoineen viimeistään ohjelmiston ylläpitovaiheessa. Ramasubbun & Kermerin mukaan artefaktien huolimattomalla käsittelyllä on haitallisia vaikutuksia ohjelmistosysteemin luotettavuuteen ja pitkän tähtäimen kehitykseen. Kun Taten mukaan ohjelmisto määritellään kestäväksi, jos sen muutos- ja ylläpitokustannukset pysyvät pitkällä tähtäimellä mahdollisimman matalina, on selvää, että kestävää ohjelmistoa kehitettäessä tulisi välttää teknisen velan syntymistä.

Ryan & Wheatcraft (2017) kertovat, kuinka vaatimusmäärittelyä tulisi tehdä. He toteavat, että vaatimusmäärittelyn tarkoitus on muuttaa erinäisten entiteettien tarpeet muodolliseksi kieleksi siten, että entiteettien aikeet ovat kaikkien osapuolten selkeästi ymmärrettävissä. Entiteeteillä he tarkoittavat ohjelmiston vaatimuksia määritteleviä osapuolia. He myös esittelevät viitekehyksen, jota voi käyttää vaatimusmäärittelyn tekemisessä. Tällaisten viitekehysten käyttö onkin hyvä tapa välttää väärinymmärryksiä määrittelyvaiheessa, mistä voisi syntyä teknisen velan kaltaisia ongelmia SDLC:n myöhemmissä vaiheissa.

Pergl (2010) toteaa, että toteutuskelpoisuustutkimus lähtee käyntiin sillä, että vaatimusmäärittelyn tuloksia analysoidaan. Tutkimuksessa selvitetään, mitä resursseja on käytettävissä, ja mitä resursseja vaatimusten täyttäminen vaatii. Tämän analyysin perusteella muodostuu toteutuskelpoisuusraportti. Perglin mukaan tätä toteutuskelpoisuusraporttia, josta käytetään myös termiä vaihtoehtoanalyysi, käytetään projektin perustelemiseen. Se

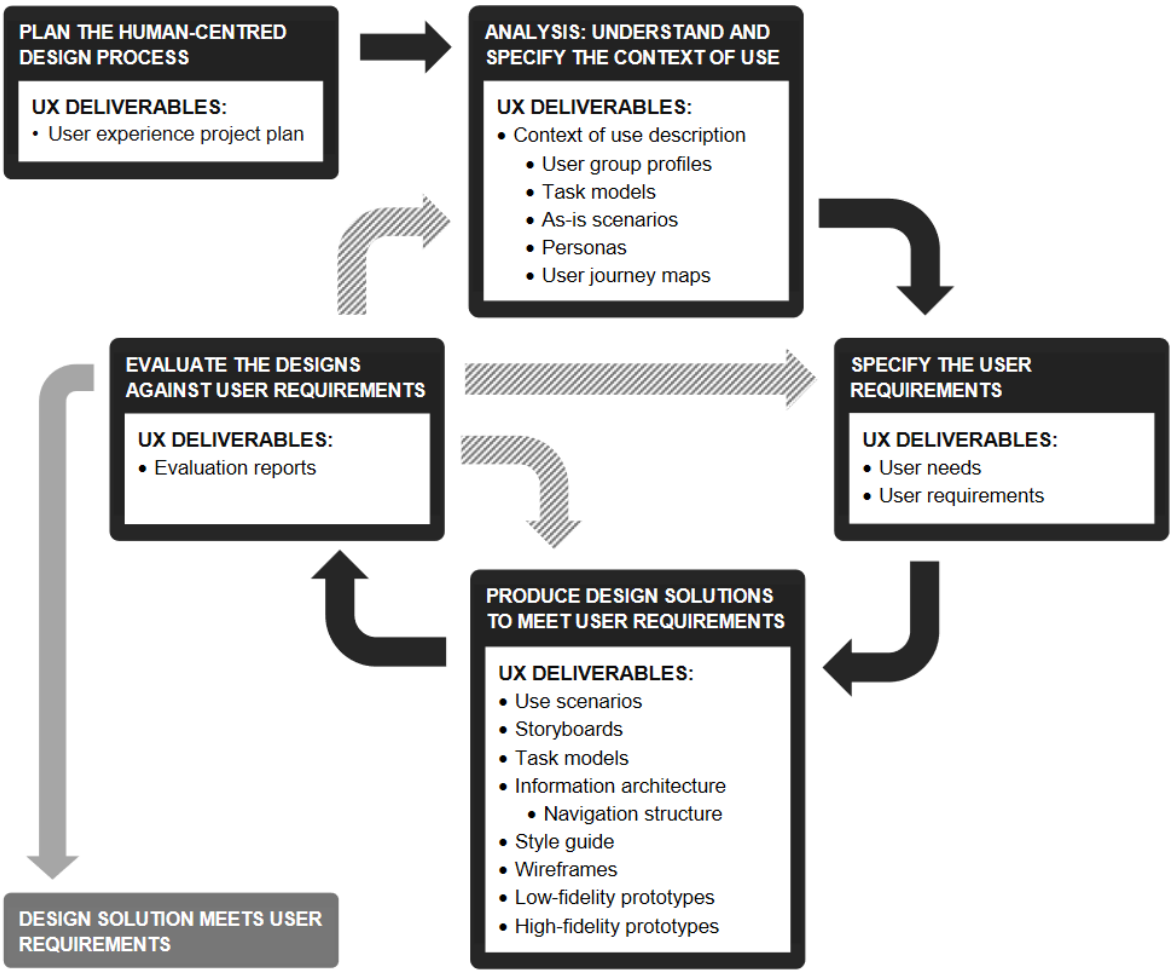
vertailee eri toteutusvaihtoehtoja niiden taloudellisen, teknisen ja operationaalisen toteutuskelpoisuuden perusteella. Pergl esittelee myös metodin, jota voi käyttää toteutuskelpoisuustutkimusten tekemiseen vaatimusmäärittelydokumentaation perusteella.

Kun on määritelty ohjelmistolle asetetut vaatimukset ja selvitetty toteutuskelpoisuusraportin perusteella, miten ja mitkä vaatimukset täytetään, on aika tehdä projektisuunnitelma. Projektisuunnitelmaan kuuluu Downeyn & Powerin mukaan aikataulu ja työn rakenteen erittely. Joissakin tapauksissa tehdään myös varasuunnitelmia, joilla varaudutaan tunnistettuihin riskeihin. Tämä tosin edellyttää, että on tehty riskianalyysi.

Suunnitelmien tekeminen on kuitenkin vaikeaa, ja Gueorguiev ja muut (2009) analysoivatkin, että projektien johto ei pysty ennakoimaan ennakoimatonta, ja heidän tulee varautua sudenkuoppiin tietämättä, mitä ne ovat. Siksi mitä huolellisemmin tehdään riskianalyysi, sitä enemmän sudenkuoppia on mahdollista tunnistaa, ja sitä paremmin niihin voidaan varautua. Toisaalta mitä enemmän aikaa käytetään riskien analyysiin ja projektin suunnitteluun, sitä kauemmin kestää, että tuote pääsee markkinoille. Tämä taas johtaa markkinaosuuden pienenemiseen ja kustannusten kasvuun Gueorguievin ja muiden mukaan. Organisaatioiden onkin tärkeä tasapainotella kestävyyden ja ketteryyden välillä, ja siihen tasapainotteluun Gueorguiev ja muut tarjoavat tutkimuksessaan työkalua.

Samanaikaisesti, kun tehdään projektisuunnitelmaa, suunnittelutiimi alkaa suunnitella, miltä ohjelmisto näyttää. Tavallinen suunnittelijatiimi koostuu ainakin käytettävyysasiantuntijasta, käyttöliittymäsuunnittelijasta, scrum masterista ja tuoteomistajasta. Käytännössä yksi työntekijä voi täyttää useita näistä rooleista. Toteutuskelpoisuustutkimuksessa on selvitetty myös, kuinka paljon resursseja suunnitteluun tulee laittaa, mikä määrittelee suunnittelutiimin koon.

Vaatimusmäärittelydokumentin avulla kommunikoidaan suunnittelijoille, mitä ominaisuuksia tulee toteuttaa. Toteutuskelpoisuusraportilla taas kommunikoidaan, miten paljon resursseja mihinkin käytetään, mikä on sekin arvokasta tietoa suunnittelijoille. Suunnittelijatiimi taas tuottaa runsaasti artefakteja, joita kehittäjät tulevat käyttämään toteuttaessaan ohjelmistoa. Näin artefakteilla kommunikoidaan eri siilojen välillä. Kuvassa 6 esitellään iteratiivinen suunnitteluprosessi, jota voi käyttää pohjana suunnittelu sprinteille. Se myös listaa, mitä artefakteja suunnitteluprosessissa syntyy.



Kuva 6 Iteratiivinen suunnitteluprosessi ja siinä syntyvät artefaktit (UXQB, 2020)

Maudet ja muut (2017) toteavat, että suunnittelijat ja kehittäjät usein työskentelevät toisistaan erillään ja projektiin kuuluukin usein vaihe, jossa suunnittelijat luovuttavat projektin kehittäjätiimille. Tässä on vaarana, että keskeistä informaatiota jää kommunikoimatta. He myös toteavat, että koska suunnittelijoilla ja kehittäjillä on eri osaamisalueet, heidän tulee kysyä toisiltaan palautetta, jotta suunnittelijat pystyvät tuottamaan toteutuskelpoisia suunnitelmia, ja jotta kehittäjät pystyvät toteuttamaan saavutettavia, tyylikkäitä ja hyvän käyttökokemuksen omaavia ohjelmistoja. Pelkkä artefaktien kautta kommunikointi ei riitä, vaan tarvitaan suoria tapaamisia suunnittelijoiden ja kehittäjien välillä sitten, kun joitakin suunnitteluartefakteja on luotu.

Maudet ja muut tunnistivat kolme suunnittelijoiden yleistä ongelmaa: Suunnittelijat eivät kommunikoi jotakin yksityiskohtaa, suunnittelijat eivät ota jotakin tilannetta huomioon ja suunnittelijat eivät ota teknistä rajoitetta huomioon. Nämä ongelmat ovat kuitenkin sellaisia, että ottamalla kehittäjät mukaan suunnitteluprosessiin, niiltä vältytään. Suunnitteluprosessi meneekin usein hieman päällekkäin ohjelmoinnin aloittamisen kanssa.

5 Kestäviä ohjelmistoja modernein, ketterin menetelmin

Downeyn & Powerin mukaan sovelluksen toteuttamisen lopputuloksena, eli rakennusvaiheen päätteeksi, syntyy toimiva ohjelmistotuote ja siihen mahdollisesti kuuluvat oppaat ja ohjeet. Ohjelma ei kuitenkaan ole valmis, kun se julkaistaan.

Tate (2006) huomauttaa, että kestävä ohjelmistokehitys sisältää kaksi ketterälle ohjelmistokehitykselle keskeistä konseptia - mukautuvien tiimien ja mukautuvien tuotteiden rakentamista. Tate toteaakin, että ohjelmistot eivät ole koskaan valmiita, vaan niihin on aina mahdollista tehdä muutoksia. Siksi niiden tulee olla mukautuvia ja muutosten tekemisen tulee olla halpaa, jotta ohjelmisto kestää ja täyttää sille asetetut vaatimukset. Jos ohjelmiston muokkaamisesta tulee liian kallista, se ei tule kestämään aikaa, vaan kilpailevat ohjelmistot korvaavat sen.

Ohjelmiston mukautuvuutta ja muokattavuutta kutsutaan ohjelmistoteollisuudessa skaalautuvuudeksi. Tate huomautti, että skaalautuvuus on ketterille menetelmille keskeinen ominaisuus, Rosenberg ja muut (2017) toteavat, että vaikka ketterien menetelmien käyttöönotto on johtanut merkittäviin edistysaskeliin ohjelmistokehityksen nopeuttamisessa, mutta varjopuolena ketterillä menetelmillä on usein juuri ongelmat ohjelmistojen resilienssin ja skaalautuvuuden kanssa.

Rosenberg ja muut ovat todenneet oman kokemuksensa perusteella, että ketterien menetelmien varhaiset versiot ovat todella ketteriä, mutta saavuttavat ketteryytensä huomioimalla vain hyvän päivän käyttötapaukset, aiheuttaen edellä mainitut varjopuolet. He toteavat, että lähestymistavat kuten Scrum ja Kanban auttavat minimoimaan ketterien menetelmien ongelmia, mutta paremmille ohjelmistokehitysmetodeille on tarvetta.

Rosenberg ja muut ovat kehittäneet oman mallinsa vastaamaan tähän tarpeeseen. Heidän mallinsa Resilient Agilen etuna on, että kehittäjät pystyvät implementoimaan sovelluksen eri osia rinnakkaisesti. Osien yhteensopivuuden varmistamiseksi käytetään muun muassa pientä määrää standardisoituja suunnitteluaskelia ja kommunikaatiota artefaktien välityksellä. Heidän kokeissaan useampi eri opiskelijaryhmä työsti samaa ohjelmistoa kolmen kuukauden jaksoina kahden vuoden ajan kommunikoimatta keskenään muuten kuin artefaktien välityksellä ilman suurempia ongelmia. Heidän kokeensa korostaa artefaktihallinnan roolia tiedon välittämisessä ja säilyttämisessä ohjelmistokehitysprosessin aikana.

Jotta voi siis tehdä kestäviä ohjelmistoja, täytyy ne tehdä siten, että ne skaalautuvat hyvin. Ketterät menetelmät ovat hyvä tapa tehdä ohjelmistoja, mutta kehitykselle on sijaa,

jotta ohjelmistokehitys ei ole vain nopeaa, vaan tuloksena syntyy myös kestäviä ohjelmistoja. Ohjelmistojen mukautuvuus ja muokattavuus paranee myös, jos otetaan käyttöön toisen jokseenkin tuoreen ohjelmistokehitysmenetelmän, DevOpsin, periaatteita.

DevOps, joka on yhdistelmä sanoista development ja operations, on vielä jokseenkin tuore termi, joka esiteltiin vuoden 2008 Agile-konferenssissa Torontossa. Sille ei ole yhtä oikeaa määritelmää. Diaz ja muut (2018) määrittelevät sen lähestymistavaksi, johon sisältyy ohjelmistokehityksen tavat, jotka kannustavat nopeaa ja usein tapahtuvaa ohjelmiston ominaisuuksien toimitusta samalla, kun varmistetaan ohjelmakoodin laatu ja tuotantoympäristön jatkuvuus. Vaikka määritelmiä on monia, ne yleisesti sisältävät koodin laadun varmistamisen ja tiuhan koodintoimitustahdin, yleensä eri tekniikoita, kuten koodin vertaisarviointia ja CI/CD-putkea hyödyntäen. Leiten ja muut (2019) määrittelevät continuous integration/continuous delivery -putken prosessiksi, jossa varmistetaan, että ohjelmiston repositorioon kommitoitu versio on aina toimiva.

DevOps-ajatusmallissa keskeistä on, että ohjelmistokehittäjät työskentelevät läheisesti organisaationsa IT-operaatiotiimin kanssa. Leiten ja muiden mukaan nykypäivänä ohjelmistoprojekteissa on joku, jota voi kuvata tittelillä DevOps-insinööri. Hänen tehtäväänsä kuuluu esimerkiksi tuotantoympäristön ja CI/CD-putken hallinta. Rooliin sisältyy käytännön DevOpsin toteuttaminen, eli kaikki toiminta, jolla voi nopeuttaa koodin toimitustahtia ja varmistaa sen laatua.

DevOps-insinöörien työnkuvaan kuuluu myös usein ohjelmointiympäristön pystyttäminen. Jotta ohjelmistoista voi tehdä todella skaalautuvia, on niiden hyvä olla alustasta riippumattomia, eli niiden tulee toimia laitteesta ja käyttöjärjestelmästä riippumatta. Sitä varten on kehitelty teknologioita kuten Docker-kontit ja niiden hallintaan Kubernetes.

Zhang ja muut (2018) toteavat, että kontittaminen (containerization) on vallankumous CI/CD työnkulkuun. He määrittelevät kontittamisen virtualisaatioteknologiana, joka mahdollistaa sovelluksen paketoimisen yhteen kaikkien sen riippuvuuksien ja ajoympäristön kanssa. Tämä paketti eli kontti on kevyt itsenäinen yksikkö, ja Docker on näiden konttien teollisuusstandardi.

Monimutkaisemmissa sovelluksissa yksi Docker-kontti ei riitä, vaan tarvitaan useampi. Vuonna 2014 Google julkaisi Kuberneteksen Docker-konttien hallinnoimiseksi. Kuberneteksen avulla Docker-konteista on mahdollista muodostaa verokosto, jonka jokaisesta solusta, joka koostuu konttiryhmästä, pääsee käsiksi kaikkiin muihin soluihin, joilla jokaisella on oma uniikki IP-osoitteensa. Sittemmin on julkaistu muitakin kontinhallintaympäristöjä, mutta Kubernetes on, kuten Docker, teollisuusstandardi.

DevOps-insinöörien lisäksi Cloud-insinöörit edistävät skaalautuvien ohjelmistojen tuotantoa. Ardagna ja muut (2014) huomauttavat, että pilviteknologiat ovat radikaalisti muuttaneet tapaa tehdä ohjelmistopalveluja. Pilviteknologia mahdollistaa nykyajan kestäville ohjelmistoille keskeisen skaalautuvuuden mahdollistavan ominaisuuden. Nyt on

mahdollista, ja usein ohjelmistotuotannon kannattavuudelle välttämätöntä, maksaa ohjelmistojen tarvitsemista pilvipalveluista sitä mukaa, kun niitä käyttää. Esimerkkeinä pilvipalveluista on vaikkapa ohjelmistojen tarvitsema laskenta ja ohjelmistojen hostaus.

Käsittelin kappaleessa runsasta määrää eri teknologioita, joita tulee käyttää, jotta projektin tuloksena syntyvistä ohjelmistoista tulee kestäviä, ja niiden ylläpito ja jatkokehitys on sujuvaa. Yksittäisen ohjelmistokehittäjän onkin haastavaa tehdä kestävää ohjelmistoa, kun hänen tulisi osata ainakin perusteet niin monesta eri asiasta.

6 Kestävää ohjelmiston ylläpitoa

Rehman ja muut (2018) toteavat, että ylläpitotyö on jopa puolet ohjelmistosysteemiin käytettävästä työmäärästä ja keskeinen vaihe ohjelmistokehityksen elinkaaressa. He huomauttavat, että Lehmanin evoluutiolain mukaan ohjelmistojärjestelmää (software system) tulee muuttaa jatkuvasti tai siitä tulee vähemmän hyödyllinen. Samalla he toteavat, että tutkijoiden ja insinöörien keskuudessa käydään paljon keskustelua siitä, kuinka suuren suosion saavuttanutta Agile-ohjelmistokehitysmenetelmää tulisi soveltaa ohjelmistojen ylläpitoon. Agilella ei ole sisäänrakennettua ohjelmistojen ylläpitoon tarkoitettua mekanismia. Rehman ja muut pyrkivät tutkimuksessaan kuvaamaan, kuinka ylläpitoa tulisi Agilella tehdä.

He huomauttavat, että ohjelmistojen ylläpidon voi jakaa neljään eri kategoriaan: mukautuvaan, korjaavaan, hiovaan ja ennaltaehkäisevään. Lisäksi he toteavat ohjelmistokehityksen ylläpidon elinkaaren jakautuvan neljään vaiheeseen. Kyseinen elinkaari alkaa esittelyvaiheesta, jossa ilmenee ensimmäiset ongelmat. Kasvuvaiheessa ilmenee syvemmän tason teknisiä ongelmia. Kypsyysvaiheessa saavutetaan merkittävää parannusta ja supistumisvaiheessa systeemiä paikkaillaan, kunnes se korvataan.

Cortellessa ja muut (2010) toteavat, että on tehty paljon tutkimusta metodeista ja työkaluista, jotka pystyvät ennustamaan ja arvioimaan ylläpitovaiheessa olevan ohjelmistojärjestelmän laatua. Laatu määritellään eri ominaisuuksien perusteella, kuten suoritusnopeuden tai luotettavuuden avulla.

Lenarduzzi ja muut (2017) tekivät systemaattisen kirjallisuuskatsauksen edeltävän neljänkymmen vuoden ajan ohjelmistojen ylläpitomallitutkimuksista. He totesivat, kuten Cortellessa ja muut, että aihetta on tutkittu paljon, mutta Lenarduzzi ja muut toteavat, että mallit ovat liian spesifejä, eikä vertaisarvioituja, eikä niistä ole juuri käytännön hyötyä. Niiden soveltaminen vaatii hyvin spesifin tilanteen ja osaamista, jota keskiverto ohjelmistokehittäjillä ei ole. Ohjelmistojen ylläpitomallien tutkimusala ei heidän mukaansa tuota mitään yleishyödyllisiä malleja, vaan aina vain kasvavan määrän erilaisia malleja.

Tapa miten ohjelmistoja ylläpidetään ei tule siis suoraan tieteen kentältä, vaan on muotoutunut ohjelmistokehitysteollisuudessa. Rehman ja muut toteavat, että ylläpitoa hoidetaan yleensä erillisillä ylläpitotiimeillä, jotka suorittavat tiettyjä tehtäviä ohjelmiston ylläpitämiseksi. Näitä tehtäviä suoritetaan, kunnes ohjelmisto korvataan. Tehtävät koostuvat jonkun käyttäjän tekemästä muutospyynnöstä, josta tehdään muutossuunnitelma, joka toteutetaan, verifioidaan, validoidaan ja julkaistaan.

Koska ylläpitotiimit koostuvat usein eri ihmisistä kuin ohjelmiston kehittänyt tiimi, tieto ylläpidettävän järjestelmän toiminnasta tapahtuu usein pitkälti artefaktien välityksellä. Ohjelmistojen ylläpito usein ulkoistetaan, eikä ylläpitäjillä ole välttämättä suoraa kontaktia kehittäjiin.

Koska ohjelmistojen ylläpito on niin työllistävä ja suuri osa SDLC-prosessia, sen työmäärän vähentämiseen on kehitetty erilaisia työkaluja. Dudekula ja muut (2012) totesivat systemaattisessa kirjallisuuskatsauksessaan, että testiautomaatio on kypsä tieteenala, mutta samaa ei voi sanoa teollisista testiautomaation sovelluksista. Sittemmin testiautomaatiota on otettu laajempaan käyttöön myös teollisuuden puolella.

Ramler ja muut (2014) kertovat, kuinka heidän tutkimuskohteensa on hyödyntänyt testiautomaatio viitekehystä (Test automation framework) siten, että jokainen muutos ohjelmistoon laukaisee automaattisen ohjelmakoodin kääntämisen ja testiprosessin. Siten voidaan varmistua siitä, ettei muutoksilla ole tahattomia sivuvaikutuksia. Tämä prosessi tapahtuu osana CI/CD-putkea, ja on yleistynyt jo teollisuusstandardiksi samalla, kun menetelmät kuten DevOps ja teknologiat kuten Jenkins ovat valloittaneet alan.

Vaikka ohjelmistojen ylläpito on työlästä ja suurin osuus ohjelmiston elinkaaresta, ylläpitoprosessia on pyritty tehostamaan, ja tätä työn määrää vähentämään. Tässä onkin pitkälti onnistuttu, mutta se on vaatinut erilaisia ulkoisia työkaluja, eikä yleistä, ainakaan mitään laajaa suosita saavuttanutta, ylläpitomenetelmää ole pystytty määrittelemään tiedeyhteisön toimesta. Uusista ohjelmistoista tulee joka tapauksessa nykyään yhä kestävämpiä, ja ne vaativat aiempaa vähemmän ylläpitoa. Ohjelmistojen ylläpito on kuitenkin mielestäni SDLC-prosessin tärkein kehityskohde.

7 Keskustelu

Ohjelmistokehitysprosessin aikana syntyviä artefakteja on tutkittu paljon. Kuitenkin nykyään tunnistetaan tarpeita uudenlaisille artefakteille, kuten esimerkiksi design systeemeille, joista ei ole vielä kovin paljoa tutkimustietoa, vaikka ne ovat laajassa käytössä. Jos tutkii jotakin periteisempää artefaktia, kuten vaatimusmäärittelydokumenttia, tutkimustietoa löytyy runsaasti. Kyseiset tutkimukset tarjoavat määrittelyjä, analyysejä ja viitekehyksiä, joita yritykset voivat hyödyntää tehdessään omia vaatimusmäärittelyjään. Vastaava pätee moniin muihinkin artefakteihin.

Uusimpien artefaktien tapauksessa kuitenkin parhaat lähteet löytyvät sen sijaan alan yritysten julkaisemasta dokumentaatiosta. Esimerkiksi, jos haluaa selvittää, kuinka tehdä hyvä design system, vaikuttaisi olevan hyvä ajatus tutkia kuinka olemassa olevat arvostetut design systemit on tehty. Niistä julkaistu tutkimustieto on tällä hetkellä varsin suppeaa, eikä juuri tarjoa tietoa, joka olisi hyödyllistä niiden suunnittelussa.

SDLC:n joka vaiheessa syntyy runsaasti artefakteja, ja jokainen vaihe vaatii erilaisia taitoja, joita kukaan yksi ihminen ei voi täysin hallita. Siksi johtotehtävät kuten scrummaster ja ohjelmistoarkkitehti ovat aina vain haastavampia rooleja, kun heidän tulee hallita tätä kaikkea. Myös tekijöiltä vaaditaan jatkuvaa opettelua, jotta he pysyvät mukana oman osaamisalueensa nopeassa kehityksessä. Koulutuksen on vaikea vastata tähän nopeaan kehitykseen, eikä tietojenkäsittelytieteen tutkinnoissa ole tilaa näiden asioiden opetteluun, vaan se on jokaisen alalle haluavan omalla vastuulla. Toisaalta laajeneva määrä rooleja ohjelmistokehitysprojekteissa lisää vaihtoehtoja alalle pyrkiville.

Yhden ihmisen onkin vaikea hallita kaikkea mitä kestävän ohjelmiston kehittäminen sisältää. Toisaalta ohjelmistot, joita pystyy kehittämään yksin ovat suhteellisen pieniä, joten niiden ylläpitokin on yksinkertaisempaa, eikä kestävyys ole silloin yhtä tärkeää kuin isojen järjestelmien tapauksessa. Jokaisen modernin ohjelmistokehittäjän on kuitenkin hyvä ainakin tietää erilaisista ohjelmistojen kestävyyttä edistävistä työtavoista ja työkaluista. Niistä merkittävintä kestävyysparannusta tarjoavista on hyvä opetella omienkin ohjelmistojen tekemistä varten perusteet. Start up -yritykset ja muut pienet ohjelmistoyritykset arvostavat kyseistä osaamista, sillä heidän harvoin on tarpeen soveltaa kaikkia kestävyystyökaluja syvällisesti.

Uusia menetelmiä ohjelmistojen kehittämiseen syntyy jatkuvasti. Viime aikoina alaa on vallannut Value Stream Management ajattelumalli. Sen tarkoituksena on tunnistaa ja mitata, missä vaiheissa syntyy arvoa ja mistä löytyy pullonkaulat kehityksessä. Siten on mahdollista allokoida resursseja oikeisiin paikkoihin. Malli myös työ bisnespuolen pois siilostaan ja auttaa heitä ymmärtämään mitä teknologiapuolen työntekijät tekevät. Muitakin uusia malleja ja työkaluja on runsaasti. Omasta mielestäni niihin kannattaa tarttua vasta, kun ne saavuttavat tietyn kriittisen massan käyttäjiä, jolloin niiden osaajille alkaa olla riittävästi kysyntää ja niiden ympärille alkaa muodostua kypsä yhteisö.

8 Yhteenveto

Ohjelmistokehitysprosessi on kehittynyt viime vuosina merkittävästi. Erityisesti ohjelmistojen kehittämisen aikana hyödynnetään runsaasti uusia teknologioita ja toimintamalleja. Jotkut teknologiat ja mallit mahdollistavat ohjelmistojen kehittämisen nopeammin, kun taas jotkut tuovat lisää vakautta ja vähentävät muutosten aiheuttamia tahattomia sivuvaikutuksia.

SDLC-prosessin kallein ja työläin vaihe on ylläpitovaihe. Kehittämällä kestäviä ohjelmistoja on mahdollista minimoida julkaistuun ohjelmistoon tehtävien muutosten määrä ja kustannukset. Ohjelmistoista on mahdollista kehittää kestävämpiä hyödyntämällä kirjoa eri ohjelmistokehitysmalleja ja –teknologioita SDLC-prosessin joka vaiheessa. Niitä kaikkia ei voi hallita kukaan yksi ihminen, vaan tarvitaan useita tiimejä, joissa on runsaasti erilaisia taitoja ja tietoja omaavia asiantuntijoita.

Jo ennen ohjelmakoodin kirjoittamisen aloittamista ja vielä pitkään ohjelmiston julkaisun jälkeen syntyy artefakteja. Artefaktien avulla tietoa välittyy tiimeiltä toisille, ja siten ohjelmistojen kehittämisprosessiin osallistuvat eri tiimit saavat tarvitsemansa taustatiedon. Artefakteja hyödyntämällä vältytään eri siilojen välisiltä ristiriidoilta. Artefaktien säilytys, hallinta ja käyttö ovatkin tärkeitä osia nykyaikaista ohjelmistojen kehittämistä, ja kaikilla prosessin osallisilla tulee olla pääsy heille relevantteihin artefakteihin. Artefaktit ovat nousseet yhä tärkeämpään rooliin, kun ohjelmistokehitysprosessiin osallistuvien asiantuntijoiden määrä ja roolikirjo ovat kasvaneet.

Lähdeluettelo

Agile Alliance (2021) https://www.agilealliance.org/agile101/ haettu 17.4.2021

C. A. Ardagna, E. Damiani, F. Frati, G. Montalbano, D. Rebeccani and M. Ughetti, *A Competitive Scalability Approach for Cloud Architectures*, 2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, 2014, pp. 610-617, DOI: 10.1109/CLOUD.2014.87.

CPUX-F Curriculum and Glossary, (2020) https://uxqb.org/public/documents/CPUX-F_EN_Curriculum-and-Glossary.pdf Haettu 07.04.2021

D. Jin, *Design Issues for Software Analysis and Maintenance Tools*, 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05), Budapest, 2005, pp. 115-117, DOI: 10.1109/STEP.2005.11.

Doug Rosenberg, Barry Boehm, Bo Wang, and Kan Qi. 2017. *Rapid, evolutionary, reliable, scalable system and software development: the resilient agile process.* In Proceedings of the 2017 International Conference on Software and System Process (ICSSP 2017). Association for Computing Machinery, New York, NY, USA, 60–69. DOI: https://doi-org.libproxy.tuni.fi/10.1145/3084100.3084107

Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. 2012. *Benefits and limitations of automated software testing: systematic literature review and practitioner survey.* In Proceedings of the 7th International Workshop on Automation of Software Test (AST '12). IEEE Press, 36–42.

Eficode (2021) https://www.eficode.com/cases/volvo-case haettu 17.4.2021

F. u. Rehman, B. Maqbool, M. Q. Riaz, U. Qamar and M. Abbas, *Scrum Software Maintenance Model: Efficient Software Maintenance in Agile Methodology*, 2018 21st Saudi Computer Society National Computer Conference *(NCC)*, Riyadh, Saudi Arabia, 2018, pp. 1-5, DOI: 10.1109/NCG.2018.8593152.

Filipe Figueiredo Correia. 2013. *Documenting software using adaptive software artifacts.* In Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity (SPLASH '13). Association for Computing Machinery, New York, NY, USA, 107–108. DOI: https://doi-org.libproxy.tuni.fi/10.1145/2508075.2514873

Jack Downey and Norah Power. 2007. *An artifact-centric framework for software development skills*. In Proceedings of the 2007 ACM SIGMIS CPR conference on Computer personnel research: The global information technology workforce (SIGMIS CPR '07). Association for Computing Machinery, New York, NY, USA, 186–195. DOI: https://doi-org.libproxy.tuni.fi/10.1145/1235000.1235042

Jessica Díaz, Rubén Almaraz, Jennifer Pérez, and Juan Garbajosa. 2018. *DevOps in practice: an exploratory case study*. In Proceedings of the 19th International Conference on Agile Software Development: Companion (XP '18). Association for Computing Machinery, New York, NY, USA, Article 1, 1–3. DOI: https://doi-org.libproxy.tuni.fi/10.1145/3234152.3234199

Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. 2019. *A Survey of DevOps Concepts and Challenges*. ACM Comput. Surv. 52, 6, Article 127 (January 2020), 35 pages. DOI: https://doi-org.libproxy.tuni.fi/10.1145/3359981

M. M. Cluts, *The Evolution of Artifacts in Cooperative Work: Constructing Meaning Through Activity*, ACM SIGGROUP 2003, Florida, USA, ACM Press, 2003, p. 144-152

M. Morisio, I. Stamelos and A. Tsoukias, *A New Method to Evaluate Software Artifacts Against Predefined Profiles*, ACM SIGSSEKE, Ischia, Italy, ACM Press, 2002, p. 811-818

N. Ramasubbu and C. F. Kemerer, I*ntegrating Technical Debt Management and Software Quality Management Processes: A Normative Framework and Field Tests*, in IEEE Transactions on Software Engineering, vol. 45, no. 3, pp. 285-300, 1 March 2019, DOI: 10.1109/TSE.2017.2774832.

Nayan B. Ruparelia. 2010. *Software development lifecycle models*. SIGSOFT Softw. Eng. Notes 35, 3 (May 2010), 8–13. DOI: https://doi-org.libproxy.tuni.fi/10.1145/1764810.1764814

Nolwenn Maudet, Germán Leiva, Michel Beaudouin-Lafon, and Wendy Mackay. 2017. *Design Breakdowns: Designer-Developer Gaps in Representing and Interpreting Interactive Systems*. In Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17). Association for Computing Machinery, New York, NY, USA, 630–641. DOI: https://doi-org.libproxy.tuni.fi/10.1145/2998181.2998190

P. Mambrey and M. Robinson, *Understanding the Role of Documents in a Hierarchical Flow of Work*, ACM SIGGROUP 1997, Phoenix, Arizona, ACM Press, 1997, p. 119-127

Robert Pergl. 2010. *Feasibility study inputs based on requirements engineering*. In Proceedings of the 6th International Workshop on Enterprise & Organizational Modeling and Simulation (EOMAS '10). CEUR-WS.org, Aachen, DEU, 121–132.

Ryan, M.J. and Wheatcraft, L.S. (2017), *On a Cohesive Set of Requirements Engineering Terms*. Syst Eng, 20: 118-130. https://doi-org.libproxy.tuni.fi/10.1002/sys.21382

Raturi, A., Penzenstadler, B., Calero, C., Richardson, D., Femmer, H., Franch, X. (2014) *Systematic mapping study on software engineering for sustainability*. https://doi-org.libproxy.tuni.fi/10.1145/2601248.2601256

Rudolf Ramler, Werner Putschögl, and Dietmar Winkler. 2014. *Automated testing of industrial automation software: practical receipts and lessons learned*. In Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation (MoSEMInA 2014). Association for Computing Machinery, New York, NY, USA, 7–16. DOI: https://doi-org.libproxy.tuni.fi/10.1145/2593783.2593788

Scrum guides (2021) https://scrumguides.org/ Schwaber & Sutherland, haettu 17.4.2021

Stefan Gueorguiev, Mark Harman, and Giuliano Antoniol. 2009. *Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering*. In Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO '09). Association for Computing Machinery, New York, NY, USA, 1673–1680. DOI: https://doi-org.libproxy.tuni.fi/10.1145/1569901.1570125

Tate, Kevin. (2006) Sustainable Software Development: An Agile Perspective

V. Cortellessa, R. Mirandola and P. Potena, *Selecting Optimal Maintenance Plans Based on Cost/Reliability Tradeoffs for Software Subject to Structural and Behavioral Changes*, 2010 14th European Conference on Software Maintenance and Reengineering, Madrid, Spain, 2010, pp. 21-30, DOI: 10.1109/CSMR.2010.15.

V. Lenarduzzi, A. Sillitti and D. Taibi, *Analyzing Forty Years of Software Maintenance Models*, 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, 2017, pp. 146-148, DOI: 10.1109/ICSE-C.2017.122.

Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. 2018. *One size does not fit all: an empirical study of containerized continuous deployment workflows*. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, 295–306. DOI: https://doi-org.libproxy.tuni.fi/10.1145/3236024.3236033