Extreme Programming ja arkkitehtuurisuunnittelu

Kenny Heinonen
Tietojenkäsittelytieteen laitos
Helsingin yliopisto
Helsinki, Suomi
kenny.heinonen@helsinki.fi

***Abstrakti*—Ketterä kehitys tuo haasteita arkkitehtuurisuunnittelulle, koska perinteisistä kehitysmenetelmistä poiketen ketterissä menetelmissä vältellään liiallista etukäteen tehtävää suunnittelua. Extreme Programming on yksi tunnettu ketterän ohjelmistokehityksen metodologia. XP:n periaate on, että suunnittelutyötä ei tulisi tehdä etukäteen muun muassa muuttuvien vaatimusten takia. Täten arkkitehtuurisuunnittelu jää vähälle huomiolle, jonka voidaan katsoa olevan yksi XP:n heikkouksista. Laajamittaisissa projekteissa arkkitehtuurisuunnittelu on tärkeää, koska järjestelmän perusrakennetta on vaikeaa ja kallista lähteä muuttamaan.**

Käytännössä XP:tä käyttävät ohjelmistokehittäjät toteuttavat etukäteen tehtävää arkkitehtuurisuunnittelua siitä huolimatta, että se rikkoo XP:n arvoja ja käytänteitä. XP ei poissulje erilaisia arkkitehtuuriin panostavia tekniikoita ja käytänteitä, kunhan ne noudattavat sen linjauksia. Lisäksi XP:ltä löytyy menetelmiä, joilla suunnittelutyötä voidaan tehdä ja järjestelmän arkkitehtuuria ohjata. Kysymykseksi jää onko edes tarpeellista tuoda XP:hen mitään uutta arkkitehtuurisuunnittelun saralla. Tässä artikkelissa tarkastellaan XP:n olemassaolevia käytänteitä, joilla on vaikutusta ohjelmiston arkkitehtuuriin. Myöhemmin käydään läpi myös erilaisia arkkitehtuurisuunnitteluun keskittyviä menetelmiä, jotka on tarkoitettu sulautettavaksi osaksi XP:tä. XP:n käytänteet tukevat näitä menetelmiä ja menetelmät itsessään kunnioittavat XP:n arvoja. Lopuksi näitä menetelmiä arvioidaan.

Avainsanat*—*extreme programming; arkkitehtuurisuunnittelu; arkkitehtuuri; ketterät menetelmät; xp

I. Johdanto

Ohjelmiston arkkitehtuuri on oleellinen osa ohjelmiston kehitystyötä. Arkkitehtuuri edustaa järjestelmän korkean tason rakennetta, ohjelmiston ydintä, jonka päälle muut komponentit rakennetaan. Laatutekijät ovat tapa ilmaista arkkitehtuurin laatua [1]. Esimerkiksi, jos ohjelmiston ylläpidettävyys ja testattavuus on hankalaa, on tämä yleensä merkki huonosta arkkitehtuurista. Hyvä arkkitehtuuri helpottaa uusien toiminnallisuuksien toteuttamista, joka antaa ohjelmistolle selkeän ja joustavan rakenteen. Etenkin laajoissa ja monimutkaisissa projekteissa arkkitehtuuri on avainasemassa.

Ketterät menetelmät ovat yleinen tapa kehittää ja tuottaa ohjelmistoja. Niiden tarkoitus on mahdollistaa kehitystyössä nopeat, usein tapahtuvat julkaisut sekä nopea reagointi muutoksiin [2]. Yhteisiä piirteitä eri menetelmien kesken ovat muun muassa iteratiivinen ja inkrementaalinen kehitys, kommunikoinnin suosiminen dokumentaation tuottamisen sijaan ja jatkuva suunnittelu (*continuous design*). Eräs suosittu menetelmä on Extreme Programming (lyh. XP), jota käytetään yleisesti eri ohjelmistoprojekteissa.

Extreme Programming on kevyt ja tehokas menetelmä, jonka tavoitteena on vähentää projektiin liittyviä riskejä, mukautua muuttuviin vaatimuksiin ja parantaa kehitysryhmän tuotteliaisuutta [3]. XP tekee tämän ohjeistamalla, että millä tavalla kehitystyötä tehdään. Esimerkiksi Scrum on erikoistunut projektinhallintaan sanelemalla millaisiin eri vaiheisiin työ jakautuu, mutta ei ohjeista miten näissä vaiheissa tehtävä työ teknisesti ottaen tulisi tehdä [4]. Sen sijaan XP:llä on joukko erilaisia hyväksi havaittuja käytänteitä ja arvoja, joita kehittäjien tulee noudattaa. Esimerkkejä näistä ovat pariohjelmointi, refaktorointi ja jatkuva integraatio. Menetelmässä painotus on keskittyä yksinkertaisuuteen ja tehdä vain niitä aktiviteetteja, jotka ovat kullakin ajan hetkellä ehdottoman tarpeellisia.

Ketterien menetelmien ongelma on, että ne soveltuvat huonosti isojen ja monimutkaisten ohjelmistojen kehittämiseen kattavan dokumentaation ja arkkitehtuurisuunnittelun puuttumisen johdosta [2, 5, 6]. Suunnittelutyötä ei tehdä alussa kokonaan muuttuvien vaatimusten takia, vaan se on asteittaista ja ohjelmiston rakenne kehittyy koko projektin ajan. XP noudattaa periaatetta nimeltä *YAGNI* (You Aren't Gonna Need It), joka painottaa, että kehittäjän tulisi tehdä vain niitä asioita, jotka ovat ajankohtaisia [3]. Myöhemmin tarvittavien toiminnallisuuksien suunnittelu ja toteutus tulisi tehdä vasta silloin, kun niitä aidosti tarvitaan. Tämä pätee myös arkkitehtuurin suunnitteluun. XP:hen voi kuitenkin tuoda uusia menetelmiä arkkitehtuurin suunnittelemiseksi, kunhan ne noudattavat XP:n vaalimia arvoja ja periaatteita.

Tämän artikkelin tarkoituksena on tarkastella XP:tä ja siihen liittyvää arkkitehtuurisuunnittelua. Luvussa 2 esitellään lyhyesti XP, sen arvot ja arkkitehtuurisuunnittelun osuus menetelmässä. Lisäksi tarkastellaan kuinka XP:tä on käytetty ohjelmistoalalla ja miten arkkitehtuuri on otettu projekteissa huomioon. Arkkitehtuurimenetelmistä, jotka rikkovat XP:n arvoja, esitetään perustelut niiden haitallisuudesta ketterälle kehitystyölle. Luvussa 3 esitellään erilaisia ohjelmiston arkkitehtuuriin keskittyviä menetelmiä, jotka voidaan sulauttaa osaksi XP:tä. Kukin esitetty menetelmä noudattaa XP:n arvoja ja ne tukevat sen käytänteitä. Luvussa 4 arvioidaan edellä esitettyjä menetelmiä. Luku 5 on yhteenveto edellisistä.

II. Extreme Programming

Extreme Programming on ketterän ohjelmistokehityksen metodologia, jonka tarkoituksena on kehittää ohjelmistoja kevyesti, tehokkaasti ja joustavasti [3]. Ketteränä menetelmänä XP:ssä kehitystyö jaetaan lyhyisiin iteraatioihin, joissa toteutetaan osa ohjelmiston vaatimuksista kerrallaan. Tämä edesauttaa varautumaan vaatimusten muutoksiin. Erityisesti

XP painottaa läheistä yhteistyötä kehittäjien sekä asiakkaan välillä, usein tapahtuvia julkaisuja ja itseorganisoituvuutta. Menetelmä sopii pienikokoisille, noin 2-10 hengen ryhmille. Sana *"extreme"* tulee XP:n tavasta viedä hyväksi havaittujen käytänteiden taso äärimmilleen, ottaen niistä kaiken hyödyn irti. Esimerkiksi koodin katselmointi vähentää ohjelmointivirheitä ja parantaa laatua, joten XP:ssä katselmointia tehdään jatkuvasti pariohjelmoinnin kautta.

Arkkitehtuurisuunnittelun rooli on XP:ssä vähäinen. Periaatteena on, että suunnittelutyötä tulisi tehdä mahdollisimman vähän myöhemmin toteutettaville vaatimuksille niiden muuttuvan luonteen takia [7]. Pääpaino on aina nykyhetken vaatimuksissa. Puutteellisen arkkitehtuurisuunnittelun katsotaan olevan XP:n heikkous. Kuitenkin XP:ltä löytyy erilaisia menetelmiä, jotka vaikuttavat arkkitehtuuriin ja joiden avulla suunnittelua voidaan tehdä.

Tämä luku jakautuu kahteen kokonaisuuteen: ensimmäiseksi käsitellään XP:n neljä arvoa, joita kehittäjien ja XP:n käytänteiden tulee noudattaa. Toiseksi käsitellään arkkitehtuurisuunnittelua XP:ssä ja siihen vaikuttavia XP:n oleellisimpia käytänteitä ja menetelmiä.

A. Arvot

XP koostuu neljästä arvosta, jotka korkeammalla tasolla ohjaavat kehitystyön suuntaa ja asettavat sille rajat. Uusien ja olemassaolevien käytänteiden sekä kehittäjien tulee noudattaa näitä arvoja kurinalaisesti. Nämä neljä arvoa ovat kommunikaatio, yksinkertaisuus, palaute ja rohkeus. On myös olemassa viides arvo, joka on kunnioitus, mutta se jätetään käsittelemättä. Näitä arvoja tarkastellaan, jotta saadaan käsitys siitä miten nämä arvot vaikuttavat uusien käytänteiden suunnitteluun ja sulauttamiseen osaksi XP:tä. Myöhemmissä luvuissa käsitellään erilaisia arkkitehtuurisuunnitteluun keskittyviä menetelmiä, joiden perustana ovat nämä arvot.

1) Kommunikointi: Perinteisissä ohjelmistotuotannon menetelmissä kattava dokumentaatio toimii tiedonlähteenä eri sidosryhmille. XP:ssä pyritään välttämään raskaan dokumentaation tuottamista, jolloin osapuolien välisellä kommunikoinnilla on merkittävä rooli. Kehittäjien tulee tietää toistensa tekemisistä ja heillä on hyvä olla sama kokonaiskuva ohjelmiston rakenteesta. Asiakkaan kanssa tulee tehdä läheistä yhteistyötä. Kommunikointia tukevia käytänteitä ovat muun muassa pariohjelmointi ja teknisten tehtävien estimointi.

2) Yksinkertaisuus: Extreme Programming korostaa mahdollisimman yksinkertaisten ratkaisujen tekoa. Kehittäjien tulee toteuttaa vain sen verran toiminnallisuutta, mitä sillä ajan hetkellä tarvitaan. Jos kehittäjät toteuttavat vasta myöhemmin tarvittavia toiminnallisuuksia, riskinä on, että tilanne muuttuu ja näitä toiminnallisuuksia ei tarvita enää mihinkään. Tällöin on tuhlattu aikaa turhien toiminnallisuuksien toteuttamiseen ja kehittäjien pitää tuhlata lisää työaikaa näiden toiminnallisuuksien poistamiseen tai muokkaamiseen.

3) Palaute: Palaute kertoo kehityksen sekä ohjelmiston tilasta ja XP:ssä se on jatkuvasti läsnä. Yksikkö- ja integraatiotestit toimivat konkreettisena palautteena, kertoen toimiiko ohjelmisto oikein. Kun kehittäjät estimoivat asiakkaan vaatimuksia, hän saa suoraa palautetta niiden laadusta. Toteutettavien vaatimusten seuraaminen kertoo ryhmälle heidän etenemisestään iteraation aikana. Asiakkaan ja testaajien kirjoittamat hyväksymätestit kertovat suoraan ohjelmiston nykytilasta. Lisäksi asiakas arvioi aikataulutusta suhteessa projektin etenemiseen ja tekee sen mukaan muutoksia. Palaute liittyy suoraan kommunikointiin ja yksinkertaisuuteen. Mitä enemmän saadaan edellä mainittua palautetta, sitä helpompaa projektista keskusteleminen on. Testit ja arviointi osoittavat milloin ohjelmistoa tulisi parantaa ja siten yksinkertaistaa.

4) Rohkeus: Rohkeus pitää ohjelmiston laadun korkeana. Jos koodissa havaitaan merkittävä vika ja sen korjaaminen aiheuttaa uusia vikoja, on parempi korjata kaikki kerralla kuin kirjoittaa lisää kömpelöä koodia päälle. Tämä vaatii rohkeutta, koska ihmisillä on usein taipumusta jättää asia sikseen, etenkin jos ohjelma on lähellä valmistumista. Rohkeutta vaatii myös ohjelmakoodin hylkääminen. Vaikka olisi kuinka kauan käyttänyt aikaa koodin tekoon, mutta se ei toimi tai on ratkaisuna huono, on parempi aloittaa kokonaan alusta. Rohkeutta tukee kaikki edelliset kolme arvoa. Kommunikoimalla voi ehdottaa riskialttiiden ratkaisujen kokeilemista. Yksinkertainen järjestelmä ei rikkoonnu helposti, jolloin on enemmän varaa tehdä kokeiluja. Palaute kertoo välittömästi ohjelmiston tilasta ja muutosten tekeminen on turvallisempaa.

B. Arkkitehtuurisuunnittelu

Yksinkertaisimmillaan XP on kurinalainen menetelmä, joka koostuu kokoelmasta äärimmilleen vietyjä käytänteitä, joita kehittäjien tulee noudattaa. XP:tä on kritisoitu sen tavasta ohjelmoida nopeasti yksinkertaisia ratkaisuja, jättäen arkkitehtuurin suunnittelun huomiotta [7]. XP:llä on kuitenkin muutamia käytänteitä, joilla ohjelmiston arkkitehtuuria voidaan kehittää. Nämä käytänteet ovat metafora, testivetoinen kehitys, jatkuva integraatio ja refaktorointi.

Metafora on yksinkertaistettu vertauskuva koko järjestelmästä ja sen toiminnasta [3]. Sen tarkoituksena on luoda kehittäjille ja asiakkaille yhteinen käsitys kehitettävästä ohjelmistosta ja sen arkkitehtuurista. Metafora kuvaa järjestelmän peruskomponentit ja niiden väliset suhteet. Jaettu vertauskuva ohjelmistosta helpottaa eri sidosryhmien välistä kommunikointia. Kun ohjelmistoa kehitetään, sen luokkien ja metodien nimeämiskäytännön tulisi juontaa juurensa metaforasta. Tällöin sekä asiakkaat että kehittäjät pystyvät helposti arvioimaan mitä mikäkin luokka tai metodi tekee. Kehittäjät vertaavat ohjelmiston rakennetta jatkuvasti metaforaan pohtien vastaako se käytännössä vertauskuvaa. Tämä auttaa kehittäjiä pitämään arkkitehtuurin koheesion korkeana.

Testivetoisen kehityksen eli TDD:n (*Test-Driven Development)* idea on yksinkertainen: ennen kuin lähdetään toteuttamaan ohjelmakoodia, kirjoitetaan sille testit etukäteen. Tämä varmistaa, että kaikki toiminnallisuus tulee testattua. Testien kirjoittamisen jälkeen kirjoitetaan itse ohjelmakoodi siten, että testit menevät läpi. Testin läpäisevän koodin tulee olla yksinkertainen: läpimenneet testit ovat tässä vaiheessa tärkeämpiä kuin koodin eleganttius. Läpäisyn jälkeen koodi refaktoroidaan siistiksi. Näitä askelia toistetaan koko kehitystyön ajan. Testejä kehitetään sitä mukaa, kun toiminnallisuutta aiotaan kasvattaa. Kattavat testit yhdessä jatkuvan integraation kanssa ovat ensiaskel arkkitehtuurin turvalliseen muokkaukseen. Jatkuva integraatio tarkoittaa

uusien toiminnallisuuksien integroimista ohjelmiston nykyiseen versioon siten, että integroinnin jälkeen uusi versio testataan ja varmistetaan sen toimivuus.

TDD ja jatkuva integraatio ovat turvamekanismeja arkkitehtuurin kehittämiselle. Todellisuudessa arkkitehtuuri kehittyy XP:ssä refaktoroinnin kautta. Refaktorointi tarkoittaa koodin sisäisen rakenteen parantamista ilman, että ulkoinen toiminnallisuus muuttuu. XP painottaa ohjelmiston yksinkertaisuutta ja refaktoroinnilla voidaan jälkeenpäin muuttaa toteutettuja ratkaisuja paremmiksi. Ohjelmiston yksinkertaisuus tarkoittaa XP:ssä seuraavia asioita:

1) Testit menevät läpi.
2) Ohjelmisto ei sisällä duplikaattikoodia.
3) Ohjelmakoodi ilmaisee käyttötarkoituksensa.
4) Ohjelmisto sisältää niin vähän luokkia ja metodeja kuin mahdollista.

Edellä mainitut asiat paljastavat milloin refaktorointi on tarpeellista. Refaktorointi muokkaa ohjelmiston rakennetta ja siten arkkitehtuuria. Mahdollisimman yksinkertaisen ohjelmiston kehittäminen johtaa selkeään ja hyvään arkkitehtuuriin.

Käytännössä hyvää arkkitehtuuria on kuitenkin vaikea saavuttaa XP:n tavalla [8]. Yksinkertaisen ratkaisun toteuttaminen ja sen hiominen jälkeenpäin tukemaan monimutkaisempia tavoitteita käy ajan myötä haastavaksi. Ohjelmistokehittäjät ovat kokeneet metaforan konseptin vaikeaksi ymmärtää ja siten eivät hyödy siitä. Refaktorointi taas ei sovellu hyvin laajamittaisten, rakenteellisten muutosten tekoon, vaan on tarkoitettu enemminkin matalan tason ratkaisujen hiomiseen. Martin Fowlerin mielestä alustavalle arkkitehtuurisuunnittelulla on sijansa XP:ssä [7]. Tällöin on tärkeää, että arkkitehtuuria ei pide-tä lopullisena ja alustavat suunnitelmat voidaan vielä peruuttaa tai muuttaa myöhemmin, mikä on ominaista ketterille menetelmille.

XP:ssä ei ole tarpeeksi ohjelmiston arkkitehtuuriin keskittyviä menetelmiä [1, 7, 8]. Refaktorointi ei ole aina riittävä hyvän arkkitehtuurin kehittämiseen. Tämän takia XP:tä käyttävissä eri projekteissa ollaan otettu mukaan arkkitehtuurisuunnitteluun keskittyviä menetelmiä. Eri tapauksissa menetelmät ovat kuitenkin lainattu suoraan perinteisistä, lineaarisen ohjelmistotuotannon menetelmistä, jotka ovat ristiriidassa yleisesti ottaen sekä ketterien menetelmien että XP:n vaalimien aatteiden kanssa [1, 9]. Kyseisille menetelmille on ominaista suunnitella arkkitehtuuri tyhjentävästi heti projektin alussa. Esimerkiksi eräällä menetelmällä nimeltä *ATAM/CBAM*[1] suunnitellaan arkkitehtuuri miettimällä ohjelmistolle asetettuja tavoitteita ja laatupiirteitä. Arkkitehtuuripäätöksiä arvioidaan luomalla niistä skenaarioita, riskiarvioita, kustannuksia ja niin edelleen. Menetelmä vaatii ja tuottaa paljon dokumentaatiota arkkitehtuurin arvioimiseksi ja kehittämiseksi. Tämä rikkoo XP:n kommunikoinnin arvoa ja on haitaksi ketterälle kehitystyölle, koska liiallinen dokumentaatio vähentää kasvokkain tapahtuvaa kommunikointia. Liiallinen suunnittelu ja aikaiset päätökset estävät myös varautumista muutoksiin.

[1]Architecture Tradeoff Analysis Method / Cost-Benefit Analysis Method [9]

Yhteensopimattomat menetelmät luovat tarpeen tuoda XP:hen uusia menetelmiä, joilla panostaa arkkitehtuuriin siten, että ne eivät ole haitaksi ketterälle kehitykselle. Menetelmien tulee siis noudattaa XP:n arvoja ja periaatteita. Näitä menetelmiä tarkastellaan seuraavassa luvussa.

III. Arkkitehtuurisuunnittelun menetelmiä ja käytänteitä osana XP:tä

XP:lle on kehitetty erilaisia käytänteitä ja menetelmiä, joilla arkkitehtuurisuunnittelun rooli nousee näkyvämmäksi. Extreme Programming sallii muiden menetelmien ja käytänteiden sulauttamisen osaksi sitä, jos ne eivät ole ristiriidassa XP:n arvojen ja periaatteiden kanssa. Seuraavaksi käydään läpi kolme menetelmää, jotka ovat kehittäjätason tarinat, jatkuva arkkitehtoninen refaktorointi ja arkkitehtuurikelpoistaminen. Kaikki edellä mainitut käytänteet noudattavat XP:n arvoja ja olemassaolevia käytänteitä.

A. Kehittäjätason tarinat

Kehittäjätason tarinat (*developer stories*) on menetelmä, jonka avulla kehittäjät voivat keskittyä ohjelmiston arkkitehtuuriin ja sen suunnitteluun [8]. Kehittäjätason tarinat muistuttavat käyttäjätason tarinoita (*user stories*), mutta kehittäjätason tarinoissa ohjelmistossa tarkastellaan niitä ominaisuuksia, jotka ovat vain kehittäjille näkyviä. Niiden tarkoituksena on parantaa arkkitehtuurin laatua kuvaamalla niitä ohjelmiston osia, joita tulisi refaktoroida.

Käytännössä tarinoita luodaan ja käsitellään ennen uuden iteraation alkua. Periaate on sama kuin käyttäjätason tarinoita luodessa: kehittäjät yhdessä pohtivat nykyisen rakenteen ongelmia ja muuntavat ne kehittäjätason tarinoiksi. Tarinat estimoidaan ja priorisoidaan. Asiakkaalle selitetään mistä tarinoissa on kyse ja hän pääsee valitsemaan mitkä kehittäjätason tarinat käyttäjätason tarinoiden lisäksi otetaan iteraatioon mukaan.

Tarinoilla on kaksi tarkoitusta: niillä voidaan suunnitella ja ilmaista konkreettisesti mitä ohjelmakoodissa pitäisi refaktoroida. Toinen tarkoitus on rakentaa ymmärrystä ohjelmiston arkkitehtuurista kehittäjien kesken. Kun kehittäjät luovat tarinoita, he pääsevät yhdessä miettimään ohjelmiston rakennetta. Ymmärrys arkkitehtuurista edesauttaa ohjelmointia ja rakenteen pysymistä yhtenäisenä.

B. Jatkuva arkkitehtoninen refaktorointi

Jatkuva arkkitehtoninen refaktorointi (*Continuous Architectural Refactoring*, lyh. CAR) on menetelmä, jolla tunnistetaan kehitettävästä ohjelmistosta erilaisia arkkitehtuuria heikentäviä rakenteita ja etsitään ratkaisuja niihin [1]. Arkkitehtuurin heikkoudet huomataan, kun ne aiheuttavat kehitystyössä hankaluuksia. Esimerkiksi huono arkkitehtuuri vaikeuttaa testaamista ja ylläpidettävyyttä. Menetelmä näkee ohjelmiston arkkitehtuurin laatutekijöinä. Laatutekijöistä tarkastellaan vain niitä, jotka ovat kehittäjille tärkeitä ja joilla on vaikutusta kehitystyöhön. Näitä laatutekijöitä ovat muun muassa muokattavuus, testattavuus, ymmärrettävyys, uudelleenkäytettävyys ja ylläpidettävyys.

Jatkuva arkkitehtoninen refaktorointi tuo arkkitehdin roolin XP:hen, jonka vastuulla on pitää ohjelmiston arkkitehtuuri

hyvänä. Iteraation aikana kehittäjillä on tavallisesti valittu lista vaatimuksia ja niihin liittyviä teknisiä tehtäviä, jotka tuovat uutta toiminnallisuutta ohjelmistoon. Kun kehittäjät ovat toteuttaneet tietyn määrän testattua toiminnallisuutta ja integroineet sen ohjelmiston uusimpaan versioon, heidän tehtävänä on laatia yksinkertainen, korkean tason malli toteuttamistaan toiminnallisuuksista. Arkkitehti vastaanottaa mallit ja yhdistää ne aikaisemmin saatuihin malleihin, rakentaen kuvaa ohjelmiston tämänhetkisestä arkkitehtuurista. Kun uudet mallit ovat yhdistetty arkkitehtuurimalliin, arkkitehti analysoi sitä ja etsii mahdollisia arkkitehtuuria heikentäviä piirteitä. Käytännössä arkkitehtuuria tarkastellessa arkkitehti on valinnut muutaman edellä mainitun laatutekijän, joita vasten arkkitehtuuria arvioidaan. Jos arkkitehti löytää heikkouksia, hän kehittää niille ratkaisut ja määrittelee ne teknisiksi tehtäviksi, jotka voidaan ottaa nykyiseen iteraatioon mukaan tai jättää myöhemmille iteraatioille toteutettaviksi. Arkkitehtoninen refaktorointi on jatkuvaa, koska läpi iteraatioiden kehittäjät luovat arkkitehdille malleja, joita hän yhdistelee nähdäkseen kokonaiskuvan arkkitehtuurista, etsien vikoja ja kehittäen ratkaisuja.

Jatkuva arkkitehtoninen refaktorointi hyödyttää arkkitehtuurisuunnittelua ja se on helppo sulauttaa osaksi XP:tä. Kehittäjien ei tarvitse opetella mitään uutta, riittää palkata arkkitehti, jolla on kohtuullinen osaaminen mallintamisesta ja joka osaa tunnistaa huonon arkkitehtuurin piirteitä ja tietää miten ratkaista ne. Käytäntö on kehittäjätason tarinoita vielä pidemmälle vietyä, koska arkkitehti etsii ongelmia ja ratkaisuja läpi iteraation. Ongelmiin pureudutaan heti ja siten arkkitehtuuri ei pääse huonontumaan.

C. Arkkitehtuurikelpoistaminen

Arkkitehtuurikelpoistaminen (*Real Architecture Qualification*, lyh. RAQ) on jatkoa jatkuvalle arkkitehtoniselle refaktoroinnille [1]. Siinä missä jatkuva arkkitehtoninen refaktorointi keskittyi kehittäjille oleellisiin laatutekijöihin, arkkitehtuurikelpoistamisen tavoitteena on pitää käyttäjälle oleellisten laatutekijöiden taso korkealla. Käyttäjälle oleelliset laatutekijät ovat ulospäin näkyviä ominaisuuksia, jotka voi huomata ohjelmistoa käyttäessä. Näitä laatutekijöitä ovat esi-merkiksi suorituskyky, käytettävyys ja luotettavuus.

Arkkitehtuurikelpoistamiseen kuuluu kokous, joka pidetään XP:ssä iteraation lopuksi. Kokoukseen saapuvat projektiin kuuluvien eri sidosryhmien edustajia, kuten kehittäjät sekä asiakas. Kokouksessa pohjustetaan kaikille osallistujille millainen ohjelmiston nykyinen arkkitehtuuri on ja mitä mahdollisia muutoksia siihen on tulossa arkkitehdin päätöksestä johtuen. Kokoukseen on valittu joukko laatutekijöitä, joista jokainen käydään läpi yksitellen ja keskustellaan toteuttaako ohjelmisto nykytilassaan kyseisen laatutekijän tarpeeksi hyvin. Jos ohjelmisto ei toteuta jotain laatutekijää riittävällä tasolla, osallistujat kehittävät arkkitehtonisia ratkaisuja ja määrittelevät ne teknisen tason tehtäviksi. Kyseiset tehtävät toteutetaan tulevissa iteraatioissa.

Arkkitehtuurikelpoistamisessa on samoja hyviä puolia kuin käyttäjätason tarinoissa: osallistujat pääsevät yhdessä keskustelemaan arkkitehtuurista ja miettimään sille parannusehdotuksia, jolloin ymmärrys arkkitehtuurista osallistujien kesken paranee. Lisäksi valittuja laatutekijöitä arvioidaan vasten ohjelmiston uusinta versiota, joka on etu. Esimerkiksi suorituskykyä on helppo arvioida, kun on olemassa konkreettinen versio ohjelmistosta, jolla sitä voi testata.

IV. Pohdinta

Viime luvussa esiteltiin kolme arkkitehtuurisuunnitteluun keskittyvää menetelmää: kehittäjätason tarinat, jatkuva arkkitehtoninen refaktorointi ja arkkitehtuurikelpoistaminen. Menetelmät ovat poikkeuksellisia, koska arkkitehtuuriin panostamisen lisäksi ne pyrkivät noudattamaan XP:n arvoja ja käytänteitä. Täten ne ovat helppo sulauttaa osaksi XP:tä. Kysymykseksi jää ovatko perinteiset suunnittelumenetelmät kelvollisia, vaikka ne rikkovat XP:n arvoja esimerkiksi panostamalla dokumentaatioon kommunikoinnin sijasta. Toinen pohdintaa herättävä asia on, että kuinka hyviä edellä mainitut, nimeenomaan XP:ta varten räätälöidyt, menetelmät ovat.

Kutsutaan XP:lle sovitettuja menetelmiä tästedes *ketteriksi arkkitehtuurisuunnittelun menetelmiksi*. XP:hen muualta tuotuja menetelmiä, joita ei ole alunperin luotu nimenomaan XP:lle sopiviksi, kutsutaan *lineaarisiksi arkkitehtuurisuunnitelun menetelmiksi*. Tämä luku jakautuu seuraavasti: ensiksi pohditaan lineaarisia menetelmiä yleisluontoisesti ja esitetään milloin niiden käyttäminen on perusteltua. Tämän jälkeen arvioidaan kehittäjätason tarinoiden, jatkuvan arkkitehtonisen refaktoroinnin ja arkkitehtuurikelpoistamisen hyviä ja huonoja puolia.

A. Lineaariset arkkitehtuurisuunnittelun menetelmät

Kehittäjät tarvitsevat tehokkaampia tapoja arkkitehtuurin suunnitteluun kuin mitä Extreme Programming luonnostaan tarjoaa. Vaihtoehtona on käyttää joko ketteriä tai lineaarisia arkkitehtuurisuunnittelun menetelmiä. Aiemmin mainittiin lineaaristen menetelmien haitoista: ne rikkovat ketterän kehityksen periaatteita ja XP:n arvoja. Menetelmissä on yleisluontoista, että arkkitehtuuri suunnitellaan kattavasti heti projektin alussa. Tuloksena syntyy usein paljon dokumentaatiota, jolloin kommunikointi vähenee, muutoksiin varautuminen huononee ja kehitysprosessi jäykistyy.

Etukäteen tehtävä arkkitehtuurisuunnittelu on toisinaan perusteltua. Jos toteutettava ohjelmisto on selkeä, asiakkaat tietävät tarkalleen mitä haluavat ja ohjelmiston kuvauksesta voidaan helposti tunnistaa komponentteja, jotka noudattavat tietynlaisia suunnittelumalleja, arkkitehtuuri voidaan suunnitella tyhjentävästi jo projektin alussa [2]. Tällöin arkkitehtuuri toimii ohjelmiston selkärankana, johon pystytään liittämään komponentteja, jotka noudattelevat näitä suunnittelumalleja. Tällaisessa tapauksessa kehitysaika lyhenee ja kehittäjien perehdyttäminen ohjelmointiin käy nopeasti. Suunnittelumallien käyttö on avuksi refaktoroinnissa, koska toteutettuja ominaisuuksia voidaan uudelleenkäyttää helpommin. Toisaalta tällainen lähestymistapa on hyvin projektikohtaista: kehitysprojekteissa lopullinen kuva ohjelmistosta on yleensä häilyvä, jolloin edellä mainittuja etuja ei voida saavuttaa eikä arkkitehtuurin täydellinen suunnittelu ole järkevää.

Martin Fowlerin mielestä alustava suunnittelu ei ole XP:ssä poissuljettu vaihtoehto [7]. Ohjelmiston komponenteille kannattaa tunnistaa erilaisia suunnittelumalleja, joita voidaan käyttää hyödyksi niiden toteutuksessa. Esimerkiksi on olemassa vakiintuneita ratkaisuja tietokannan kanssa

kommunikointiin, ohjelmiston kerrosarkkitehtuurin muodostamiseen ja niin edelleen. Kun kehittäjien tuntemus eri malleista kasvaa, heidän tulisi hyödyntää niitä. Kuitenkin XP:n periaatetta varautua muutoksiin ei tule unohtaa, vaan arkkitehtuuripäätökset tulisi pystyä peruuttamaan aina tarvittaessa. XP:n käytänteet, kuten yksinkertainen suunnittelu ja kattava testaus tukevat arkkitehtuurin muuttamista.

Arkkitehtuurisuunnittelun tulee tapahtua aikaisessa vaiheessa missä esimerkiksi vaatimusten määrittelykin. Tärkeää on, että ketterien menetelmien tapaan mitään ei pidetä varmana tai lopullisena, vaan muutoksiin ollaan varauduttu. Lineaariset suunnittelumenetelmät soveltuvat hyvin määritellyille projekteille, joissa asiakas tietää tarkalleen mitä haluaa ja vaatimukset ovat selkeät.

B. Ketterät arkkitehtuurisuunnittelun menetelmät

1) Kehittäjätason tarinat: Kehittäjätason tarinat edustavat ketterää arkkitehtuurisuunnittelua. Kaikki kehittäjät pääsevät yhdessä pohtimaan ohjelmiston arkkitehtuurin ongelmakohtia ja ratkaisemaan niitä. Tarinoilla kuvataan ohjelmistolle tehtäviä rakenteellisia muutoksia, jotka voivat olla pieniä tai suuria. Kun koko kehitysryhmä pääsee osallistumaan tarinoiden suunnitteluun ja arkkitehtuurista keskustelemiseen, kehittäjille muodostuu yhteinen näkemys ohjelmiston rakenteesta. Ymmärrys arkkitehtuurista helpottaa ohjelmiston komponenttien pitämistä yhtenevänä. Tarinoiden suunnittelussa asiakkaan osallistumista ei ole koettu tarpeelliseksi, mutta hänetkin pidetään prosessissa mukana kertomalla hänelle suunnitelluista rakenteellisista muutoksista.

Menetelmä kunnioittaa XP:n arvoja sillä tarinoiden suunnittelu vahvistaa kehittäjien välistä kommunikointia. Tarinoiden avulla ohjelmiston rakenne voidaan pitää yksinkertaisena. Arkkitehtuurin säännöllinen tarkastelu joka iteraation alussa voidaan nähdä palautteen saamisena, jossa kehittäjät arvioivat olivatko tehdyt muutokset ohjelmiston kannalta hyödyllisiä. Lisäksi tarinoiden teko rohkaisee kehittäjiä muuttamaan ohjelmiston arkkitehtuuria, jos se alkaa huonontua.

Arvojen noudattamisen lisäksi menetelmästä on hyötyä muille XP:n käytänteille ja vastaavasti XP:n käytänteet ovat hyödyksi kehittäjätason tarinoille. Menetelmä täydentää käyttäjätason tarinoita. Siinä missä käyttäjätason tarinat ottavat kantaa vain käyttäjille oleellisiin toiminnallisuuksiin, kehittäjätason tarinat huomioivat niitä ominaisuuksia, jotka näkyvät vain kehittäjille. Näistä esimerkkejä on monet laatutekijät, kuten ylläpidettävyys ja testattavuus. Pariohjelmointi parantaa kehittäjien osaamista ja ymmärrystä ohjelmiston kokonaiskuvasta [10]. Valmis ymmärrys ohjelmistosta on hyödyksi arkkitehtuurin ongelmakohdista keskustellessa ja kehittäjätason tarinoiden suunnittelu helpottuu. Vastaavasti menetelmän järjestämä keskustelu arkkitehtuurista nopeuttaa pariohjelmoinnissa tapahtuvaa oppimista ohjelmiston kokonaiskuvasta.

Menetelmä sopii hyvin XP:lle ja se parantaa kehitystyötä monilta osin. Se on kevyt ja lisää työn määrää vähäisesti. Kehitysprosessiin tulee vain yksi kokous lisää ja tarinat, jotka sulautuvat käyttäjätason tarinoiden joukkoon.

2) Jatkuva arkkitehtoninen refaktorointi: Jatkuvan arkkitehtonisen refaktoroinnin perusidea on selkeä: arkkitehtuurin ongelmia etsitään läpi iteraation ja kun ohjelmiston uusimpaan toimivaan versioon lisätään toteutettuja toiminnallisuuksia, kehittäjät luovat toteutuksista mallit. Mallien perusteella arkkitehti päivittää arkkitehtuurimallin vastaamaan ohjelmiston nykyversiota. Arkkitehtonisista ongelmista tehdään tehtäviä, joilla ongelmat korjataan tulevissa iteraatioissa. Menetelmä on kevyt erityisesti kehittäjien kannalta, koska heidän ei tarvitse huolehtia arkkitehtuurin kunnosta, vaan sen tekee yksittäinen arkkitehti.

Menetelmässä on kuitenkin ongelmia. Ohjelmistotuotannossa on olemassa mittari nimeltä *rekkaefekti* (truck factor) [11]. Mittarilla kuvataan niiden avainkehittäjien määrää, joiden lamaantuminen eli ”rekan alle jääminen” pysäyttää koko projektin. Projekti ei pysty jatkumaan, koska heillä on projektin etenemisen kannalta oleellista tietämystä, jota muilta kehittäjiltä ei löydy. Täydellinen arvo 0 tarkoittaa, että yksikään kehittäjä ei ole projektille elintärkeä. Tässä tapauksessa rekkaefektin arvo on pahin mahdollinen eli 1: jos arkkitehti syystä tai toisesta ei kykene jatkamaan projektia, ovat kehittäjät pulassa.

Arkkitehdin vastuu koko arkkitehtuurista voi olla muillakin tavoin haitaksi. Kehittäjien ei käytännössä tarvitse miettiä ohjelmiston arkkitehtuuria ollenkaan, vaan he toteuttavat vaatimukset parhaaksi katsomillaan tavoilla. Jos arkkitehtuuri menee huonoksi, arkkitehti luo kehittäjille uusia vaatimuksia, joilla arkkitehtuuria parannetaan. Tästä voi seurata paljon turhaa työtä. Kehittäjät eivät välttämättä opi virheistään, vaan jatkavat ohjelmiston kehittämistä tavoilla, joilla arkkitehtuuri huononee. Arkkitehti taas joutuu jatkuvasti tekemään korjaustoimenpiteitä pitääkseen ohjelmiston rakenteen hyvänä. Ongelma johtuu kommunikaation puutteesta: kehittäjät toimittavat malleja arkkitehdille ja arkkitehti toimittaa korjausvaatimuksia kehittäjille. Suoraa kommunikointia ei ole. Kun arkkitehtuuri ei ole koko kehitysryhmän vastuulla, sen pitäminen yhtenevänä on hankalaa. Tällaisenaan menetelmä ei ole XP:lle sopiva. Jatkuvasta arkkitehtonisesta refaktoroinnista tulee kelvollinen, kun siihen liitetään arkkitehtuurikelpoistaminen menetelmän jatkoksi.

3) Arkkitehtuurikelpoistaminen: Arkkitehtuurikelpoistamisessa kehittäjät ja asiakas ottavat yhdessä vastuun ohjelmiston arkkitehtuurista. Käyttäjälle oleellisia laatutekijöitä tarkastellaan ja keksitään ratkaisuja, joilla niiden laatu pysyy hyvänä. Arkkitehtuurikelpoistaminen on suoraa jatkoa jatkuvalle arkkitehtoniselle refaktoroinnille.

Menetelmä korjaa jatkuvan arkkitehtonisen refaktoroinnin puutteet tuomalla kehittäjät ja asiakkaat mukaan arkkitehtuurin suunnitteluun. Ratkaisu ei ole täysin ongelmaton, koska suunnittelutyö ja vastuut ovat jaettu kahteen osaan: arkkitehdin tehtävä on huolehtia laatutekijöistä, jotka helpottavat toteutustyötä eli ovat siten kehittäjille tärkeitä. Arkkitehtuurikelpoistamisen järjestämässä kokouksessa kehittäjät ja asiakas taas ovat vastuussa laatutekijöistä, jotka ovat käyttäjille tärkeitä ja näkyviä. Menetelmän mukaan kehittäjillä on oikeus kuulla lyhyet perustelut arkkitehdin suunnittelemille muutoksille ja ilmaista mielipiteensä niistä, mutta lopullinen päätösvastuu on edelleen arkkitehdilla.

Laatutekijöihin liittyvien vastuiden jakaminen saattaa ai-

heuttaa ongelmia ohjelmiston suunnittelussa, koska laatutekijät voivat olla ristiriidassa keskenään. Kokouksessa arkkitehdilla ja kehittäjillä on mahdollisuus ratkaista nämä ristiriidat. Ongelmana on, että arkkitehti suunnittelee muutoksia koko iteraation ajan. Kehittäjät pääsevät ilmaisemaan mielipiteensä vasta kokouksessa, joka on iteraation lopussa. Tästä voi seurata konflikteja, koska kehittäjät pääsevät vaikuttamaan asioihin vasta, kun arkkitehti on ehtinyt tehdä omat päätöksensä. Yksi kokous ei välttämättä riitä ratkaisemaan ristiriitoja, joita arkkitehdin ja kehittäjien suunnittelemat arkkitehtuurimuutokset aiheuttavat.

Jos laatutekijät ovat valittu hyvin ja kommunikointia pystytään tehostamaan arkkitehdin ja kehittäjien välillä, menetelmä toimii. Kaikki oleelliset henkilöt pääsevät keskustelemaan arkkitehtuurista ja osallistumaan sen kehittämiseen. Jaettu näkemys arkkitehtuurista helpottaa kehitystyötä. Huomionarvoista on, että arkkitehtuurikelpoistaminen menetelmänä vaatii arkkitehdin roolin. Menetelmää voi käyttää yhdessä jonkin toisen menetelmän kanssa, joka ohjeistaa arkkitehdin tehtävistä, mutta se toimii myös itsenäisesti.

Menetelmistä voi huomata yhtenäisiä piirteitä. Ne pyrkivät noudattamaan ketteriä periaatteita ja mahdollistamaan arkkitehtuurisuunnittelun tehokkaasti sekä kevyesti. Arkkitehtuurin suunnitteluun tulisi osallistua siihen vaikuttavat henkilöt eli kehittäjät ja asiakas. Jatkuva arkkitehtoninen suunnittelu ja arkkitehtuurikelpoistaminen jakoivat arkkitehtuuriin liittyviä vastuita, joista on mahdollisesti harmia. Sen sijaan kehittäjätason tarinoissa kaikki kehittäjät olivat vastuussa arkkitehtuurin kaikista piirteistä. Näistä kolmesta menetelmästä juuri kehittäjätason tarinat vaikuttaa sopivimmalta XP:lle.

V. Yhteenveto

Arkkitehtuurisuunnittelu on tärkeä osa ohjelmistokehitystä. XP:ssä arkkitehtuuria kehitetään yksinkertaisen suunnittelun, metaforan ja refaktoroinnin kautta. Kuitenkin edellä mainitut käytänteet eivät riitä pitämään arkkitehtuuria hyvänä koko kehitystyön ajan etenkin laajemmissa projekteissa.

Käytännössä isompiin projekteihin otetaan mukaan menetelmiä, joilla voidaan keskittyä arkkitehtuurisuunnitteluun paremmin. Menetelmät ovat toisinaan lainattu lineaarisista kehitysmalleista, joiden toteutustavat ja ohjeistukset ovat ristiriidassa XP:n arvojen ja muiden käytänteiden kanssa. Yleensä nämä menetelmät keskittyvät etukäteen tehtävään suunnitteluun, jolla pyritään lukitsemaan ohjelmiston arkkitehtuuri heti alussa tietynlaiseksi. Lopputuloksena kehittäjät joutuvat käymään läpi raskaita suunnitteluprosesseja, joissa tuotettavan dokumentaation määrä kasvaa, kommunikointi vähenee ja muutoksiin on vaikeampi varautua.

Arkkitehtuurisuunnittelu on XP:ssä monilta osin puutteellinen. Lineaariset suunnittelumenetelmät taas ovat haitaksi ketterälle kehitykselle. Ongelmaa on ratkaistu kehittämällä menetelmiä, joilla arkkitehtuuria voidaan suunnitella ja kehittää XP:ssä ketterästi ja tehokkaasti. Näitä menetelmiä ovat muun muassa kehittäjätason tarinat, jatkuva arkkitehtoninen refaktorointi ja arkkitehtuurikelpoistaminen. Menetelmät pyrkivät noudattamaan ketteriä periaatteita sekä XP:n arvoja ja käytänteitä. Menetelmistä voidaan havaita, että yleisesti ottaen ne koettavat pitää arkkitehtuurisuunnittelun kevyenä ja iteratiivisena. Kaikkien kehittäjien ja asiakkaan tulisi jakaa yhteinen vastuu arkkitehtuurista ja etenkin oppia hahmottamaan kokonaiskuva ohjelmiston rakenteesta, jotta toteutetut komponentit pysyvät yhtenevinä.

Kaikki menetelmät eivät kuitenkaan saavuta täydellisesti näitä tavoitteita. XP on suosittu ketterä menetelmä, mutta sen puutteet arkkitehtuurisuunnittelun suhteen kaipaavat edelleen vakiintuneita ratkaisuja. XP:lle soveltuvia ketteriä arkkitehtuurin suunnittelumenetelmiä tulisi tutkia käytännön kokeiluissa enemmän nähdäksemme niiden todellisen hyödyn. Ohjelmiston arkkitehtuuri on kuitenkin kehitystyön yksi oleellisimmista asioista.

References

[1] A. Sharifloo, A. Saffarian, and F. Shams, "Embedding architectural practices into extreme programming," in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, March 2008, pp. 310–319.

[2] L. Cao, K. Mohan, P. Xu, and B. Ramesh, "How extreme does extreme programming have to be? adapting xp practices to large-scale projects," in *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 3 - Volume 3*, ser. HICSS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 30 083.3–. [Online]. Available: http://dl.acm.org/citation.cfm?id=962751.962914

[3] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[4] P. Deemer and G. Benefield, "The scrum primer. an introduction to agile project management with scrum," 2007. [Online]. Available: http://www.rallydev.com/documents/scrumprimer.pdf

[5] B. Boehm, "Get ready for agile methods, with care," *Computer*, vol. 35, no. 1, pp. 64–69, Jan. 2002. [Online]. Available: http://dx.doi.org/10.1109/2.976920

[6] J. Rasmussen, "Introducing xp into greenfield projects: lessons learned," *Software, IEEE*, vol. 20, no. 3, pp. 21–28, May 2003.

[7] M. Fowler, "Extreme programming examined," G. Succi and M. Marchesi, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, ch. Is Design Dead?, pp. 3–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=377517.377518

[8] R. N. Jensen, T. Møller, P. Sönder, and G. Tjørnehøj, "Architecture and design in extreme programming; introducing "developer stories"," in *Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering*, ser. XP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 133–142.

[9] R. L. Nord, J. E. Tomayko, and R. Wojcik, *Integrating Software-Architecture-Centric Methods into Extreme Programming (XP)*, September 2004. [Online]. Available: http://repository.cmu.edu/cgi/viewcontent.cgi?article=1494&context=sei

[10] A. Begel and N. Nagappan, "Pair programming: what's in it for me?" in *Proceedings of the Second ACM-IEEE international symposium on Empirical software*

engineering and measurement, ser. ESEM '08. New York, NY, USA: ACM, 2008, pp. 120–128. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414026

[11] G. Succi, M. Marchesi, L. Williams, and J. D. Wells, *Extreme Programming Perspectives*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.