LAPPEENRANNAN TEKNILLINEN YLIOPISTO

SÄHKÖTEKNIIKKA

DIPLOMITYÖ

Ketterien menetelmien hyödyntäminen sulautettujen järjestelmien kehityksessä.

ESKO KASILA

KÄRPÄNPOLKU 7

36200 KANGASALA

ESKO.KASILA@LUT.FI

PUH. 050 4869800

TIIVISTELMÄ

Lappeenrannan teknillinen ylipisto
Sähkötekniikan osasto

Esko Kasila

Ketterien menetelmien hyödyntäminen sulautettujen järjestelmien kehityksessä

DIPLOMITYÖ

2013

52 SIVUA, 5 KUVAA, 3 TAULUKKOA

TARKASTAJAT: JERO AHOLA, ANTTI KOSONEN

HAKUSANAT: SULAUTETUT JÄRJESTELMÄT, KETTERÄT MENETELMÄT, OHJELMISTOTUOTANNON MENETELMÄT

KEYWORDS: EMBEDDED SYSTEMS, AGILE METHODS, SOFTWARE ENGINEERING

Sulautettujen järjestelmien projekti voidaan toteuttaa monella tavalla. Projektiin liittyy aina ohjelmiston, sekä laitteiston kehittäminen. Ohjelmiston suunnittelulla on suuri painoarvo ja tämä näkyy erityisesti varsinkin kulutuselektroniikassa. Kannettavien laitteiden räjähdysmäisesti lisääntynyt myynti ja käyttö ovat tuoneet markkinoille lisää rahaa ja mielenkiintoa. Tästä johtuen markkinoille tulee joka vuosi entistä kehittyneempiä laitteita. Laitteiston kehittymisen sekä asiakkaiden vaatimusten lisääntyessä ohjelmistojen koko on kasvanut. Tämä on luonut tarpeen myös sulautettujen järjestelmien projekteille ottaa käyttöön jokin tietty metodi ohjelmistojen tuotannossa. Ongelmana on kuitenkin se, että sulautettujen järjestelmien projekteihin on sovellettu metodeita, joita ei ole alun perin suunniteltu laitteiston ja ohjelmiston yhteissuunnitteluun ja toteuttamiseen. Miten voidaan valita oikea metodi sulautettujen järjestelmien projektiin? Tässä työssä esitellään perinteisiä ohjelmistotuotannon metodeita, sekä keskitytään eri ketterien metodien tutkimiseen. Tämä työ selvittää mikä vaikuttaa metodin valintaan sulautetun järjestelmän projektille. Tässä tutkimuksessa päädytään siihen johtopäätökseen, että sulautetuin järjestelmän suunnittelussa ja toteutuksessa ketterien menetelmien käyttö parantaa projektin mahdollisuutta onnistua täyttämään asiakkaan vaatimukset. Ketterien menetelmien käyttö ei poista tarvetta kehittää menetelmää, joka lähtökohtaisesti ottaa huomioon laitteiston ja ohjelmiston yhteissuunnittelun.

ABSTRACT

Lappeenranta University of Technology

Department of Electrical Engineering

Esko Kasila

Using agile methods in engineering in embedded systems

2013

52 PAGES 5 FIGURES, 3 TABLES

SUPERVISORS: JERO AHOLA, ANTTI KOSONEN

KEYWORDS: EMBEDDED SYSTEMS, AGILE METHODS, SOFTWARE ENGINEERING

Methods creating embedded systems vary a lot. Embedded systems always contain software and hardware and their deep interaction. As more and more consumer electronics have embedded intelligence in them, the shift from hardware is moving the focus to develop better software in embedded systems. As the markets grow and sales increase there is more money and interest to create even more fantastic products. This has created the growing demand for better methods in software and hardware design and development. Currently methods used to create embedded systems have been used in software production. How to select the right method for right embedded system project is difficult. Latest additions to software development are agile methods. Traditional and agile software engineering methods are introduced in this paper. The purpose of this paper is to find out what methods are out there and how to choose the right method that fits best for different kind of embedded system projects. The conclusion found in this paper is that the agile provides higher chance of success per embedded system project, but the co-design method for embedded system creation is needed.

LYHENTEET

DSDM	Dynamic Systems Development Method
XP	Extreme Programming
TOMI	Toiminnallisen mallinnuksen iteraatio
KAI	Käyttöönotto iteraatio
SUTI	Suunnittelun ja toteutuksen iteraatio
LIMA	Liiketoiminnan määrittely

TAULUKOT

KUVAT

SISÄLLYSLUETTELO

1 Johdanto

1.1 Taustaa

Sulautettu järjestelmä tarkoittaa sellaista järjestelmää, joka koostuu laitteistosta ja sitä ohjaavasta ohjelmistosta. Ohjelmiston ja laitteiston summana on laite, joka toteuttaa sille suunniteltua tehtävää. Sulautettu järjestelmä on usein tehty niin, että se toimii itsenäisesti, eli käyttäjän ei tarvitse olla tietoinen kuinka laite toimii, kun se on käytössä. Sulautettu järjestelmä voi siis olla yhdellä led-näytöllä varustettu sydämientahdistaja tai älypuhelin matkapuhelin. Sulautettujen järjestelmien teollinen tuotanto ja erityisesti niiden kehittäminen vaatii erilaisia ohjelmisto- ja laitteistotuotannon menetelmiä. Laitteisto onkin joskus suunniteltu ja toteutettu kokonaan ennen kuin ohjelmistoa on alettu toteuttaa. (Luciano, Sangiovanni-Vincentelli & Sentovich, 1999). Ohjelmistotuotannon menetelmiä on perinteisesti käytetty puhtaaseen ohjelmiston tuottamiseen.

Ohjelmistotuotannon eri menetelmiä on tarvittu poistamaan erilaisia ohjelmistotuotannon ongelma-alueita, kuten projektin aikataulun venymistä, virheitä, monimutkaisten ohjelmistojen rakentamista ja henkilöstön loppuun palamista. Nämä menetelmät vaihtelevat vesiputousmallin yksinkertaisesta prosessista, aina ketterien menetelmien päivärytmin sanelemaan henkilöiden roolitukiseen. Näistä erilaisista menetelmistä voidaan ohjelmistotuotannossa valita aina kaikista tehokkain tapa saada haluttu lopputulos. Sulautetuissa järjestelmissä voidaan toimia samalla tavalla, mutta ottaen huomioon sen seikan, että ohjelmisto on riippuvainen fyysisestä toimintaympäristöstä. Tavallisten kotitietokoneiden ohjelmistojen joustavuus ja hyvyys perustuvat juuri sille, että ne ovat riippumattomia fyysisestä laitteistosta (Henzinger & Sifakis 2006) Sulautetun järjestelmän, kuten nykyaikaisen älypuhelimen on otettava huomioon ohjelmallisesti esimerkiksi miten päin laite on kädessä.

1.2 Työn tavoitteet

Työssä on tarkoituksena etsiä sulautettujen järjestelmien erityispiirteitä, sekä esitellä ja

vertailla erilaisia ohjelmistotuotannon malleja. Sulautettujen järjestelmien erityispiirteet ja niiden vaikutukset ohjelmistotuotantoon ovat tutkimuksessa keskeisesti mukana. Tässä työssä yleisen katsauksen lisäksi keskitytään ketterien menetelmien soveltamiseen osana sulautettujen järjestelmien projektia. Työn lopputuloksena saadaan analyysi siitä, miten yleisimmät metodit toimivat, sekä tarkempi kuvaus ketterien menetelmien käytöstä sulautetuissa järjestelmissä. Työ tehdään tutkimalla kirjallisuutta, sekä yhdistelemällä kokemusperäistä tietoa Nokia Oy:ssä tekemäni työni perusteella.

1.3 Työn rajaus

Tässä diplomityössä ei toteuteta mitään ohjelmistoprojektia, vaan esitellään erilaisia ohjelmistotuotannon menetelmiä ja niiden soveltuvuutta sulautettujen järjestelmien projektiin. Itselläni on kokemusta ketterästä metodista nimeltä Scrum ja tätä osaamista hyödynnän tässä työssä.

1.4 Työn rakenne

Tässä työssä esitellään ensin mitä tarkoittaa sulautettu järjestelmä. Tämän jälkeen esitetään yksityiskohtaisesti mitä eri osia sulautetun järjestelmän laitteistoon kuuluu. Seuraavaksi esitellään sulautetun järjestelmän ohjelmistolliset osat, eli eri ohjelmistotasot. Lopuksi kappaleessa kaksi tehdään yhteenveto sulautetun järjestelmän erityispiirteistä. Kolmannessa kappaleessa esitellään ensin suunnitelman keskeiset menetelmät, joita käytetään ohjelmistotuotannossa. Kappaleen tarkoituksena on antaa yleiskuva kaikista yleisesti käytössä olevista ohjelmiston tuottamiseen käytetyistä menetelmistä. Tämän jälkeen esitellään ketterät menetelmät.

Työn varsinainen sisältö ja kiinnostava tutkimus on näiden eri menetelmien soveltaminen sulautetun järjestelmän projektissa kappaleessa kaksi esiteltyjen erityisvaatimusten osalta. Lopuksi esitetään johtopäätökset ja yhteenveto eri menetelmistä.

1.5 Tutkimusalue

Tässä työssä esitellään eri ohjelmistotuotannon menetelmiä, joita voidaan käyttää

projektin eri vaiheissa. Tutkittavia menetelmiä ovat “plan driven”-toteutus, evoluutiomallit sekä ketterä-malli. Näistä tarkemmin tutkitaan ketterää mallia.. Lopputuloksena saadaan eri ohjelmistotuotannon menetelmien vertailu, jossa on merkittynä hyvät ja huonot puolet eri menetelmien käytöstä sulautettujen järjestelmien projektissa. Ketterien menetelmien osalta tuloksena on yksityiskohtaisempi tulos siitä miten erilaisten ketterien menetelmien käyttö vaikuttaa projektin onnistumiseen.

2 Sulautettu järjestelmä

Sulautetut järjestelmät sisältävät jonkin ohjelmiston ja laitteiston yhdistelmän, joka suorittaa tiettyä ennalta määrättyä tehtävää. Laitteistot vaihtelevat massatuotetusta kulutuselektroniikasta aina avaruusluotaimiin. Sulautettu järjestelmä poikkeaa normaalista tietokoneesta, koska sulautetulla järjestelmällä on tietokoneeseen verrattuna tarkempi tehtävä. Lisäksi järjestelmän tulee toimia autonomisesti ilman yhtäjaksoista valvontaa tai ulkopuolista ohjausta. Sillä on lisäksi oltava tapa saada ulkoisia signaaleja. (Luukko 2002) Jotta kaikki edellä mainitut asiat olisivat mahdollisia, on järjestelmän suoriuduttava sille asetetuista vaatimuksista nopeuden, luotettavuuden, sekä vasteen osalta. Laitteen ollessa kannettava, on sen myös pystyttävä toimimaan erittäin energiatehokkaasti. Koodin tulee olla tehokasta, virheetöntä ja tarkoituksen mukaista. Tämä tarkoittaa sitä, että jos käytettäisiin esimerkiksi Microsoftin erilaisia kaupallisia ohjelmistoympäristöjä projektissa, niin näillä on usein tapana varautua laajennettavuuteen ja näin varata resursseja tulevaisuuden varalle. Pahimmassa tapauksessa koodaaja ei todellisuudessa tiedä miten koodi toimii. Tämä vaatimus on usein syy sille, ettei yleistä ratkaisua voida tehdä. Esimerkiksi avaruuteen ei voida lähettää laitetta, joka on suunniteltu maanpinnalla toimivaksi. Näin jäljelle jää vain vaihtoehto suunnitella juuri sellainen sulautettu järjestelmä, joka voi toimia vaadituissa olosuhteissa. Edellä mainitusta kuvauksesta huolimatta voi olla joskus vaikeaa tunnistaa sulautettua järjestelmää ja siksi seuraavaksi esitellään yksityiskohtainen esimerkki sulautetusta järjestelmästä.

2.1 Esimerkki sulautetusta järjestelmästä

Esimerkkinä sulautetusta järjestelmästä käytetään kannettavaa CD-soitinta. Ennen CD-soitinta käytettiin kasettisoitinta ja LP soitinta. Vertaamalla CD-soitinta analogiseen LP-soittimeen, voidaan helposti havainnollistaa sulautettujen järjestelmien ydin. LP- ja CD-soitin toteuttavat saman tavoitteen toiminnon eli toistavat levyltä halutun kappaleen.

Kannettavan CD-soittimen suunnittelun lähtökohdat ovat sulautetulle järjestelmälle tyypilliset. Laite tulee olla kestävä, sillä sitä pidetään repussa tai laukussa. Se on kannettava eli se ei voi luottaa verkkovirtaan toimiakseen. Lisäksi siinä tulee olla ”älyä” virheenkorjaukseen, kappaleen valintaan, äänenvoimakkuuden hallintaan ja muihin mahdollisiin toimintoihin. Lisäksi laitteisto on kuluttajille tarkoitettu, eli

massatuotettava ja hinnaltaan kohtuullinen. Mielenkiintoiseksi tämän esimerkin tekee erityisesti se, että käyttäjä voi periaatteessa käyttää CD-soitinta kuten LP-soitinta. Käyttäjätapaukset ovat LP- ja CD-soittimen käyttäjille melkein identtiset: Käyttäjä ottaa levyn (joko LP tai CD levy) ja laittaa sen soittimeen. Käynnistää laitteen ja valitsee kappaleen. Neulan fyysinen siirtely on ainoa merkittävä ero CD- ja LP-soittimien välillä käyttäjän kannalta.

Laitteen toiminnan kuvaus paljastaa CD-soittimen sulautetun luonteen. CD-levyn asettamisen jälkeen käyttäjä sulkee kannen. CD-soitin etsii ensimmäisen kappaleen levyltä. CD:n sisältö on täysin digitaalista. Perinteisesti CD:n sisältö on 16 bittistä 44,1 kHz näytteenottotaajuudella. (wikipedia, CD-Levy) Tätä sisältöä luetaan ja käsitellään laitteen prosessorissa, joka muutetaan audiosignaaliksi ja lopulta käyttäjän kuunneltavaksi. CD-soittimen hallinta tapahtuu yleensä painikkeilla laitteen kyljessä. Perustoimintoja ovat kappaleen vaihtaminen, pikakelaus, soiton aloittaminen ja lopettaminen. Kaikki nämä toiminnot tapahtuvat prosessorin ohjauksessa. Kun käyttäjä käyttää play-painiketta, CD-soitin aloittaa levyn pyörittämisen ja etsii ensimmäisen kappaleen. Se lukee raitaa ja hakee dataa levyltä. Tässä vaiheessa usein suoritetaan myös virheenkorjausta ja puskurointia. Jos laite tärähtää ja ”neula” eli lukupää, hukkaa kohdan, jossa se oli menossa, on laitteessa puskuri, johon dataa ladataan etukäteen esimerkiksi 5 sekunniksi. Tässä ajassa ”neula” löytää kohdan, jossa se oli menossa ennen häiriötä. Kaikki tämä tapahtuu ilman käyttäjän apua ja tätä sanotaan tavallisesti laitteen älyksi. Sulautettujen järjestelmien yleistymisestä käytetäänkin kansanomaista termiä: “lisätä älyä laitteisiin”. CD-soittimesta voidaan lisäksi tunnistaa helposti sulautetun järjestelmän eri perusosat. CD-soitin tarvitsee signaalin käsittelyyn tehokkaan prosessorin, sekä muistia puskurille ja ohjelmistolleen. Lisäksi se tarvitsee energiaa akusta ja toimivan fyysisen koteloinnin. Laitteessa pitää lisäksi olla näyttö ja käyttöliittymä kuluttajan tarpeisiin. Lopullinen signaali esivahvistetaan ja toimitetaan korviin mahdollisilla kaiuttimilla.

CD-soitimen teollinen tuotanto ja kehittäminen vaativat kolmen eri osa-alueen tuntemusta: Ohjelmiston rakentamista, laitteiston tuntemista sekä syntyvän tuotteen käyttäjäkunnan tuntemista. Yhdenkin näistä puuttuminen sulautetussa järjestelmässä vaarantaa lopullisen tuotteen onnistumisen. CD-soittimen ohjelmiston rakentaminen ei ole kuitenkaan erillinen asia laitteiston suunnittelusta. Näitä pitää pystyä rakentamaan ja

suunnittelemaan yhtä aikaa, jotta ne olisivat enemmän kuin osiensa summa. (Henzinger & Sifakis 2006). Seuraavassa kuvataan sulauteltujen järjestelmien eri osat tarkemmin.

2.1.1 Prosessori

Prosessori on laitteen laskennan suorittava yksikkö. Prosessorit voivat erota toisistaan huomattavasti. Prosessori ajaa ohjelmaa, joka on järjestelmän ohjelmamuistissa. Se lisäksi ottaa sisään halutut syötteet ja toteuttaa niiden perusteella ohjelmiston määräämät toiminnot. Se käyttää työmuistia, sekä mahdollista tallennustilaa. Prosessori on koko laitteen ydin ja sen mitoittamisessa tehdään ratkaisuja, jotka vaikuttavat kaikkiin muihin laitteisiin. Käyttötarkoitus sanelee prosessorin laskentatehon tarpeen. Kun oikea prosessoriarkkitehtuuri on valittu projektiin, sen vaihtaminen on myöhemmin vaikeaa. Prosessorin arkkitehtuuri vaikuttaa kaikkiin laitteistoa lähellä oleviin rajapintoihin, sekä muihin laitteisiin ja liitännäisiin. Prosessorin tilalla voi olla mikrokontrolleri tai jokin puhtaasti signaalinkäsittelyyn tarkoitettu komponentti tarpeen mukaan. Prosessorin ollessa koko laitteen ydin, on sen ehdoton vaatimus luotettavuus. Laitteistojen ollessa usein kannettavia on huomioitava laitteen iskunkestävyys.

2.1.2 Muisti

Sulautettujen järjestelmien äly tulee ohjelmasta ja sitä käyttävästä prosessorista tai mikrokontrolleista. Tämän mahdollistavat erilaiset muistit, joita on vaihteleva määrä riippuen laitteen tehtävästä. Muistin määrä ja sen toteuttamisen arkkitehtuuri määrittävät laitteen tehonkulutusta, vasteaikoja ja tuotteen lopullista hintaa. Tyypillisesti sulautetussa järjestelmässä käytetään muistiarkkitehtuuria, joka jaetaan kahteen osaan, jotka ovat luku ja työmuisti. (Park, Seo, Seo, Kim & Kim, 2003)

Lukumuistia voidaan vain lukea normaalimenettelyin sulautetun järjestelmän käytön aikana. Erillistä käyttömuistia tai työmuistia voidaan lukea ja kirjoittaa laitteen ollessa päällä. Lukumuistia käytetään tallentamaan ajurit ja ohjelmat, jotka mahdollistavat laitteen käynnistämisen ja joskus koko käyttöjärjestelmä on tallennettu sinne. Nämä muistit voivat olla myös prosessorissa integroituina, sekä niitä voi olla useita. Näitä ovat esimerkiksi väylien lukemiseen käytettävät puskurimuistit, käskyliukuhihnamuistit ja prosessorin sisäiset välimuistit. Liukuhihnaa ja prosessorin omaa välimuistia käytetään tehostamaan prosessorin toimintaa niin, että välimuisti tallentaa ajossa olevia ohjelmia

nopeasti pieneen muistialueeseen, jotta ohjelman saa taas nopeasti ajoon. Liukuhihnaa käytetään, jotta jokaisella prosessorin kellojaksolla voidaan ajaa tehokkaasti ohjelmia.

Sulautetuissa järjestelmissä tarvitaan usein myös massamuistia. Tällä tarkoitetaan muistia, johon tallennetaan suuria määriä tietoa, jota ei tarvita esimerkiksi käyttöjärjestelmän ajamiseen, kuten esimerkiksi tietokoneen kovalevy tai valokuvien sijaintipaikka kamerassa. Tämä muistialue saattaa sisältää osia käyttöjärjestelmästä, mutta silti se ladataan aina ensin käyttömuistiin ennen ajoa. Sulautettujen järjestelmien ollessa yhä useammin kannettavia ja monipuolisia laitteita, ovat muistien vaatimukset kasvaneet viime vuosina. Nykyään kaikkien muistityyppien pitää olla nopeita sekä suunniteltavasta kokonaisuudesta riippuen laajennettavia. Flash-muistien ansiosta, jopa lukumuistilla olevat ajurit ja käyttöjärjestelmä ovat päivitettävissä.

Kaikkiin muistityyppeihin ja muistiarkkitehtuureihin liittyy aina energiankulutus. Se kuinka nopea muisti on, lisää suorassa suhteessa muistin energiankulutusta. Lisäksi erilaiset algoritmit nopeasti yleistyneen Flash-muistin lukemisen, kirjoittamisen ja kulumisen optimoimisiksi vaikuttavat muistin käytöstä johtuvien kustannusten ja energiankäytön lisääntymiseen. (Shiue & Chakrabarti, 1999)

2.1.3 Kotelo ja liittimet

Katsottaessa sulautettua järjestelmää, käyttäjä näkee ensimmäisenä koteloinnin. Laitteiston kotelointiin on omat vaatimukset. Koteloinnin tarkoituksena on suojata sisällä olevia herkempiä laitteita esimerkiksi mekaaniselta rasitukselta tai ympäristön epäpuhtauksilta. Kotelon tulee suojata laitetta ulkopuolisilta elektromagneettisilta häiriöiltä. Lisäksi kotelo suojaa ympäristöä laitteiston itsensä tuottamalta häiriöltä. Laitteiston koteloinnin heikoin kohta ovat usein liitokset ja liittimet. Näitä ovat esimerkiksi erilaiset verkko- tai virtaliittimet. Käytännössä melkein kaikissa sulautetuissa järjestelmissä on oltava ainakin yksi sisään menevä fyysinen liitos. Tämä mahdollistaa laitteen lataamisen tai muun käyttövirran laitteelle. Nämä laitteiston fyysiset rajapinnat ovat lian ja sähkömagneettisen häiriön lisäksi alttiita mekaanisille rasituksille. Laitteen kytkeminen laturiin aiheuttaa suoraa painetta joko piirilevyyn tai kotelon kiinnityksiin. Erillisten liittimien käyttäminen rasittaa laitteistoa ja näihin on varauduttava jo suunnittelussa. Esimerkiksi Nokian kännyköissä ja Applen MacBook Pro kannettavissa tietokoneissa on juuri ulkoisten liittimien mitoitus ratkaissut sen,

miten ohut laite voi olla. Kun suunnitellaan kannettavaa laitetta, esimerkiksi kännykkää, on sen koteloinnin suojattava sitä myös siinä tapauksessa, jos laite putoaa. Laitteen sisälle koteloinnin ja piirilevyn väliin voidaan laittaa vaimentavaa materiaalia, mikä vähentää kiihtyvyyden vaikutusta itse laitteiston herkempiin sisärakenteisiin. Laitteiston koteloinnin on myös vastattava sähköturvallisuuden vaatimuksia kaikilta osin. Laite ei saa aiheuttaa vaaratilannetta edes väärin käytettynä.

2.1.4 Anturit ja näytöt

Sulautetulla järjestelmällä ei tarvitse olla välttämättä mitään näyttöä. Usein jo pelkästään laitteen kehittämisen kannalta on järkevää tehdä sellainen. Se voi olla esimerkiksi led valo laitteen kyljessä tai ääni. Laitteessa pitää olla jokin ulostulo sisään menevän kanavan lisäksi.

Kehittyneempien sulautettujen järjestelmien, kuten kännykän kosketusnäytöllä on monimutkaisempia vaatimuksia. Näiden lisäksi erilaiset värinät ja audiosignaalit toimivat monitoreina laitteen hallintaan ja laitteen keinona kommunikoida käyttäjän kanssa. Kesällä 2008 laskin silloisesta Nokian uutuuskännykästä kaikki tulot ja lähdöt, mitä teknisestä dokumentista löytyi. Erilaisia liittymiä olivat seuraavat: Infrapunalinkki, GSM, WLAN, näyttö, kaksi näppäimistöä, GPS, latausliitin, usb-liitin, Nokian oma standardiliitin, akun liitos, näppäimistön kääntämisen havaitseva kytkin, asentoanturi, kamera, taustavaloanturi, läheisyysanturi, mikrofoni, kaiutin, värinä ja ledi äänettömänä hälyttimenä. Sulautettujen järjestelmien toteuttamisessa nämä kaikki erilaiset rajapinnat on hallittava. Tämä asettaa vaatimuksia koko laitteen arkkitehtuurille, koska voidaan joutua luomaan yhteinen tapa hoitaa eri liitokset, varsinkin kun niitä paljon.

Laitteiston anturit tai muut syöttökanavat tulee olla luotettavia, näin sisällä oleva ohjelmisto saa oikeaa dataa. Ei siis riitä, että ohjelmisto on luotettava, jos anturit menevät rikki. (Barr, M. 2012) Anturien suunnittelussa on myös varmistettava, että siirtotie on tarpeeksi nopea. Tällä tarkoitetaan itse johdinta laitteeseen, liitintä sekä väylää ja väylän puskureita. Nämä asettavat aina vaatimuksia yhteensopivuuksille eri anturien, siirtoteiden ja prosessorien välille.

2.1.5 Energian kulutus

Energian kulutuksen vaatimus riippuu projektista mihin laitetta suunnitellaan. Tämä voi vaihdella lasten lelusta aina USA:n käyttämiin miehittämättömiin lennokkeihin. Mitä

suurempi rooli on energian säästämisellä, sitä suurempi vaikutus sillä on muun laitteiston suunnitteluun ja ohjelmistoon. Yleisesti laitteen suunnittelussa tulee mitoituksen vastata tarvetta, joka taas tarkoittaa sitä, että aina yritetään alentaa energian kulutusta mahdollisimman paljon. Esimerkiksi Nokialla tämä tarkoitti sitä, että uuden puhelimen tullessa moduulitestaukseen ensimmäistä kertaa, se kulutti enemmän tehoa kuin lopullinen tuote. Kun ohjelmisto oli valmis, sitä jouduttiin optimoimaan vähemmän energiaa kuluttavaksi. Kun laitteisto vastasi ohjelmiston vaatimuksia, työ onnistui hyvin nopeasti. Akun mitoituksen ollessa liian pieni, ei puhelinta saatu toimimaan tarpeeksi pitkään.

Energian käytön mitoitus on erittäin haastavaa, kun kyseessä on reaaliaikakäyttöjärjestelmä. Käyttäjä voi käynnistää useita eri ohjelmia ja muodostaa näin prosessoria käyttävän kombinaation ja kuluttaa akun tyhjäksi. Näitä eri mahdollisia kombinaatioita voi olla lukemattomia ja niitä kaikkia ei voi ennakoida etukäteen. Tämä on usein suurimpia rajoittavia tekijöitä, jos kyseessä on kannettava laite. Tähän on keksitty erilaisia menetelmiä. Yksi tapa on luoda laitteistosta ohjelmistollinen vaste, jotta koko laitteen käyttöä voidaan simuloida. Eräs tällainen tapa on toteuttaa ohjelmiston ja laitteiston yhteiskehitys SystemC/C++:lla, joka mahdollistaa PKtool:n käytön. (Springer, 2011) Erilaisten emulaattoreiden käyttö yleistyy kokoajan akun kestoajan pidentämiseksi, koska kannettavat laitteet ovat yleistyneet räjähdysmäisesti.

2.2 Yhteenveto sulautettujen järjestelmien laitteiston osien vaatimuksista

Sulautettujen järjestelmien vaatimukset riippuvat suoraan siitä, mihin suunniteltavaa laitetta aiotaan käyttää. Voidaan kuitenkin todeta, että ohjelmiston toiminnan lisäksi laitteisto pitää suunnitella niin, että siinä on tarpeeksi suorituskykyä ja luotettavuutta sopivaan hintaan. (Wolf 1994) Yhteenvetona voidaan esittää seuraava taulukko 1. Tässä taulukossa on esitetty yleiset vaatimukset sulautetun järjestelmän eri osille.

Taulukko 1. Laitteiston vaatimuksia

Laitteiston osa:	Vaatimus
Prosessori/mikrokontrolleri	Luotettavuus, reagointinopeus, laskentateho, energiatehokkuus, häiriösieto
Kotelo ja liittimet	Kestävyys, EMC, mekaanisen rasituksen kesto.
Tallennusmuisti	Luotettavuus, energiantehokkuus, nopeus, koko.
Keskusmuisti	Luotettavuus, nopeus, määrä.
Anturit ja näytöt	Luotettavuus, mekaanisen rasituksen kesto
Energian kulutus (akku)	Luotettavuus, sähköturvallisuus, kapasiteetti.

Kaikissa laitteiston osissa on vaatimuksena luotettavuus ja kestävyys. Sulautetun järjestelmän laitteiston komponenttien luotettavuudella saavutetaan monta muutakin hyötyä kuin loppukäyttäjän tyytyväisyys. Laitteen testaaminen on huomattavasti helpompaa, jos testin epäonnistuessa ei tarvitse aina epäillä rikkinäistä komponenttia. Testauksessa ja erityisesti integraatiovaiheessa laitteistoa saattavat testata puhtaasti ohjelmisto-osaajat, joilla ei välttämättä ole edes laitteistoa komponenttipohjaisten ongelmien selvittämiseen tai ratkaisemiseen. Tämän lisäksi mahdolliset korjaukset tulevat melkein poikkeuksetta ohjelmistoon, vaikka syy olisikin laitteistossa. Nokialla laitteistoa testattiin huomattavasti aikaisemmin ennen kuin varsinainen ohjelmisto oli valmis. Tämä tarkoitti sitä, että laitteistoa testattiin edellisen sukupolven tekniikalla. Lisäksi laitteisto oli päässyt testeistä lävitse vanhalla ohjelmistolla, joten uuden ohjelman aiheuttaessa ongelmia laitteistolle, voidaan muutokset tehdä vasta seuraavan sukupolven laitteistoon. Tämä on tietenkin sulautettujen järjestelmien lähtökohdista poikkeuksellista, mutta aikataulut ja markkinat aiheuttavat paineita erityisesti laitteistojen kehityssyklien minimoimiseen. Yleisesti voidaan sanoa, että laitteiston vaatimukset ja ohjelmiston vaatimukset käsitellään erillisinä kokonaisuuksina. Tässä piilee juuri se perimmäinen virhelähde, joka rajoittaa sulautettuja järjestelmiä. (Henziner & Sifakis, 2007)

2.3 Ohjelmisto

Sulautettujen järjestelmien ohjelmisto sisältää usein laitteistoläheistä ohjelmointia. Projektin laitteisto antaa omat reunaehdot muistille, tallennustilalle, prosessorille. Kaikki nämä täytyy ottaa huomioon ohjelmistoa suunnitellessa. Laitteisto tulee olla projektin tavoitteisiin sopivasti mitoitettu, mutta projektista riippuen voidaan joutua käyttämään esimerkiksi tiettyä prosessoriarkkitehtuuria. Näin ei voida aina käyttää jo entuudestaan opittuja käytäntöjä, vaan joudutaan opettelemaan uusia tapoja ratkaista samantyyppisiä ongelmia. Esimerkkinä on siirtyminen yksiytimisistä prosessoreista moniytimisiin prosessoreihin. Samat asiat jouduttiin tekemään uudella tavalla. Sulautetun järjestelmän ohjelmiston kompleksisuus ja koko voivat vaihdella kevyestä mikrokontrollerin logiikan käytöstä aina Symbian:n tai Android:n kaltaisiin reaaliaikakäyttöjärjestelmiin. Sulautettujen prosessien ohjelmisto suunnitellaan kuitenkin niin, että se toteuttaa vain tarvittavat rutiinit selvitäkseen tehtävästään. Näin mitoitettuna ohjelmisto joudutaan aina suunnittelemaan niin, että se käyttää muistia, tallennustilaa, prosessoria ja energiaa niin vähän kuin mahdollista. Näin laitteen kokonaiskustannukset pysyvät alhaisempina, koska kaikki turhat ominaisuudet lisäävät komponentti kustannuksia ja vähentävät laitteen mahdollista akun kestoa. Sulautettujen järjestelmien suuntautuessa massamarkkinoihin, niiden kustannuspaineet alkavat näkyä myös ohjelmistojen tuottamisessa. Onnistuneen projektin jälkeen leikataan kustannuksia esimerkiksi niin, että käytetään samoja menetelmiä ja samoja työkaluja uuteen, seuraavan sukupolven laitteeseen. Tämä aiheuttaa yleisiä ratkaisumalleja ja se johtaa suoraan laitteen tehokkuuden hitaaseen, mutta varmaan alenemiseen. (Henziner & Sifakis, 2007)

2.3.1 Laitteistoläheinen ohjelmointi

Laitteistoläheisellä ohjelmoinnilla tarkoitetaan alimman tason ohjelmointia, joka ajetaan kääntäjältä suoraan prosessorin ohjelmistomuistiin. Laitteistoläheinen ohjelmointi on tavallista sulautettujen järjestelmien projektissa, sillä juuri laitteiston ja ohjelmiston rajapintana toimii tämä taso. Usein puhutaan firmwaresta, kun tarkoitetaan tätä tasoa. Laitteiston tehokas käyttö vaatii alimman tason hyvää hallintaa, sillä ongelmat perustuksissa heijastuvat koko projektiin. Laitteistoläheinen ohjelmointi mahdollistaa myös ongelmien tehokkaamman ratkaisun verrattuna yleisiin ratkaisuihin. Laitteistoläheisen ohjelmoinnin lähtökohta on se, että toteutetaan haluttu ominaisuus

niin, että se käyttää tehokkaasti hyväkseen laitteiston antamat mahdollisuudet. Näitä voivat olla esimerkiksi prosessorin omat palvelut, joiden käyttämättä jättäminen ja niiden toteuttaminen esimerkiksi pelkästään koodaamalla on sekä prosessorin ominaisuuksien haaskaamista että ohjelmoijan ajan tuhlaamista. Tämä tarkoittaa sitä, että ohjelmoijan on tunnettava tarkasti laite, jolle ohjelmisto tulee. Kolmannen osapuolen käyttäminen laitteistoläheiseen ohjelmistoon on näin vaikeaa.

2.3.2 Välitason ohjelmointi

Välitason ohjelmointi (*eng. middleware*) on laitteistoläheisen ohjelmiston, kuten käyttöjärjestelmän ja loppukäyttäjälle näkyvien sovellusten väliin rakennettu taso. Toisaalta koko ohjelma saattaa olla laitteiston kiinteään muistiin kirjoitettuna firmwaressa, jolloin välitasoa ei tarvita, mutta välitasoa voidaan käyttää myös niin, että se häivyttää sovelluksen tarpeen tietää tarkasti kuinka käyttöjärjestelmä toimii. Tämä lisää abstraktiotasoa verrattuna alempaan laitteistoläheiseen tason ohjelmointiin. Välitasoa voidaan käyttää myös laitteistoläheisen ohjelmoinnin helpottamiseksi ylimmän tason vaatimuksiin nähden. Välitason etu on se, että sovelluksen loppukäyttäjälle tekevän koodaajan ei tarvitse tietää miten signaali käsitellään sen tullessa kannettavaan laitteeseen. Välitaso hoitaa tiedon välityksen aina portilta sovellukseen ja sovelluksesta välitason lävitse takaisin porttiin. Tämä taso lisää kuitenkin laitteiston vaatimuksia ja siitä ei ole hyötyä, jos laite on suunniteltu yhteen täsmälliseen tarkoitukseen. Täsmällisessä tapauksessa saadaan paremmat vasteet laitteistosta ja ohjelmistosta, jos ohjelma voi suoraan ajaa haluamiaan prosessorin palveluja. Välitason ohjelmointi on parhaimmillaan, jos laitteisto vaihtuu, mutta sovellusten tekijät haluavat käyttää samaa ohjelmistoympäristöä. Tämä pätee myös silloin kun ohjelmisto pysyy samankaltaisena ja laitteisto vaihtuu usein.

2.3.3 Sovellukset

Sovellusten toteuttaminen sulautetulle järjestelmälle vaihtelee suuresti. Alimmalla tasolla voidaan kirjoittaa koko toiminnallisuus yhteen ohjelmaan, joka voi sijaita firmwaressa eli laiteohjelmistossa. Käytetään esimerkkinä matkapuhelinta. Matkapuhelin on vain alusta, johon voi kuka tahansa ohjelmoida omaa ohjelmaa välittämättä edes käyttöjärjestelmän versiosta. Suurimmat erot sovelluksen kannalta

liittyvät siihen, miten paljon abstraktiota liittyy itse alustaan, jonka päälle sovellus toteutetaan. Pesukoneen tapauksessa sovellus käyttää suoraan sähkömoottoria ja abstraktiotaso on erittäin matala. Jos kyseessä on kännykkä, niin välitaso hoitaa pääasiassa kaiken liikenteen laitteiston ja käyttöjärjestelmän välillä. Näin sovelluskehittäjän ei tarvitse käyttää edes käyttöjärjestelmän palveluja suoraan vaan sovellus käyttää sille rakennettua hiekkalaatikkoa, joka välittää sen viestit käyttöjärjestelmälle. Mitä korkeammalle tasolle mennään sovellusten näkökulmasta, niin sitä vaikeampi on erottaa, onko kyseessä sulautetun järjestelmän projekti vai puhdas ohjelmistoprojekti. Tähän Nokia pyrki omalla Symbian-projektillaan, kun se julkaisi oman SDK:n (Software Development Kit). Tämä antoi kolmannelle osapuolelle mahdollisuuden kirjoittaa omia sovelluksia ja ajaa niitä matkapuhelimessa. Android-puhelinten suosion yksi syy on juuri se, että laitteistotaso on saatu häivytettyä niin hyvin, että yhdellä SDK:lla voi ohjelmoida sovelluksia melkein kaikkiin saatavilla oleviin Android-puhelimiin. Kuten edellä mainitaan, yleisemmät ratkaisut luovat myös ongelmia. Varsinkin reaaliaikakäyttöjärjestelmissä abstraktion kasvaessa voidaan päätyä tilanteisiin, joita ei ole osattu ennakoida tai laitteen kapasiteetti ei riitä. Ohjelma kasvaa eksponentiaalisesti abstraktiotason noustessa ja myös koodausvirheiden mahdollisuus kasvaa.

3 Ohjelmistotuotannon metodit

Metodi määritellään Wikipediassa seuraavasti: ”**Metodi** tarkoittaa menetelmää, tapaa suorittaa määrämuotoisesti askel askeleelta edistyvä toimintoketju, jossa saavutetaan tavoiteltu tehtävä tai päämäärä. Metodin määritelmä on lavea ja sanan metodi rinnalla voidaan tässä työssä käyttää myös mallia tai menetelmää. Ohjelmistotuotannon kannalta metodi vastaa aina kahteen perus kysymykseen:

- Missä järjestyksessä työt tehdään?
- Milloin siirrytään seuraavaan vaiheeseen?

Työllä tarkoitetaan koodaamista, dokumentointia, projektin hallintaa ja kaikkia niitä projektin osia, jotta tuote saadaan valmiiksi. (Boechm, 1988) Työn vaiheilla tarkoitetaan kaikkia eri tapahtumaketjun osia, joiden päätteeksi tuote on valmis. Se miten määritellään valmis, vaihtelee eri metodien välillä. Esimerkiksi vesiputousmallissa se on valmis testausta varten, kun ohjelmisto saadaan valmiiksi, mutta ketterässä metodissa osaohjelma ei ole valmis ennen kuin sen kirjoittanut henkilö on testannut sen. Lisäksi eri metodit voivat painottaa eri asioita tai vaiheita projektissa. Esimerkkinä on ketterien menetelmien tapa painottaa toimivaa ohjelmaa tai demoa enemmän kuin kattavaa dokumentaatiota.

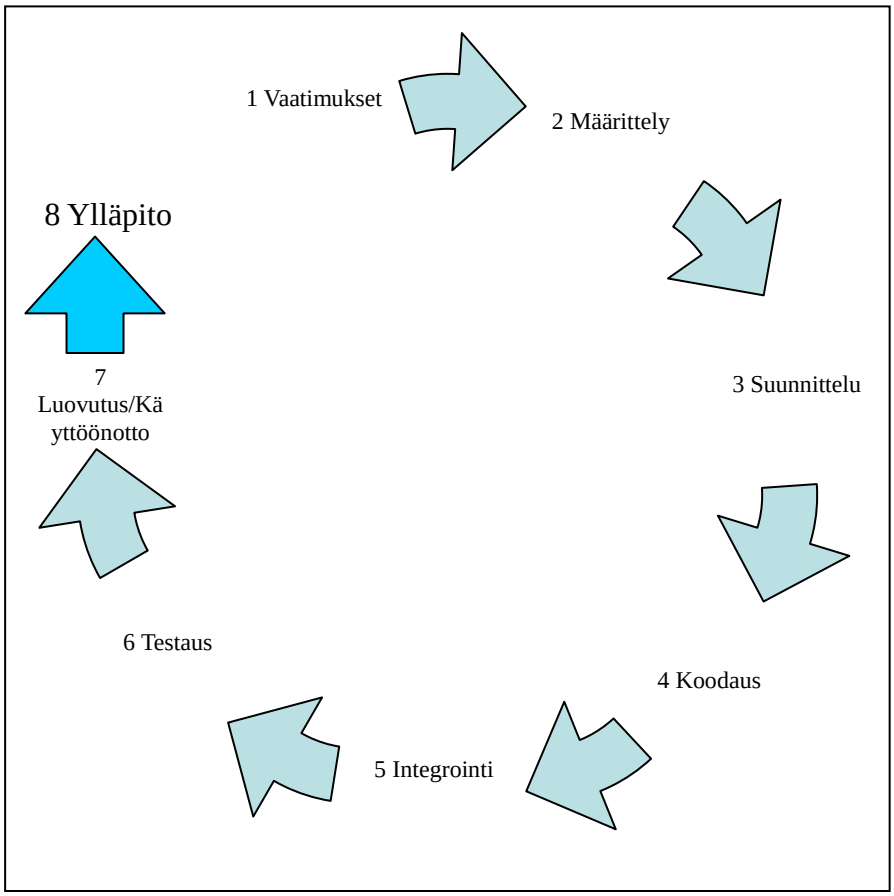
Ohjelmistotuotannon tavoite on tuottaa toimiva ohjelmisto sen tilaajalle hänen asettamien vaatimusten mukaisesti metodista riippumatta. Asiakkaiden vaatimusten käsittely ja asiakkaan osallistuminen tilaamaansa projektiin ovat riippuvaisia käytettävästä metodista.

Seuraavaksi käsitellään kolmea eri metodia: suunnitelmakeskeiset menetelmät (*eng. Plan-driven method*), evoluutiomalli (*eng. Evolutionary model*) ja ketterät menetelmät (*eng. Agile methods*).

3.1 Suunnitelmakeskeiset metodit

Ohjelmistokehitysmenetelmän suunnitelmakeskeisyydellä tarkoitetaan menetelmän

sisältävän paljon suunnittelutyötä ennen varsinaisen toteutuksen alkamista. (*eng. Plan/specification driven*) Ohjelmiston elinkaari on esitetty kuvassa 1. Ohjelmiston kehitys nähdään elinkaarena, joka alkaa tarpeiden määrittelystä ja päättyy ylläpitoon. (Haikala Märijärvi, 2004)



Kuva 1. Ohjelmiston elinkaari (Haikala Märijärvi, 2004)

Ohjelmiston elinkaari kuvaa hyvin myös vesiputousmallin eri vaiheita. Ensimmäisessä vaiheessa asiakkaan vaatimukset kuullaan ja ne kirjataan. Toisessa vaiheessa asiakkaan vaatimukset määritellään erikokoisiksi osakokonaisuuksiksi. Vaatimuksiin lisätään ne vaatimukset, joita asiakas ei ole osannut pyytää. Näitä voivat olla esimerkiksi tietoturva tai esimerkiksi lakiin liittyvät asiat. Määrittelyn jälkeen tulee suunnittelu. Tässä suunnitellaan ohjelmistoarkkitehtuuri ja tehdään suunnitteludokumentit valmiiksi. Suunnitteludokumentit annetaan eteenpäin ohjelmiston koodaajille. Ohjelmointi suoritetaan tavalla, joka on määritelty arkkitehtuuridokumenteissa ja muissa dokumenteissa. Tällä ei ainoastaan tarkoiteta tiettyä tapaa ratkaista ohjelmallisesti asiakkaan ongelma, vaan tapaa kirjoittaa koodia. Esimerkiksi muuttujien nimeämiseen

liittyvät säännöt löytyvät suunnitteludokumentaatiosta. (Haikala Märijärvi, 2004) Integroinnissa yhdistetään koko koodi yhdeksi ohjelmaksi tai jos ohjelmisto on suuri, se kootaan ensin osakokonaisuuksiin ja ne kokoamalla ohjelmisto on valmis. Testauksessa käydään lävitse asiakkaan vaatimukset tehdyllä ohjelmalla. Viimeisenä vaiheena on luovutus ja ylläpito. Ylläpidolla tarkoitetaan mahdollisia ohjelmistopäivityksiä sekä virheiden korjausta.

3.1.1 Vesiputousmalli

Vesiputousmalli toimii usein esimerkkinä kuinka voidaan toteuttaa ohjelmisto yhdellä kerralla ilman paluuta edelliseen vaiheeseen. Tavoitteena on toteuttaa projekti kuvan 1 tavalla. Kaikki vaiheet tulevat järjestyksessä ja mitään takaisinkytkentää ei ole. Vesiputousmallissa suunta on aina eteenpäin, mutta vaihetta, jossa ollaan menossa, voidaan jatkaa kunnes siihen ollaan tyytyväisiä. Näin esimerkiksi suunnitteluvaihetta voidaan jatkaa niin pitkään, että kaikki asiakkaan vaatimukset ovat saatu tutkittua ja analysoitua. Todellisuudessa on kuitenkin niin, että asiakkaan vaatimukset tulevat muuttumaan kesken projektin ja tämä metodi ei ota sitä huomioon millään tavalla. Vesiputousmalli toimii paremmin periaatteena eri työvaiheista kuin varsinaisena monimutkaisen ohjelmiston tai sulautetun järjestelmän prosessimallina. Vesiputousmalli on kuitenkin hyvin intuitiivinen malli. Normaalisti se on tapa, jolla ihmiset hoitavat arkipäiväiset asiansa. Seuraavassa on esimerkki vesiputousmallin käytöstä: ”Ruoan ostaminen kaupasta”.

Jääkaappi on tyhjä ja sen omistajalla on nälkä. Tämä on projektin vaatimus ja määritelmä on tietenkin saada syötyä. Ensimmäisenä nälkäinen päättää lähteä kauppaan ja laatii kauppalistan. Hän menee kauppaan, suorittaa ostokset ja palaa kotiin. Vesiputousmalli toimii mainiosti, kun asiat ovat yksinkertaisia ja kaikkien osapuolten ymmärtämiä.

Jos kuitenkin lisäämme esimerkkiin kaksi lasta ja vaimon, niin vesiputousmallin käyttäminen muuttuu vaikeammaksi. Kuka laatii kauppalistan? Kuka päättää mitä syödään? Minne kauppaan mennään? Juuri ennen kotiin pääsyä joku muuttaa mielensä siitä mitä haluaakin syödä. Esimerkistä käy selvästi ilmi, että metodiin pitää lisätä keskusteluja ja mahdollisia välitavoitteiden tarkasteluja. Lisäksi kaupassa saattaa olla myynnissä jotain sellaista einestä, mitä päätetäänkin syödä alkuperäisten suunnitelmien

mukaan. Vesiputousmalli on kuitenkin kaikesta huolimatta tarpeellinen ymmärtää, sillä se antaa hyvän pohjan käsitellä kehittyneempiä metodeja. Vesiputousmalli on erittäin hyödyllinen myös referenssinä verratessa eri metodeja keskenään.

Koska vesiputousmalli ei palaa taaksepäin työvaiheissa, se aiheuttaa ongelmia, jos määrittely alussa ei ole täydellistä. Lisäksi se ei kykene reagoimaan vaihtuviin vaatimuksiin kesken projektin. Vesiputousmallilla ei ole keinoa lisätä vaatimuksia myöhemmin. Tämän päivän ohjelmistoprojekteissa pitää ottaa myös se huomioon, että ihmisten eli työväen liikkuvuus on suurta. Vesiputousta käytettäessä tämä tulee suureksi ongelmaksi. Ihmisten vaihtuvuuden ollessa suurta, on projektien oltava lyhyitä, jotta tiimi tai yksilö saisi kokemusta kaikista vesiputousmallin vaiheista. Mallin syvällinen ymmärtäminen on vaikeaa, jos ei ole käynyt projektin jokaista vaihetta kerran lävitse. Minkä tahansa metodin tai mallin opettelu vaatii kurinalaisuutta ja siinä olevien ihmisten kokemattomuus on haitaksi projektille. Tämä on erityisen totta nimenomaan vesiputousmallissa, sillä huono koodaus tai suunnittelu voi johtaa koko projektin kaatumiseen. Projektin johdolle vesiputousmalli ei anna näkyvyyttä projektin edistymiseen ennen kuin siirrytään vaiheesta toiseen. Tämä tarkoittaa sitä, että projektin johdossa on vaikeuksia ennakoida mahdollisia viivästyksiä tai saada hyötyä mahdollisista harppauksista eteenpäin. Testaukselle vesiputousmalli on huono, koska lopullinen tuote on tarkastettavissa ja testattavissa vasta kun koko projekti on käytännössä tehty. Mahdolliset arkkitehtuuriset ongelmat, tai sulautettujen järjestelmien tapauksessa laitteiston väärät valinnat ovat melkein mahdottomia korjata. (Kettunen & Laanti, 2004) Korjausten kustannukset voivat olla todella suuri osa kokonaisbudjetista. Mitä aikaisemmin testaus löytää virheet ohjelmasta tai laitteistosta, sitä pienemmillä kustannuksilla päästään.

Vesiputousmalli toimii metodina hyvin muun muassa siinä, että turhaa ja ylimääräistä dokumentaatiota ei tule. Kaikki on suunniteltu jo alussa. Koodaajan ei tarvitse itse suunnitella testausta. Projektin johdon kannalta vesiputousmalli antaa vastauksia muun muassa siihen, milloin projekti on valmis. Jos suunnitelma on laadittu hyvin ja toteutus on arvioitu realistisesti, niin johto näkee heti milloin projekti on valmis. Vesiputousmallia käytetäänkin valtavien projektien yläkäsitteenä. Mahdolliset välitavoitteet voidaan ajatella toteutettavan vesiputousmallilla. (Haikala, Märijärvi, 2004)

3.1.2 Inkrementaalinen malli

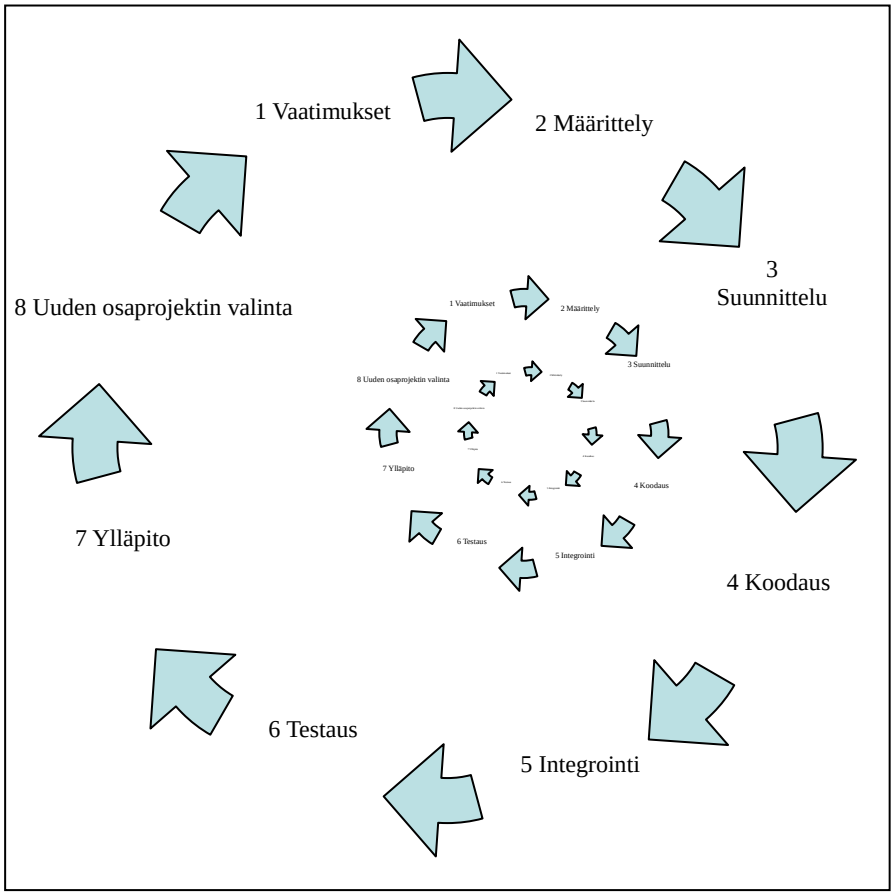
Kun projekti pilkotaan pienemmiksi osiksi ja ne toteutetaan vesiputousmallilla, niin saadaan inkrementaalinen malli. Projekti on valmis, kun kaikki sen osat ovat tehtyjä. Koko projekti joudutaan kuitenkin suunnittelemaan melkein kokonaan alusta asti, mutta liikkumavaraa on jonkin verran esimerkiksi virheiden korjaamiseen. Voidaankin sanoa yleisesti, että jos projektin tavoitteista tiedetään ainakin 80 % voi inkrementaalinen kehitysmalli toimia. Kun projekti pilkotaan pienemmiksi palasiksi, se voi vaikeuttaa koodaajan ja projektin johdon hahmottamista koko projektista. Projektien osat voidaan kuitenkin toimittaa eri tiimeille ja niiden prioriteetteja voidaan vaihtaa. Tämä tarkoittaa sitä, että tiimit voivat erikoistua johonkin osa-alueeseen ja osaava projektin johto voi käyttää tätä erityisosaamista hyödyksi.

Inkrementaalisessa mallissa projektin alussa ensimmäisten projektin osien on valmistuttava ajallaan, jotta voidaan siirtyä projektin muihin vaiheisiin. Projektissa tuotettujen inkrementaalisten osien liittäminen toisiinsa vaatii testaamista. Testaamisesta saadaan suuri hyöty myös lopulliseen tuotteeseen, sillä jos testauksessa löytyy virhe, ei sen korjaaminen ole niin kallista kuin käytettäessä vesiputousmallia. Lisäksi testaamista helpottaa se, että uutta koodia tulee suhteessa vähemmän kuin vesiputousmallissa, jossa käytännössä testattavana on koko koodipohja.

Jos jokin projektin osa myöhästyy, ei inkrementaalisessa mallissa ole mitään keinoa saavuttaa tavoitetta. Lopullisen tuotteen kaikki ominaisuudet ovat pakko toteuttaa ja niitä ei ole mahdollista vaihtaa projektin aikana. Näin aikataulun on pakko joustaa. Käytännössä kaikki resurssien tehokas käyttö ja eri tiimeille suunnitellut osaprojektit eivät tuo ajan säästöä, jos jokin osaprojekti myöhästyy. Kaikki suunnitellut vaiheet on tehtävä suunnitellussa järjestyksessä, mutta ne voidaan toteuttaa rinnan eri tiimien kesken. Mutta jos jokin tiimi ei pysty toimittamaan omaa osuuttaan, kaikki muut tiimit joutuvat odottamaan.

Koko projektin dokumentointi on vaativampaa sillä, eri projektin osien rajapinnat pitää määrittää alussa vaikka osakokonaisuuksia ei ole edes vielä valmiina. Tämä tarkoittaa sitä, että suunnitelmiin joudutaan palaamaan ja niitä muuttamalla joudutaan muuttamaan mahdollisen seuraavan projektin osat. Samaan ongelmaan joudutaan, jos projekti osoittautuu haasteellisemmaksi mitä alun perin osattiin arvata. Yllättävä monimutkaisuuden kasvu voi muuttaa osaprojektia ja näin vaikuttaa kaikkiin loppuihin

osiin. (Haikala, Märijärvi, 2004) Tämä on totta myös kun yhtälöön lisätään mahdolliset laitteiston ongelmat ja muutokset. Projektin myöhemmissä osissa saattaa paljastua, että alun perin määritelty laitteisto on ollut alimitoitettua. Inkrementaalisella kehitysmallilla ei ole keinoa vaihtaa laitteistoa järkevästi toiseen tehokkaampaan. Tämä tarkoittaa sitä, että jos laitteistoa ei ole ennen käytetty esimerkiksi ulkona tai sateessa, on riski valtavan suuri siihen, että vasta lopullisessa vaiheessa laite todetaan toimimattomaksi. Kuvassa 2 on esitetty inkrementaalinen malli, joka kuvaa ensimmäisen osaprojektin kehittymistä. Viimeisessä kohdassa kahdeksan päätetään onko sen hetkinen osuus valmis ennen kuin aloitetaan seuraava osuus projektissa. Toisen projektin alussa voidaan käyttää jo olemassa olevaa dokumentointia, joka on jo tehty ennen ensimmäisen osaprojektin alkamista tai valmistumista. Näin kohta 1 ja 2 voidaan jättää pois. Kierrosta jatketaan, kunnes koko projekti on valmis. Ylläpitoa ei välttämättä ole jokaisen projektin jälkeen, mutta jos muutoksia tulee, niin tähän vaiheeseen tulee mahdollisten edellisessä vaiheessa valmistuneiden projektien dokumentoinnin korjaaminen. Kuvassa 2 on kolme kierrosta kestävä projekti. Inkrementaalinen malli on parhaimmillaan, jos projekti on alusta asti selkeä ja muutoksia tulee projektin alkamisen jälkeen vähän. Tämä malli opettaa tätä käyttävän tiimin tai yksilön yhdessä kierroksessa. Vesiputousmallissa saman henkilön pitää olla alusta loppuun asti käydäkseen kaikki projektin osat läpi. Tämä malli kertoo varsin hyvin sen, mitä pitää tehdä ensin ennen kuin toinen projekti voi alkaa. Tässä on kuitenkin koko mallin suurin ongelma. Jos yksi osakokonaisuus myöhästyy, niin on koko projektin valmistumisaikaa siirrettävä vastaavasti. Tällä metodilla voidaan toteuttaa valtavia projekteja ja jokainen valmistuva projektin osa antaa projektin johdolle merkkipaalun siitä, missä vaiheessa projektia ollaan menossa.



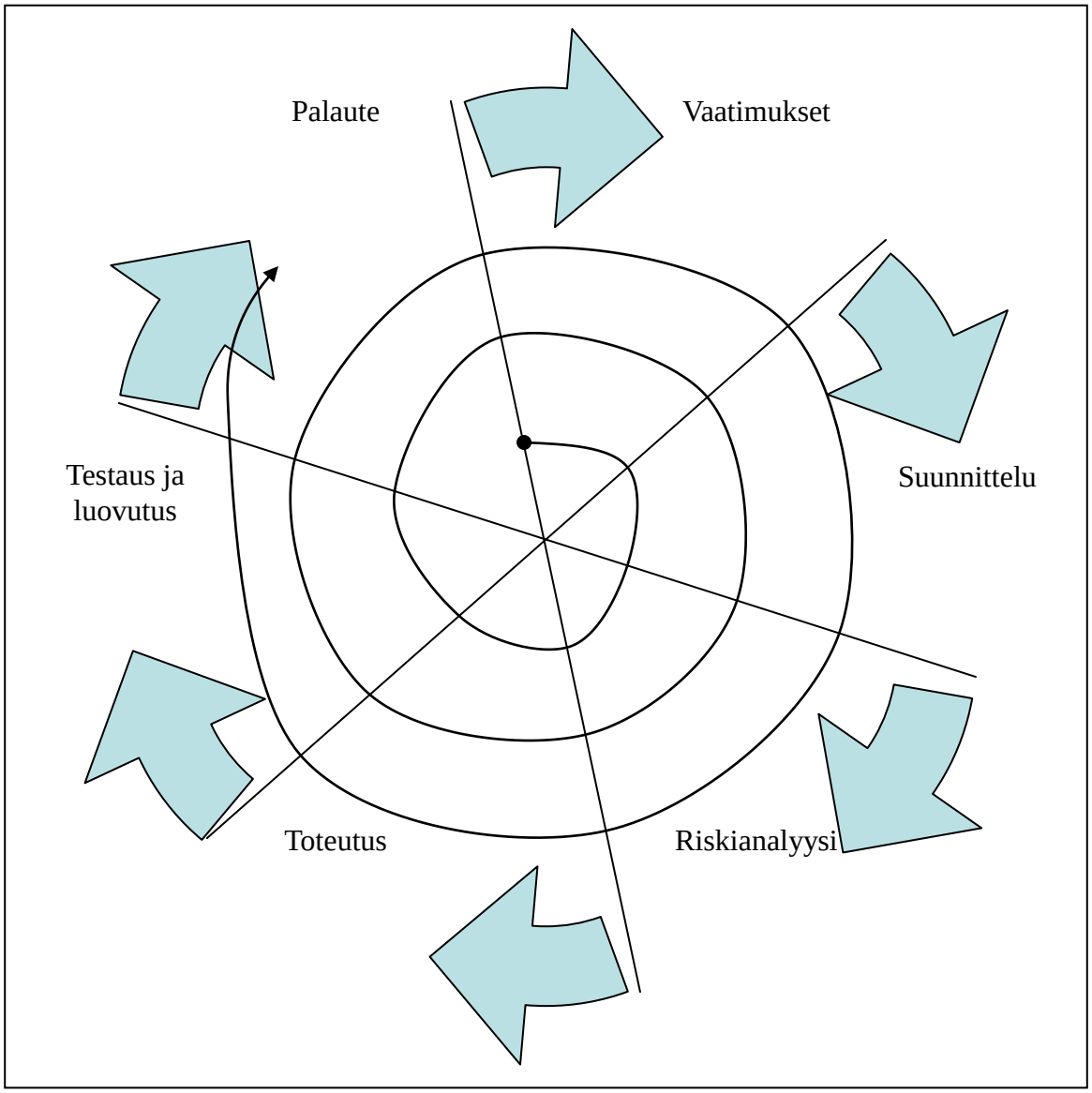
Kuva 2. Inkrementaalinen ohjelmistotuotannon metodi

Inkrementaalinen ohjelmistotuotannon kehitysmalli voi olla myös osittain käytössä. Se voi olla johdon tukena vaikka alemmat tasot ja koodaajat käyttävät ketteriä menetelmiä. Inkrementaalinen malli voidaan myös liittää suoraan vaatimusten hallintaan. Näin tehtiin Nokia Oy:llä minun sinne töihin tullessani 2006.

3.2 Evoluutiomalli

Evoluutiomallilla tarkoitetaan sellaisia ohjelmistotuotannon menetelmiä, joissa kehitetään ohjelmakoodia niin, että se toimii kuten inkrementaalinen malli, mutta suunnitelmat, mahdollinen toteutus ja riskianalyysi tehdään jokaisen inkrementaation päätteeksi. Katsotaan missä mennään ja tehdään suunnitelmat niiden pohjalta. Tämä mahdollistaa liikkumavaraa esimerkiksi asiakkaan toiveissa ja evoluutiomalleissa voidaan palata vanhaan jo tehtyyn koodiin. Tämä malli voi olla, joskus vaikea erottaa koodaa-ja-korjaa metodista, jonka vuoksi vesiputousmalli alun perin kehitettiin. Tähän ajaudutaan, jos projektilla ei ole selkeää tavoitetta. (Boechm, 1988) Inkrementaalista mallia voidaan käyttää evoluutiomallin tapaan. Evoluutiomallia voidaan usein käyttää, sulautettujen järjestelmien niin sanottuun protoiluun, joidenkin inkrementtien päätteeksi

tehdään prototyyppi.



Kuva 3. Spiraalimallin ja sen eri vaiheet

Spiraalimalli on seuraava kehitysaskel inkrementaalisesta mallista. Se toimii lähes samalla tavalla kuin inkrementaalinen malli, mutta spiraalimallin etuna on mahdollisuus suunnitella ensin tärkeimmät ja kriittisimmät osiot, toteuttaa ne ja saada niistä palaute asiakkaalta. Kun palautetta on saatu tarpeeksi, jatketaan olemassa olevilla vaatimuksilla tai vaihdetaan suuntaa. Asiakas voi näin nähdä ja vaikuttaa kerran syklissä tuotteeseen. Lisäksi ohjelmistosuunnittelussa voidaan hyödyntää heti kaikki asiakkaan palaute nopeasti. (Pressman, 1982) Spiraalimallin aikana voi myös tuottaa prototyyppejä. Spiraalimallin huonoina puolina voidaan pitää sitä, että sen luonne on kokeilevaa, eli projektin aikataulut saattavat venyä. Pahimmassa tapauksessa koko projekti ei valmistu.

On erittäin haastavaa toteuttaa sulautettujen järjestelmien laitteistoa, jos vaatimukset muuttuvat jatkuvasti. Spiraalimalli mahdollistaa laitteiston kehittämisen sen omilla ehdoilla. Jos inkrementaalisen mallin testauksessa löytyy virhe, on se korjattava ja koko muu projekti odottaa. Näin ei ole spiraalimallissa, koska seuraava iteraatio voidaan käyttää esimerkiksi vakavan virheen korjaamiseen. (Kettunen & Laanti, 2004) Tämä on hyvä lopputuotteen luotettavuuden kannalta, mutta aikataulut venyvät.

3.3 Ketterät menetelmät

Ketterillä menetelmillä tarkoitetaan metodeja, jotka keskittyvät enemmän koodin tuottamiseen kuin dokumentointiin ja organisaatioon juuri tämän vuoksi ketterät menetelmät pystyvät nopeisiin muutoksiin. Tässä menetelmässä tiimin motivoituminen ja itseohjautuva organisoituminen ovat tärkeitä. Ketterissä menetelmissä asiakkaan tapaamisen valinta ja toimivan sovelluksen esittäminen ovat tärkeämpiä kuin dokumenttien esittäminen. Tästä hyötyy asiakas ja työtä tekevä tiimi. Asiakkaan ei tarvitse osata täydellisesti kuvailla kaikkia vaatimuksia kerralla, vaan tämä on jatkuvassa yhteydenpidossa työtä tekevän tiimin kanssa. Ketterissä menetelmissä keskitytään nopeaan reagointiin kehityssuunnitelmien muuttuessa, millä varmistetaan, että valmistuttuaan ohjelmisto vastaa hyvin asiakkaan tarpeita.

Ketterät menetelmät sai alkunsa niin sanotusta agile-manifestista, joka julkaistiin 2001. Siinä on seuraavat periaatteet: ”Me etsimme parempia keinoja ohjelmistojen kehittämiseen tekemällä sitä itse ja auttamalla siinä muita. Tässä työssämme olemme päätyneet arvostamaan

- **Yksilöitä ja vuorovaikutusta** enemmän kuin prosesseja ja työkaluja
- **Toimivaa sovellusta** enemmän kuin kokonaisvaltaista dokumentaatiota
- **Asiakasyhteistyötä** enemmän kuin sopimusneuvotteluita
- **Muutokseen reagoimista** enemmän kuin suunnitelman noudattamista.

Vaikka oikeallakin puolella on arvoa, me arvostamme vasemmalla olevia asioita enemmän.” (http://www.agilealliance.org/ 2013)

Tämä tarkoittaa sitä, että yksilöllä eli asiakkaalla tai tiimin jäsenellä on sananvaltaa vaatia muutoksia. Näin muutos tuotteeseen voi lähteä niin asiakkaasta kuin ohjelmoijasta. Toimivan sovelluksen näyttäminen asiakkaalle on tehokkaampaa ja kuin

yrittää kuvailla omia halujaan ja mielikuviaan lopputuotteesta kirjallisesti. Käytännössä on todettu, että kun asiakas näkee puolivalmiin tuotteen, niin hän vasta alkaa ymmärtää mitä on saamassa. Tällöin projektin oletettu keskivaihe voi olla vasta alku. Ketterät menetelmät muuttavat asiakkaan ja toimittajan vastakkainasettelun yhteistyöasetelmaksi, jossa asiakas integroituu tiimiin tavalla tai toisella. Tämä luo suoran yhteyden asiakkaaseen ja asiakas ymmärtää paremmin työtä tekevää tiimiä. Ketterissä menetelmissä on myös piirteitä, jotka ravistavat koko ajatusta siitä hierarkiasta, joka on vanhan tehdasteollisuuden perinteitä. Vastuu kuuluu niille, jotka tuottavat koodin, ja tätä kautta voidaan jopa vähentää keskijohtoa, sillä tuotteen omistajien (*eng. product owner*) tehtävä on pitää prioriteetit tiimille järjestyksessä, mutta varsinaisen ratkaisun tuottaminen kuuluu tiimille. Näin saadaan optimaalisia ratkaisuita teknisestä näkökulmasta, sillä tekniseen täydellisyyteen pyrkiminen on harvemmin johdon toivelistalla. Tässä on toki löydettävä tasapaino, mutta valmis määritelmä on kuitenkin lopputuotteen kannalta kehittäjillä. Tämä valta ja vastuu aiheuttavat paineita muun muassa rekrytoimiseen, sillä kaikkien odotetaan ottavan vastuuta.

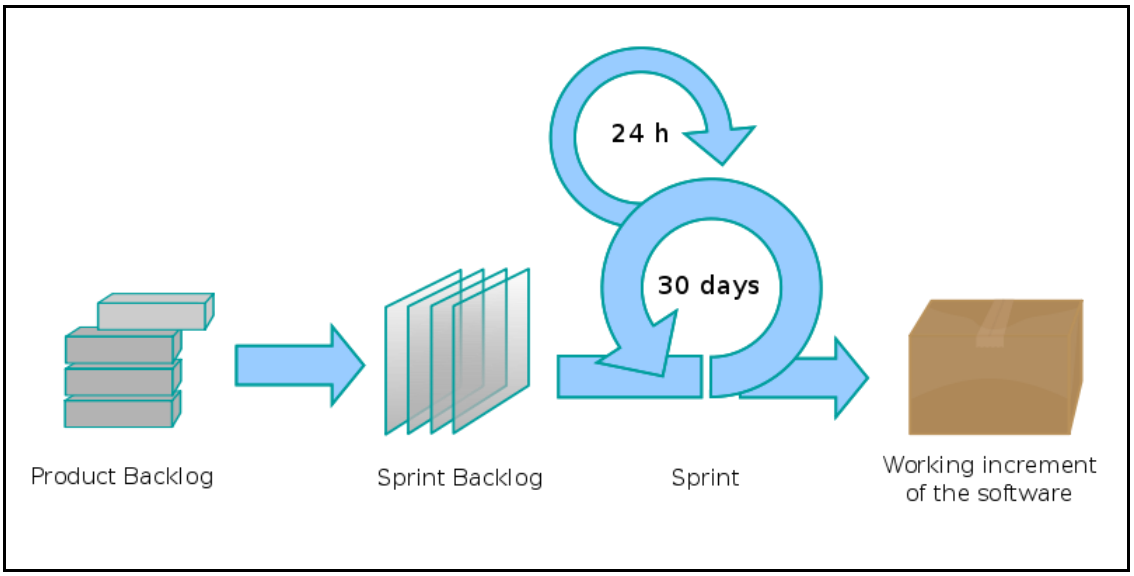
Muutokseen reagoimista ei pidetä pahana, vaan siihen ollaan kaikin keinoin valmiita. Paradoksaalista on kuitenkin se, että ketterät metodit suojelevat ohjelmoijia hyvinkin tehokkaasti yllättäviltä muutoksilta. Ketterien menetelmien tärkeä termi on valmiin määritelmä. Tämän määrittää usein kyseistä tehtävää tekevä yksilö. Tiimin pitää yhdessä sopia mitä tämä tarkoittaa. Onko esimerkiksi testattu tuote vasta valmis? Entäs, jos ominaisuutta ei voi testata ennen, kuin toinen osakokonaisuus on valmis? Eli valmiin määritelmä voi vaihdella projektista ja tiimistä riippuen. Ketterät menetelmät tai metodit sisältävät erilaisia tapoja painottaa asioita. Yleistä niille on niiden suuri ero perinteisempiin menetelmiin siitä, miten kommunikaatio ja vastuu jaetaan. Perinteisemmissä menetelmissä, kuten inkrementaalisessa metodissa, vastuu työstä on sillä tiimillä, joka toteuttaa haluttua osaa ohjelmistosta.

Ketteriä menetelmiä on lukuisia. Niiden lähtökohdat ovat kuitenkin samat, mutta erot ovat suuret. Ketteristä menetelmistä voidaankin sanoa, että ne poikkeavat perinteisistä menetelmistä siksi, että ne eivät ole minkään perinteisen menetelmän jatkokehittelyn synnyttämiä variaatioita. Ketterät menetelmät poikkeavat silti niin paljon toisistaan, että

ne on käsiteltävä erilisinä, jotta voitaisiin todella analysoida niiden käyttöä ja mahdollisia hyötyjä sulautettujenjärjestelmien kannalta.

3.4 Scrum

Scrum-metodissa ei varsinaisesti määritä sitä tai niitä tapoja, jolla ohjelmisto tuotetaan. Scrum antaa ihmisille toimintatavan vuorovaikutukseen sekä keinot kehittää ohjelmistoa nopeasti vaihtuvista vaatimuksista huolimatta. Scrummiin kuuluu kolme tärkeää termiä: Product backlog, Sprint backlog sekä toimitettava tuote eli product. Näistä ensimmäinen product backlog on se, mikä määrittää koko tuotteen. Tämä lista kokoaa kaikki asiakkaan vaatimuksista eli kertomuksista saadut erilaiset tehtävät ja niiden asiakkaan antaman tärkeysjärjestyksen. Sprint backlog tarkoittaa sitä osaa tehtävistä, mikä toteutetaan yhdessä sprintissä. Toimitettavalla tuotteella tarkoitetaan toimitettavaa lopputulosta, joka toimitetaan tai demonstroidaan asiakkaalle. Kuvassa 4 on esitetty nämä termit ja miten ne liittyvät ketterään scrum menetelmään.



Kuva 4. Scrum menetelmän termistö ja yhden iteraatiokierroksen kuvaus.

Scrum-menetelmään kuluu kolme erilaista roolia. Ensimmäinen on tuotteen omistaja, joka vastaa tuotteen arvosta ja siitä mikä tuottaa tuotteelle lisäarvoa. Arvolla tarkoitetaan rahallista arvoa tai tuottavuutta. Tuotteen omistajan vastuulla on arvioida milloin asiakas on tarpeeksi tyytyväinen tuotteeseen. Hän keskustelee asiakkaan kanssa, mitkä ominaisuudet toteutetaan ja missä järjestyksessä. Näistä ominaisuuksista ja vaatimuksista syntyy lista, jota sanotaan Product backlogiksi. Tämä lista käydään

jokaisen sprintin alussa läpi.

Toinen tärkeä rooli on scrum-masteri. Scrum-masterin tehtävä on pitää ohjelmistotiimi tuottavana ja mahdollistaa päivittäinen työ. Hän toimii linkkinä tuotteen omistajan ja tuotetta koodaavan tiimin välillä. Jos vaatimuksissa on jotain epäselvää tai jokin muu syy estää tiimin työtä, on scrum-masterin tehtävä ratkaista ongelma. Hänen tehtävänsä on myös pitää huolta siitä, että kaikilla tiimin jäsenillä on tarpeeksi töitä. Tiimi ottaa vastuuta eri työtehtävistä itsenäisesti. Tuotteen omistajan antama product backlogin tehtävät arvioidaan ja jaetaan tiimin sisällä ihmisten omien halujen ja taitojen mukaan. Scrum-masterin tehtävä on pitää huolta, että tiimi tekee tämän ja että tiedot erilaisista ratkaisuista eivät jää vain niitä tekevän yksilön tietoon. Kolmas rooli on kehittäjä. Hän vastaa yksittäisten tehtävien eli taskien toteuttamisesta. Hän voi olla koodaaja, arkkitehti, testaaja tai muu tuotetta tekevä tiimin jäsen. Kehittäjä voi myös kaksoisroolissa toimia samalla scrum masterina.

Scrum-metodiin kuuluu säännöllisiä kokoontumisia. Päivittäistä kokousta kutsutaan Scrum palaveriksi. Tämä kokous pidetään Scrum-masterin ja tiimin yhteisenä. Tässä kokouksessa käydään lyhyesti lävitse, mitä kukin tekee tänään ja onko mitään mikä estää työtehtävien tekemisen tai jokin ongelma missä tarvitaan kollegoiden apua. Scrum-kokouksen tarkoituksena on ratkaista mahdolliset ongelmat sekä yleisesti ymmärtää mitä muut tiimissä tekevät. Tämän lisäksi kukin tiimin jäsen voi kertoa mitä on tekemässä ja mitä aikoo tehdä seuraavaksi.

Scrummissa yhtä iteraatiokierrosta kutsutaan sprintiksi. Tämä sprintti kestää 2–4 viikkoa. Sprintin ensimmäistä kokousta kutsutaan nimeltä sprint planning. Tässä kokouksessa tiimi ja tuotteen omistaja käyvät lävitse ominaisuuksia, joita halutaan tuotteeseen. Kokouksessa tiimi arvioi tehtävien vaativuutta ja antaa jonkinlaista ideaa siitä, miten pitkään tehtävässä kuluu aikaa. Kun sprintti on loppumassa, tiimi pitää katselmoinnin, jossa esitellään uudet ominaisuudet ja parannukset tuotteen omistajalle. Tässä katselmoinnissa on asiakas yleensä mukana. Viimeisenä, ennen kuin aloitetaan uusi sprintti, suoritetaan vielä sprintin palauteosuus. Tässä keskitytään miettimään sitä, mitä tehtiin hyvin ja mitä voisi parantaa prosessissa. Tämä pidetään tiimin sisäisenä ilman asiakasta.

Kuvassa 4 on lisäksi esitetty 24 tunnin päivittäinen vaihe ja 30 päivän kierto, joka tarkoittaa sprintin pituutta. Tämä voi vaihdella riippuen tiimin omista mieltymyksistä.

Sprintin ollessa kesken, ei uusia tehtäviä oteta enää tiimin kuluvaan sprinttiin. Tämä tarkoittaa sitä, että jos haluttu ominaisuus ei pääse sprinttiin huomenna, voi ominaisuuden valmistumista joutua odottamaan 60 päivää. Tämä sprint backlogin lukitseminen mahdollistaa tiimin rauhallisen työn, koska muutokset eivät vaivaa ohjelmistoa rakentavaa tiimiä. Muutokset kirjataan kuitenkin ylös ja niihin palataan aina jokaisen sprintin alussa.

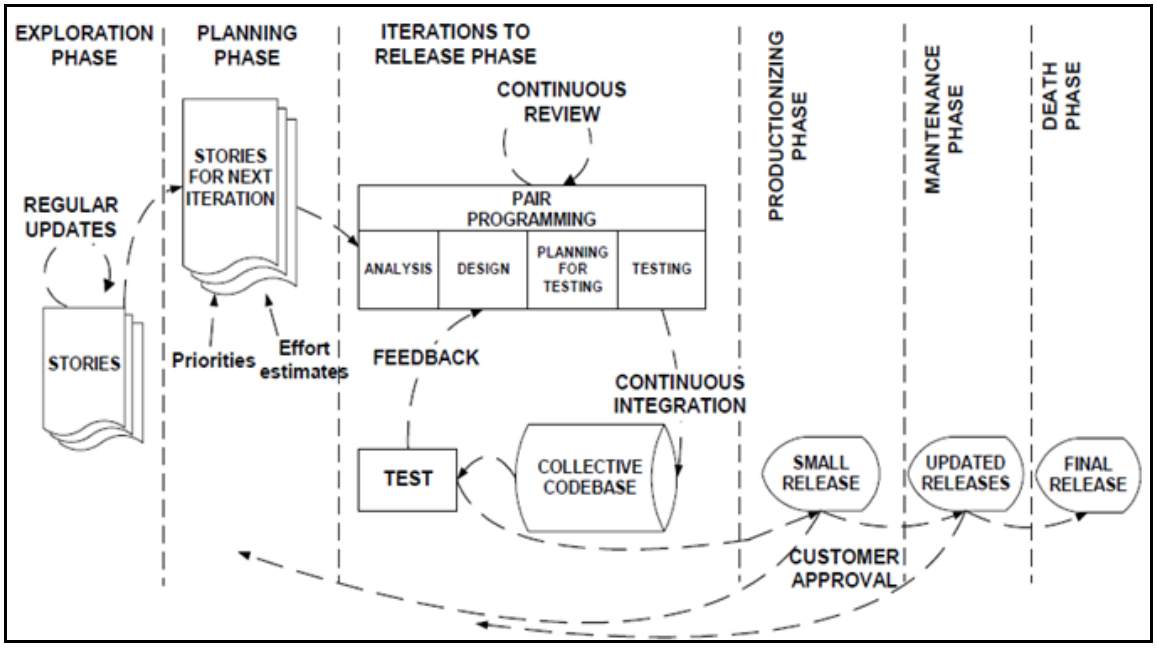
3.5 Extreme Programming

Extreme Programming on ketterä menetelmä, jonka käyttö on levinnyt laajalle. Se on suunniteltu pienille tiimeille. Ytimeltään XP (Extreme Programming) toteuttaa seuraavia teesejä: yksinkertaisuus, palaute, kommunikaatio ja rohkeus. (Jeffries & Lindstrom, 2004) Näiden sanojen merkitys korostuu, kun niitä vertaa ei-ketteriin menetelmiin, joita esittelin edellä. XP:ssä korostetaan muutoksiin tarttumista eli rohkeutta, kun perinteisissä menetelmissä tämän tilalla on muutosvastarinta. Hyvänä esimerkkinä on Nokia Oy:ssä käytetty termi: “Liikaa riskiä”. Tällä perusteltiin jo valmiita sovelluksia, joita ei uskallettu laittaa seuraavaan tuotteeseen. Rohkeuden lisäksi olisi vaadittu myös notkeutta, jota silloinen organisaatio tai käytetty ohjelmistotuotannon menetelmä eivät tarjonneet. XP antaa rohkeudelle myös uuden merkityksen. On rohkeaa laittaa lopulliseen tuotteeseen viimehetken innovaatio, mutta todellista rohkeutta on myös se, että yritys tai esimerkiksi laiteportfolion omistaja luottaa yksittäiseen insinööriin, joka vakuuttaa ohjelmiston olevan valmis. XP:n toinen periaate, eli kommunikaatio, tukee myös rohkeutta. Organisaation rohkaistessa siinä työskenteleviä ihmisiä, ovat he avoimempia myös kuulemaan muiden innovaatioita tai ongelmia. Kommunikaation lisääntyessä muutkin puhuvat rohkeasti vaikeuksistaan ja onnistumisistaan, joka vakuuttaa muita tiimissä tekemään myös niin. Palaute tulee luonnollisesti, jos kommunikaatio ja rohkeus löytyvät tiimiltä. Palaute korostuu varsinkin asiakkaan ja työtä tekevän tiimin välisessä vuorovaikutuksessa. Yksinkertaisuus tarkoittaa sitä, että organisaatio on mahdollisimman matala. Koko XP:n onnistumisen kannalta olennaista on kommunikaatio ja tälle vaatimuksena on yksinkertainen ja matala organisaatio (Jeffries & Lindstrom, 2004).

XP:llä on samankaltainen viikkorytmi kuin scrummilla. XP:n tapauksessa iteraatio eli sprintti on kaksi viikkoa. Projektin alkaessa on kuitenkin tutkimusvaihe. Tutkimus vaihe (*eng. Exploration Phase*) kestää yleensä viikon. Asiakas on mukana jokaisessa

vaiheessa. Kuvassa 5 on esitetty XP projekti alusta loppuun. Tarinat (*eng. Stories*) ovat kertomuksia siitä, miten asiakas näkee lopputuotteen. Tutkimusvaiheessa luodaan tarina siitä, miten käyttäjät käyttävät laitetta. Asiakkaan on helpompi kertoa miten esimerkiksi laite palvelee asiakasta eri tilanteissa kuin kertoa yksittäisiä vaatimuksia.

Tutkimusvaiheen jälkeen tiimi saa käsiteltäväksi tarinoita. Nämä asetetaan asiakkaan kanssa tärkeysjärjestykseen. Tarinat toteuttava tiimi antaa arviot miten kauan eri tarinoiden toteuttamiseen menee. Tämä voi olla aikaa, vaivaa tai muuta suuretta määrittävä termistö tai numero. Usein kuitenkin ketterissä menetelmissä käytetään jotain muuta kuin selkeää numeroa. Tämä tarkoittaa sitä, että tiimi voi sanoa pystyvänsä yhdessä iteraatiossa toteuttamaan kolme isoa tarinaa ja kymmenen pientä. Tässä piilee ketterien menetelmien vahvuus. Tiimi tietää mitä se kykenee toimittamaan. Mikä on pieni tai iso tarina, riippuu kokonaan tiimistä. Näin asiakas saa parhaan mahdollisen arvion projektin kokonaiskestosta nimenomaan niiltä ihmisiltä, jotka tekevät työn. Perinteisissä menetelmissä suuri ongelma on juuri arvio siitä, miten kauan tuotteen tekeminen kestää, ja saadaanko siihen mahtumaan vaaditut toiminnallisuudet (feature). (Jeffries & Lindstrom, 2004)



Kuva 5. XP:n eri vaiheet (Abrahamsson et al, 2002)

Kuvassa 5 on esitetty suunnitteluvaihe. Suunnitteluvaihe on kahden viikon iteraation ensimmäinen vaihe. Jokaisen iteraation ja toimituksen jälkeen voidaan palata tähän

vaiheeseen, kunnes projekti siirtyy loppuvaiheisiin. Suunnitteluvaiheen jälkeen XP etenee iteraatiovaiheeseen. Tämä vaihe toistetaan kunnes kaikki halutut tarinat on toteutettu. Asiakas päättää sen milloin tuote on valmis. Hän tekee sen tuotantovaiheen päätteeksi. Iteraatiovaiheessa suoritetaan varsinainen tuotteen rakentaminen eli koodaaminen.

XP:n tapauksessa ohjelman kirjoittaminen toteutetaan *pariohjelmoinnilla*. Pariohjelmoinnissa kaksi ohjelmoijaa tekee työtä vuoroin. Toinen ohjelmoi ja on varsinaisesti näppäimistöllä. Toinen ohjelmoija seuraa ja korjaa virheitä ja osallistuu muuten ohjelmiston tuottamiseen. Tämä vaikuttaa suoraan ohjelmoinnin laatuun ja poistaa virheitä. Lisäksi ohjelmointikuri ja yleinen tietämys ohjelmiston toteuttamisesta leviävät koodaajien kesken. Ainakin kaksi koodaajaa on nähnyt toteutetun toiminnon ja näin ollen esimerkiksi virheiden etsintä on kaksi kertaa helpompaa, kuin että kaikki koodaisivat omilta koneiltaan yksin. Se, että yksi ihminen tuottaa koodia, ei tee tästä tavasta yhtään sen hitaampaa, kuin että nämä toimisivat omilta koneiltaan. Virheiden määrä vähenee dramaattisesti ja periteisesti kaksi päätä ajattelee paremmin kuin yksin. Cockburn laskee tutkimuksessaan pariohjelmoinnin tuottavan 15% hitaammin koodia, mutta yksittäisen koodaajan tuottamien virheiden korjaamiseen menee 15 kertaa enemmän aikaa, kuin pari ohjelmoinnista johtuvaan koodin tuottamisen hidastumiseen. Lisäksi koodaamisen hidastumista vähentää työssä oppiminen ja viihtyminen, jotka lisääntyvät pariohjelmoinnissa. (Cockburn & Williams, 2000)

Pariohjelmoinnin lisäksi XP:n toiminta vaatii jatkuvaa integrointia (*eng. Continuous integration*). Jatkuvassa integroinnissa valmiiksi saatu tarina eli käytännössä koodi, testataan ja ajetaan yhteiseen viimeisimmän toimituksen päälle koodiin. Tätä sanotaan kääntämiseksi (*eng. Build*). Tässä koodissa on siis koko edellisen julkaisun koodi ja jo tässä iteraatiossa valmiiksi saatua koodia. Näin moduulin sisäisen testauksen lisäksi nähdään, kuinka koodi kääntyy vasten koko koodin rajapintoja. Tällä nopeutetaan virheiden löytymistä merkittävästi verrattuna siihen, että perinteisin menetelmin yritettäisiin integroida kaikkea yhtä aikaa projektin loppupuolella. Jatkuva integraatio vaatii kuitenkin toimivan infrastruktuurin ja tehokkaat serverit. Symbianin tapauksessa koko koodin kääntämiseen kului 15–17 tuntia, joten jatkuvaa integrointia oli suoritettava vain moduulitasolla. Lisäksi jatkuva integrointi vaatii niin sanottuja tynkäohjelmia (*eng. stub*). Jos arkkitehtuuri vaatii joidenkin osien vastaavan, jotta toista

osaa voidaan testata, niin ohjelmoija luo tyngän. Tämä tynkä vastaa vain saman viestin takaisin, mutta näin saadaan testattua sitä vaativia osia. Näin ollen ohjelmistoarkkitehtuurin pitää ottaa huomioon jatkuvan integroinnin käyttöönotto.

Jatkuvan itegraation seurauksena saadaan testaus automatisoitua helposti. Kun koko projekti voidaan kääntää suhteellisen nopeasti, niin voidaan jatkuvassa integroinnissa kääntää koko koodi aina yhden valmistuvan komponentin syöttämisen jälkeen. Tällöin kääntämisen perään toteutetaan automaattinen testaus. Automaattisen testauksen tulokset lähetetään suoraan komponentin valmistajalle. Näin palaute on välitön ja jos testaus menee virheettä läpi, voi ohjelmoija jatkaa seuraavaan tehtävään. Tämä on ylivertainen verrattuna perinteisten menetelmien hitaaseen testaussykliin.

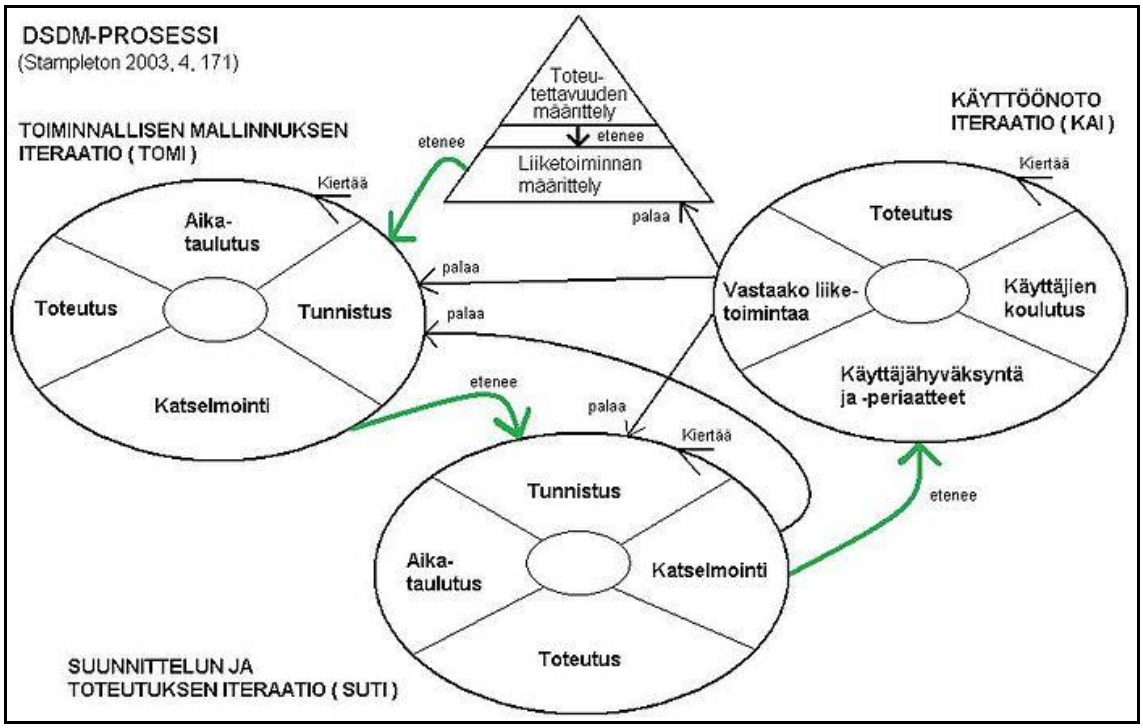
Kun tuote on asiakkaan mielestä valmis, voidaan siirtyä projektin lopetusvaiheeseen. Ensin tuotteistamisvaiheessa varmistetaan lisätyllä testauksella tuotteen laatu ja asiakkaan tyytyväisyys tuotteeseen. Tässä vaiheessa lopulliseen koodiin voidaan vielä lisätä ominaisuuksia. Seuraavaksi on ylläpitovaihe. Tämä vaihe voi sisältää vielä jotain uutta koodia, mutta tässä vaiheessa varmistetaan tuotteen toimivuus ja ylläpitoon voidaan kouluttaa uusia ihmisiä. Ketterien menetelmien tapa jättää dokumentointi minimiin voidaan täydentää tässä. Lisäksi dokumentointi voidaan kirjoittaa käyttäjien näkökulmasta. Lopetusvaiheessa asiakas on tyytyväinen tuotteeseen ja siihen ei lisätä tai tehdä enää muutoksia. Lopullinen dokumentaatio kirjoitetaan tässä vaiheessa. Lopetusvaiheessa varmistetaan myös koko toimitettavan laitteiston ja ohjelmiston luotettavuus. XP antaa siis suorat toimintaohjeet tiimin päivittäiseen tapaan tehdä töitä. Se sisältää myös yksityiskohtaiset ohjeet siitä miten työparityöskentely voidaan toteuttaa tehokkaimmin. XP on myös otettava kokonaisena käyttöön tiimissä ja sen ulkopuolella, toisin kuin scrum, joka on joustavampi.

3.6 DSDM (Dynamic Systems Development Method)

DSDM (dynaaminen järjestelmänkehitysmenetelmä) luetaan ketteriin menetelmiin ja vaikka se ei suoraan liity esimerkiksi varsinaisen koodia tuottavan insinöörin päivärytmiin. DSDM on otettava käyttöön jokaisella organisaation tasolla, jotta se toimisi hyvin. Tämän tutkimuksen kannalta DSDM ei ole kaikilta osin kiinnostava, koska DSDM ei huomioi tarvitaanko lopullisen tuotteen valmistamiseen laitteistoa vai ei. Periaatteessa DSDM tarvitsee vain tuotteen, joka markkinoidaan yleensä massamarkkinoille, sillä sen keskiössä on käyttäjältä saatava palaute. DSDM on tässä

tutkimuksessa kiinnostava niiltä osin, jotka liittyvät palautteesta tuleviin laitteiston tai ohjelmiston muutoksiin. Kuinka nopeasti voidaan reagoida esimerkiksi markkinointitutkimusten tuloksiin? Nokialla tämän olisi voinut tarkoittaa esimerkiksi sitä, että kuinka kauan kestää saada simpukkamallinen puhelin kauppoihin, kun tieto niiden trendikkyydestä saapui asiasta päättävien ihmisten korviin. Seuraavassa käydään lyhyesti läpi prosessin periaatteet. (Stapleton, 1997)

Ylemmältä tasolta organisaatiota katsottuna DSDM jakaa prosessin neljään erilliseen lohkoon. Näistä kolme ovat iteratiivisia jatkuvasti toimivia lohkoja. Ne ovat: Toiminnallisen mallinnuksen iteraatio (TOMI), Käyttöönottoiteraatio (KAI) ja suunnittelu ja toteutuksen iteraatio (SUTI). Lisäksi mukana on ei-iteratiivinen toteutettavuuden määrittely (TOMA). Tämä on kuvassa 6 ylhäällä kolmiona. Tällä mallilla on myös toinen nimi: Kolme pizzaa ja juusto.



Kuva 6. DSDM-Prosessi (Stampleton 2003)

TOMA toteutetaan aina ensimmäisenä, kun projekti aloitetaan. Tässä päätetään perusasiat eli voidaanko projekti toteuttaa, onko sillä markkinat ja voiko siihen käyttää DSDM-prosessia. Tämä vaihe kestää viikosta kahteen. Tämän tuloksena saadaan alustava suunnitelma lopputuotteesta, jos projektille on lisäarvoa prototyypistä, se

luodaan tässä yhteydessä. Toteutettavuuden määrittelyn jälkeen siirrytään liiketoiminnan määrittelyyn (LIMA). Tässä vaiheessa luodaan toimintaedellytykset koko projektille. Kaikki suunnitelmat toteutetaan tässä vaiheessa: Liiketoiminta-alue, järjestelmäarkkitehtuuri ja kehityssuunnitelma. Taulukossa 2 on esitetty tarkemmin eri suunnitelmien sisältöjä.

Taulukko 2. Liiketoiminta suunnitelman sisältö.

Liiketoiminta suunnitelman sisältö	Merkitys	Huomio
Liiketoiminta-alueen määrittely	Kuvaus tuotteen toteuttamasta automaatiosta	-
Järjestelmän arkkitehtuurin määrittely	Kuvaa toteutettavan järjestelmän arkkitehtuurin ja sen toteuttavan ympäristön.	-
Kehityssuunnitelman määrittely	Kuvaa sekä toiminnallisen mallin iteraation toteutuksen, että suunnittelun ja toteutuksen iteraation.	Tämä sisältää myös kuvaukset protoilusta, testauksesta ja kokoonpanon toteuttamisesta.

Toiminnallisen iteraation kierroksien aikana syvennetään ja täydennetään liiketoiminnan määrittelyssä syntynyt dokumentti ja tietosisältö. Tämä tarkoittaa tarpeellisten yksityiskohtien lisäämistä. Lisäksi koko järjestelmän huolellinen analyysi, malli ja toiminnallisuuden toteuttavat komponentit varmistetaan tässä vaiheessa. Ei-toiminnallisia vaatimuksia lisätään dokumentaatioon. Tämän iteraation lopputuloksena saadaan toiminnallisuudet ja niiden priorisoinnit. Käyttöönottosuunnitelmat luodaan myös TOMI:ssa. Mahdollisen prototyypin käyttäjäkokemukset ovat keskeisimpiä tuloksia tästä iteraatiosta. Suunnittelun ja toteutuksen iteraatiossa toteutetaan koko ohjelmisto ja saadaan aikaan testattu ohjelmisto. Käyttöönoton iteraatiossa ohjelmisto toimitetaan ja asennetaan sen lopulliseen sijoituspaikkaan. Tässä vaiheessa, jos ohjelmisto toimitetaan kerralla, ei tarvita useampia iteraatioita. Tämä vaihe on ohi ja voidaan siirtyä loppuvaiheeseen, jos kysymykseen “Vastaako järjestelmä liiketoiminnan vaatimuksia?” vastataan myöntävästi. (Stampleton, 2003)

Tämän työn kannalta olennaista on havaita, että DSDM ei suunnittele kaikkea kerralla, mutta jokaisen eri iteraation jälkeen suunnitelmat ovat täydellisemmät. Näin alun virheet korjautuvat itsestään, mitä enemmän liiketoiminnan määritelmästä saadaan

selville. Periaatteessa eri pizzojen eli iteraatiosilmukoiden järjestys on aina sama, mutta liikettä niiden välillä tapahtuu kokoajan. Painopiste eri iteraatoiden välillä vaihtelee projektin vaiheesta riippuen. Lisäksi seuraavat periaatteet ajavat projektia eteenpäin:

- Projekti toteuttaa vain ne ominaisuudet, jotka liiketoiminta edellyttää.
- Koodaajat tai suunnittelijat, jotka tavoittelevat täydellisyyttä ovat haitaksi projektin valmistumiselle määräajassa. Projektin pääpainopiste on sanoilla "Riittävän hyvä"

DSDM käyttää vaatimusten priorisointeja neljään eri luokkaan. Tämä eroaa mm. scrummista ja XP:stä. Näitä priorisointeja ovat: välttämätön, tarpeellinen, toivottava ja unelma. Luonnollisesti tärkeimmät toiminnot systeemin toiminnan kannalta ovat välttämättömiä, joita ovat myös ne liiketoiminnan kannalta tarvittavat ydintoiminnot. Tarpeelliset ovat niitä, jotka esimerkiksi parantavat suunniteltavan systeemin käytettävyyttä, mutta eivät ole pakollisia. Toivottava on lähinnä siksi, että niitä voidaan tehdä, jos käytettävässä aikaikkunassa saadaankin kaikki pakolliset tehtyä. Nämä ovat myös puskurina, jos haluttuja välttämättömiä toimintoja onkin mitoitettu liikaa.

DSDM sisältää rooleja kuten kaikki ketterät menetelmät. Nämä roolit on esitetty taulukossa 3. (Stampleton, 2003).

Taulukko 3. DSDM roolit.

Rooli	Kuvaus	Huomio
Visioonäärikäyttäjä	Visionääri käyttäjä vastaa kokonaisuudesta ja erityisesti kaikkien tärkeimpien ominaisuuksien mukanaolosta.	Vastaa laadusta kokonaisuudessaan.
Suurlähettiläskäyttäjä	Vastaa liiketoiminnan vaatimusten kirjaamisesta ja tarvittaessa hankkii lisätietoja.	
Tekninen yhdyshenkilö	Vastaa teknisestä toteuttamisesta ja arkkitehtuurisista ratkaisuista. Hän vastaa laadusta.	Huolehtii teknisellä tasolla laadusta.
Testaaja	Suorittaa testauksen ja toimittaa tulokset tekniselle yhdyshenkilölle.	
Kehittäjä	Koodaaja, suunnittelija ja analyytikko	

DSDM pohjaa vahvasti koulutukseen, jotta työntekijä voisi toimia missään roolissa tai ylipäätään DSDM-projektissa, on hänen osallistuttava suulliseen kokeeseen ja osattava

tarvittava terminologia. Lisäksi prosessien tunteminen myös muiden osalta tulee osata. DSDM onkin menetelmä, joka pitää ottaa kokonaan käyttöön tai ei ollenkaan.

3.7 Sulautettujen järjestelmien erityisvaatimusten vaikutus ohjelmistotuotannon menetelmiin

Erilaisten ohjelmistotuotannon menetelmien vertailu on hankalaa, koska sulautettujen järjestelmien kirjo on niin laaja. Seuraavassa kappaleessa pohditaan erilaisin esimerkein mitkä ovat eri menetelmien hyvät ja huonot puolet. Tämän lisäksi tutkitaan sulautettujen järjestelmien erityispiirteiden vaikutukset eri menetelmien kannalta.

3.7.1 Vesiputousmalli

Vesiputousmalli toimii parhaiten, kun projektin alkuperäinen suunnitelma pystytään pitämään samana koko projektin ajan. Vesiputousmallin hyvyyttä on vaikea mitata muulla kuin sillä, että soveltuuko se vai ei. Tosin kun kaikki vesiputousmallin jälkeen tulleet menetelmät, vesiputous ei reagoi muutoksiin, jotka poikkeavat alkuperäisestä suunnitelmasta. Vesiputousmallissa on mahdollisuus suorittaa edes takaista liikettä eri projektin vaiheiden välillä, kuten integrointi ja koodaus sekä mahdollisuus protoiluun. (Boehn, 1988) Vesiputousmallia ei ole kuitenkaan tehty hallitsemaan muutoksia. Tätä ei välttämättä voi pitää negatiivisena asiana, sillä esimerkiksi sulautettujen järjestelmien laitteiston vaatimukset osoittavat, että luotettavuus on pääasia. Tätä voidaan pitää siis hyvänä asiana, jos vaatimukset tai suunnitelmat eivät muutu prosessin aikana. Tämä tosin tarkoittaa sitä, että vaatimukset ovat täysin oikeita eikä liitännäiset hankkeilla olevaan laitteeseen muutu. Jos jompikumpi näistä edellä mainituista uhista realisoituu, niin vesiputousmalli on vaikeuksissa. Virheet laitteistossa, kuten esimerkiksi prosessorissa, voivat tulla esille vasta projektin loppuvaiheessa, milloin vasta koodia aletaan ajaa oikeassa prosessorissa simulaation sijaan. Virhe saattaa olla niin vakava, että koodia tai jopa arkkitehtuuria joudutaan muuttamaan. Vesiputousmalli epäonnistuu tai ainakin projekti myöhästyy pahasti. Vesiputousmalli palaa suunnitteluun ja virheenkorjaus käy lävitse koko ketjun. Virheen korjaamista pahempi ongelma on, jos laitteen vaatimukset muuttuvat. Jos näin käy, niin projektiin on valittu väärä tekniikka.

Hyvänä asiana voidaan pitää vesiputousmallin tehokkuutta vaatimusten ja ongelmien ollessa minimissään. Projektin ollessa pieni ja ryhmän ollessa tehokas toistensa tunteva yksikkö, on vesiputousmallilla mahdollisuudet tuottaa lopullinen tuote nopeasti ja

tehokkaasti. Laitteiston vaatimuksien kannalta tällainen projekti onnistuu parhaiten, jos käytetään laitekantaa, joka on jo ennestään tuttu. Ei oteta minkäänlaisia riskejä laitteistoa hankittaessa, eli hieman ylimitoitetaan vaatimuksiin nähden tehoja ja muita ominaisuuksia. Tämä voi tarkoittaa pientä kustannusten nousua, mutta projektin ollessa pieni, projektin kustannukset laskevat mahdollisen nopean kehitystyön seurauksena. Laitteiston virheiden mahdollisuutta voidaan myös minimoida käyttökohteiden rajoittamisella. Rajataan lopullisen laitteen toimintaympäristöt sellaisiksi, että ne ovat mahdollisimman optimaaliset. Laitteiden ollessa usein akkukäyttöisiä ja mukana kulkevia, ei tämä ole aina edes mahdollista. Henkilöstönvaihdokset eivät aiheuta suuria ongelmia, sillä dokumentointi on hoidettu kattavasti. Tämä toimii yleisesti hyvin vesiputousmallissa.

Vesiputousmalli sekä ohjelmistotuotannossa että sulautettujen järjestelmien projektissa toimii hyvin valtavan kokoisissa, erittäin paljon rakennetta vaativissa projekteissa. Suuret projektit vaativat alusta asti hyvät suunnitelmat ja erittäin paljon valvontaa (Kettunen & Laanti 2004).

Testaus on erittäin ongelmallinen osuus vesiputousmallissa. Pienet virheet, kuten semanttisetja kirjoitusvirheet, voidaan korjata, mutta lopullisen tuotteen testauksessa löytyvät suuremmat virheet ovat vaikeita korjata. Toisaalta, jos virhe saadaan korjattua, korjauksen saattaminen varsinaiseen koodiin on helppoa, sillä koko projekti on pysähdyksissä kunnes korjaus saadaan aikaiseksi. Myöhemmin tulevissa metodeissa ja malleissa, virheen löytyessä testauksessa, on jo tämän virheen päälle rakennettu uutta ja riippuvuudet joudutaan testaamaan. Virheiden korjaaminen aiheuttaa siis vähemmän piileviä mahdollisia ongelmia, mutta pääasiassa enemmän kustannuksia vesiputousmallissa. Voidaankin todeta, että vesiputousmallin heikkoudet ovat suuremmat kuin sen edut, mikä on totta erityisesti ohjelmiston kannalta. Laitteistoläheisessä projektissa on helppo kuvitella pienehkö projekti, joka on erittäin hyvin suunniteltu ja toteutetaan kurinalaisesti. Esimerkkejä ovat laitteet, joissa ei ole juurikaan tarvetta käyttöliittymälle ja niiden toiminta on hyvin ennakoitavissa. (Boehm, 1988) Ensimmäisiä sulautettuja järjestelmiä olivatkin mannerten väliset ohjukset kylmänsodan aikaan. (Kosunen, 2003) Tosin vesiputousmallia käytettiin tällöin myös siitä syystä, että silloin se oli kaikista kehittynein saatavilla oleva malli. Vesiputousmalli kattaa koko laitteiston elinkaaren. Koko elinkaari voidaan suunnitella alusta asti.

3.7.2 Inkrementaalinen malli

Inkrementaalinen malli on realistinen malli vesiputousmallista. Ihmiset tekevät virheitä kaikkialla projektissa: suunnittelussa, vaatimuksissa ja toteutuksessa. Näihin varautuminen on hallittava, jollain tavalla. Inkrementaalinen malli toimii pienemmissä osissa, jolloin yhteistyö laitteiston ja ohjelmiston välillä on helpompaa. Laitteiston suunnittelussa voidaan toimittaa jokaisen inkrementin jälkeen uusi ohjelmisto testaukseen. Tämä ei tarkoita sitä, että ohjelmisto ja laitteisto olisivat aina saman inkrementtikierroksen tulos, mutta yhteistyölle ja virheiden löytymisille on paljon enemmän mahdollisuuksia kuin vesiputous mallissa. Nokialla tämä aiheutti mielenkiintoisia ongelmia. Ohjelmiston integrointi tarvitsi aina uusimmat firmware-ajurit jokaiselle integrointikierrokselle. Integrointiin oli suunniteltu kaksi viikkoa, joten firmware piti toimittaa kahden viikon pituisen iteraation päätteeksi. Lisäksi firmware testattiin niin, että edellisen iteraation integroitu osuus onnistui käynnistymään uuden firmwaren päällä. Näin saattoi tulla ongelmia, jossa integrointi ei saanut toimitettua omaa iteraatiotaan ulos ja firmware ei saanut toimitettua omaa tulostaan eteenpäin, koska heillä ei ollut uusinta integroitua kerrosta testaukseen. Kahden viikon syklit olivat lisäksi liian nopeita firmware-kehityksessä. Inkrementaalinen malli ei ota huomioon, sitä että laitteiston ja ohjelmiston inkrementit saattavat olla eripituisia. Ongelma ratkeaa sillä, jos inkrementit ovat esimerkiksi kahden ja kolmen viikon pituisia.

Inkrementaalinen malli on parhaimmillaan, jos kyseessä on projekti, jossa on suuri vaatimus dokumentointiin ja se voidaan rakentaa eri osakokonaisuuksista. Tämä tarkoittaa ennakoitavia toimituksia joko sisäisesti tai firman ulkopuolelle. Tämä malli sopii hyvin laitteistoläheisiin projekteihin, missä on suunnitelmat ja käyttötarkoitus selkeästi esillä. Päällekkäisen dokumentoinnin ja lisääntyvän tiedottamisen vaatimukset laskevat tehokkuutta ongelmattomissa tapauksissa verrattuna vesiputousmalliin, mutta ihmisten tehdessä virheitä ja vaatimusten muuttuessa on malli huomattavasti realistisempi kuin vesiputousmalli.

3.7.3 Evoluutiomalli

Evoluutiomallin käyttäminen antaa taas uusia työkaluja sulautettujen järjestelmien projektin käyttöön. Edellisessä mallissa Osakokonaisuuksien valmistumiset olivat vahvasti yhteydessä toisiinsa aikataulullisesti. Evoluutiomallissa, jokaisen iteraation alussa tehdään riskianalyysi ja mahdollinen tulevaisuuden suunnitelmien analysointi.

Parhaimmillaan tämä tarkoittaa esimerkiksi sitä, että jos laitteiston kehitys on huomattavasti edellä ohjelmistoa, voivat he jatkaa sen rakentamista ohjelmiston viivytyksistä huolimatta. Tämä mahdollistaa myös sen, että esimerkissä käytetty laitteisto voi muuttaa suunnitelmiaan yhdessä ohjelmistojen kanssa niin, että aluksi suunnitelmasta pudotettu ominaisuus voidaan ottaa tavoitteeksi. Tämä on mahdollista, myös vanhemmissa metodeissa, mutta sillä luodaan ongelmia projektin suunnitelmiin ja aikatauluihin. Tämä lisääntynyt joustavuus mahdollistaa tuotteen kehittämisen myös pitkissä projekteissa. Esimerkiksi matkapuhelinten valmistus ideasta tuotteeksi voi kestää kahdesta kolmeen vuotta. Matkapuhelimen tapauksessa kyseessä on trendikäs tuote ja projektin on kyettävä muuttumaan niiden mukaan. Esimerkiksi matkapuhelimen muotoilun tai laitteiston tehon vaatimukset vaihtuvat markkinoiden mukaan. Näin markkinat ja kysyntä pitää yllä tarpeen muutoksille. Sulautetun järjestelmän projektia ajatellen on evoluutiomalli vedenjakaja vanhan plan-driven-mallien ja uusien ketterien menetelmien välillä. Tämä esittääkin kysymyksen, onko toteutettava projekti altis muutoksille vai ei?

3.7.4 Ketterät menetelmät - Scrum

Ketteristä menetelmistä scrum on lähinnä perinteisiä menetelmiä toteuttaa ohjelmistotuotanto. Se johtuu siitä, että se ottaa kantaa vain lähinnä siihen, miten työt jaetaan ja miten reagoidaan muutoksiin. Sulautettujen järjestelmien kannalta juuri kommunikaatio kaikkien eri osapuolten välillä on erittäin tärkeää, koska ohjelmistoa ja laitteistoa kehitetään yhtä aikaa. Scrum lähtee ensisijaisesti liikkeelle siitä, että kaikki työ perustuu asiakkaan luomiin käyttäjäkertomuksiin ja niistä tiimin itsensä luomiin tehtäviin. Tämä tarkoittaa sitä, että kaksi erilaisissa tehtävissä olevaa tiimiä voivat katselmoida toistensa työn etenemistä. Näin tietoa siitä, mitä eri ohjelmointitasoilla tehdään, voidaan levittää kaikkien projektissa olevien tahojen tietoon. Asiakkaan jatkuva mukanaolo varmistaa lisäksi koko projektin onnistumisen kannalta kriittisimmän kriteerin eli asiakastyytyväisyyden. Sulautetun järjestelmän ollessa kyseessä on asiakkaalle pystyttävä konkreettisesti ja selkeästi näyttämään mahdottomat vaatimukset ja tämä onnistuu esimerkiksi siten, että asiakkaan vaatimus ei käänny miksikään tehtäväksi millään projektia valmistavalla tiimillä. Tämä jäljitettävyys, joka siis seuraa jokaista asiakkaan vaatimusta, varmistaa yhtälailla asiakkaan tyytyväisyyden kuin asiakkaan vaatimusten mielekkyyden seuraamisen.

Scrum voidaan ottaa käyttöön varsin pienillä muutoksilla organisaatioon, mutta jotta edellä esitetty läpinäkyvyys saavutettaisiin, on se otettava käyttöön kaikilla tasoilla organisaatiossa. Osittainen käyttöönotto voi kuitenkin helpottaa perinteisin menetelmin toimivaa organisaatiota siirtymään tähän ketterään menetelmään.

Scrumin lisäämän avoimuuden vuoksi, myös työssä jaksaminen parantuu. Avun saaminen ja ammattitaidon jakaminen ovat ensiarvoisen tärkeitä juuri nimenomaan sulautettujen järjestelmien parissa, missä eri alojen erityisosaamiset korostuvat. Samassa projektissa voi toimia tehoelektroniikan, ohjelmistotekniikan, c++ ja käytettävyyden osaajia, ja näiden yhteistyön tuloksena syntyvä tuote tarvitsee monialaista näkemystä ja sen hyväksikäyttöä. Jokapäiväiset kokoontumiset ja tehtävien selkeäkielinen asiakkaan kuvaus mahdollistavat näiden eri alojen ihmisten keskustelun erilaisista ratkaisuista.

3.7.5 Ketterät menetelmät – Extreme Programming

Extreme Programming antaa monia hyödyllisiä työkaluja sulautettujen järjestelmien käyttöön. Asiakkaan vaatimusten hallinta on samalla tasolla kuin Scrum-metodissa, mutta XP:n vaatimus matalalle organisaatiolle tuo asiakkaan lähemmäksi varsinaista koodaajaa sekä projektin hallinnan kannalta asiakkaan palautetta on helpompi saada. Näin esimerkiksi laitteen nopeuteen liittyvät kysymykset tulevat helpommin esille, sillä juuri laitteen “tarvittava nopeus” on vaikea määrittää suunnittelussa. Laitteen tarvittava nopeus on helpointa määrittää antamalla laite asiakkaan käteen ja kysyä toimiiko se tarpeeksi nopeasti. Tätä tukee XP:n tapa ajaa testaus jokaisen integraation jälkeen. Jatkuva integraatio mahdollistaa nopean tavan testata koodia sekä mahdollistaa laitteiston käytettävyyden jatkuvan tarkistamisen. Reaaliaikakäyttöjärjestelmän tapauksessa voi olla joskus vaikea mitata tai arvioida muistin kulutusta ja laitteen reagointinopeutta muuta kuin testaamalla.

Pariohjelmointia hyödyntäen on mahdollista levittää osaamista kahden eri ohjelmoijan välillä. Tämä tarkoittaa sitä, että laitteistosuunnittelija voi istua laitteistorajapintaa toteuttavan ohjelmoijan vieressä ja auttaa tätä saavuttamaan yhteiset tavoitteet. Muistia ja laitteiston nopeutta säästävät ratkaisut eivät aina tule mieleen korkeampia ohjelmointikieliä käyttävien ohjelmoijien keskuudessa. Tämä toimii myös päinvastoin, sillä perinteisesti laitteistoläheinen ohjelmointi käyttää yksinkertaisempia

ohjelmistoympäristöjä. Nokia Oy:ssä pariohjelmointia käytettiin usein tapauksissa, joissa muutoksia vaadittiin tapahtuvaksi laitteistoajureissa ja käyttöjärjestelmätasolla. Näin kävi usein, kun jokin uusi fyysinen ominaisuus tuli laitteistoon ja sen ohjaus käyttöliittymältä vaati kaikkien ohjelmistokerroksien läpäisyä.

3.7.6 Ketterät menetelmät – DSDM

DSDM:n käyttö sulautettujen järjestelmien suunnittelussa ja toteutuksessa tuo mukanaan muutamia poikkeuksellisia hyötyjä. Laitteiston ja ohjelmiston suunnittelussa käytettävä vähitellen tarkentuva malli sopii laitteiston ja ohjelmiston yhteiskehittämisen kannalta hyvin. Asiakkaan tarinat antavat hyvän yleiskuvan siitä mitä halutaan, ja yhtäaikaisesti voidaan alkaa rakentamaan laitteistoa ja ohjelmistoa. Ohjelmiston ja laitteiston tiukka suunnittelu on mahdollista, sillä alussa määritelty liiketoimintaympäristö antaa tavoitteet ja niiden prioriteetit selkeästi. Laitteiston suunnittelu on vasteen kannalta helppoa, sillä tavoitteena on tuottaa liiketoimintaympäristöön sopiva kokonaisuus mahdollisimman vähillä komponenteilla. Lisäksi tärkeä ominaisuus on DSDM:ssä se, että se panostaa metodin opettamiseen ja kouluttamiseen organisaatiossa. Kaikkien tulee opetella DSDM termistö ja tuotantotavat ja näin kaikilla on yhteiset tavoitteet. Nokia Oy:ssä työn tekemisen esteenä olivat väärinkäsitykset terminologiassa. Työn alkuperäinen tilaaja ei osannut kertoa mitä hänen vaatimansa kirjainyhdistelmä tarkoitti. Pahimmassa tapauksessa toimittaja ei osannut toimittaa komponenttejaan oikeaan paikkaan, koska hänelle ei ollut kerrottu, että tavanomaisesti alalla käytetty termi olikin vaihdettu toiseksi, koska toimituksen vastaanottamiseen suunniteltu ohjelmistokehittäjä ei tiennyt oikeaa termistöä.

3.7.8 Ketterien menetelmien soveltaminen

Kaikkien ketterien menetelmien variaatioiden vertailu on tämän tutkimuksen piirissä mahdotonta ja siksi tähän työhön on valittu kolme yleisintä menetelmää. Ne kattavat niin yksittäisen ohjelmoijan, kommunikaatioprosessit ja mahdollisen ylemmän tason projektin hallinnan. Seuraavassa esimerkki siitä, miten Intelin suunnittelutiimi otti käyttöön eri osia erilaisista menetelmistä ja sovelsi sitä sulautetun järjestelmän projektissa.

Intelin prosessoreja suunnitteleva ja kehittävä tiimi otti käyttöön ketterän menetelmän saadakseen hallintaan koodin, joka oli pääasiassa assembly-kieltä. Tiimi koodasi

firmwarea assemblyllä ja C:llä. Assembly-koodi ei ollut suuren optimoinnin takia luettavaa. Koodaajat eivät pitäneet yllä yhtenäisiä koodaustapoja. Heillä oli lisäksi ongelmia pitää suunnitelmista kiinni. Neljä kertaa vuodessa laadittava suunnitelma oli ensimmäisen viikon jälkeen mahdoton noudattaa. Tiimin tekniset taidot olivat erittäin eriytyneitä. Jäsenet eivät siis tienneet tarpeeksi toistensa töistä pystyäkseen auttamaan toisiaan tehokkaasti. Projektina olivat Intel Itanium-perheen tuotteet. Toimintaympäristö oli erittäin altis muutoksille sekä erityispiirteenä oli se, että uuden prosessorin tuleminen tehtaalta kesti kuukauden. Tämä tarkoitti sitä, että pääasiassa kaikki korjaukset tehtiin firmware-muutoksilla. Tästä seurasi se, että kun korjattu prosessori tuli tehtaalta, koodista joutui poistamaan virhettä korjannut firmware. (Greene, 2004) Jos koodin päälle on rakennettu uutta koodia, on sillä suuri vaara aiheuttaa regressiota eli testitulokset huononivat.

He ottivat ketteristä menetelmistä käyttöön Scrumin sekä Extreme Programming ja pariohjelmoinnin osittain, jonka jälkeen heidän suunnitelmansa paranivat. Kolmen kuukauden suunnitelmien sijaan he saivat suunnitella kuukauden mittaisen jakson. Tämä paransi täsmällisyyttä ja aikatauluista voitiin pitää kiinni. Sprint Planning-vaiheessa tehtävien tuoma arvo projektille ymmärrettiin paremmin, sekä ennen kesken jääneitä tai venyviä tehtäviä ei enää tullut. Lisäksi Assembly-koodi selkeni, sillä ymmärrettiin, että optimointi vain aikakriittisissä kohdissa oli arvoa lisäävää työtä. Sprintissä oli vain tehtyjä ja tekemättömiä tehtäviä. Näin tieto siitä, missä vaiheessa projekti oli menossa, saatiin välitettyä myös projektin johdolle ja ylempään johtoon. Scrum-hetki paransi tiimin henkilöiden motivaatiota ja päivittäinen tapaaminen auttoi ymmärtämään, mitä tiimin muut jäsenet tekevät. Lisäksi Scrum-hetkessä ongelmat selviävät ja kukaan ei ole yksin ongelmansa kanssa päivää pitempään. (Greene, 2004)

Itse koodi muuttui luettavammaksi kahdesta syystä. Toinen oli se, että ihmiset keskustelivat päivittäin eri ratkaisuista ja toinen taas oli pariohjelmoinnin ottaminen mukaan. Ihmiset oppivat asioita, jotka eivät olleet heidän ominta erityisalaansa. Heidän aikaisemmin laatimansa standardikoodaamistyyli tuli paremmin esiin pariohjelmoinnissa. Parityöskentely haittasi kuitenkin nopeutta silloin, kun oli kiire tai kun loppurutistus alkoi Sprintissä. (Greene, 2004)

Toinen ongelma liittyi eri teknologia-alueiden omistukseen. Tyypillisesti ihmiset pitävät jotain aluetta omanaan. Erityisesti jos heillä on siihen liittyvä vahva osaaminen.

Ketterissä menetelmissä tämä vastuu on kollektiivisesti koko tiimillä. Tästä tiimi pääsi yli vasta, kun ketterä menetelmä oli ollut käytössä jo pitempään. (Greene, 2004)

Ketterissä menetelmissä vaatimusten hallinta on erityisen hyvin toimiva asia. Product Backlog ja erillinen Sprint Backlog antavat näkyvyyttä projektin tällä hetkellä meneillään olevasta työtilanteesta sekä projektin johdolle näkyvyyttä tulevaisuuteen. Lisäksi Sprintin aikana ei tule uusia vaatimuksia, vaan tiimi saa rauhassa toteuttaa annettuja tehtäviä. Sulautettujen järjestelmien kannalta laitteiston vaatimusten jatkuva kehittäminen on vaikeaa, mutta ketterässä menetelmässä voidaan esimerkiksi Sprintin Backlogiin kirjata erilaiset vaatimukset, jotka sitten myöhemmin korjataan varsinaiseen prosessoriin. Kun korjaus on tehty, lisätään korjauksen poistaminen firwaresta Product Backlogiin, jolloin se tulee tehtäväksi seuraavaan Sprinttiin. (Greene, 2004)

Produckt backlog antaa näkyvyyden tiimin ulkopuolisille projektin jäsenille. Tämä tarkoittaa sitä, että näitä tulevia ominaisuuksia voi ketjuttaa, jos ketteriä menetelmiä käytetään ympärillä olevissa tiimeissä. Product Ownerit voivat kerääntyä ja muodostaa oman Sprint Backloginsa. Tällä keinolla voidaan myös yhdistää kuilu varsinaisen laitteiston rakentajien, firmwaren ja ohjelmistotuotannon välille. Kriittisellä polulla olevien toimintojen varmistaminen voidaan tehdä niin, että laite ja ohjelmistotuotannon vaatimukset näkyvät eri toimijoille. Näin eri osaprojektit voivat toteuttaa kaikki tarvittavat tehtävät oikeassa järjestyksessä.

4 Johtopäätökset

Projektin koko

Ohjelmistotuotannon menetelmistä on tehty monia tutkimuksia ja kirjallisuutta on löydettävissä erityisesti internet-aikakaudella runsaasti. Laitteistoläheisen ohjelmoinnin eri menetelmistä tätä materiaalia on saatavilla vähemmän, mutta silti runsaasti. Erillisten menetelmien vertaaminen on vaikeaa yleensä suurissa organisaatioissa ja jopa tarpeetonta, sillä laitteiston ja ohjelmiston kehittämiset voivat olla niin erotetut, ettei voida edes kunnolla puhua sulautumisesta. Se voidaan kuitenkin todeta kirjallisuuden perusteella, että mitä suurempi suunniteltava projekti on laitteistoltaan ja ohjelmistoltaan, sitä todennäköisempää on se, että projekti toteutetaan suunnitelmavetoisella metodilla. Sitä vastoin pienemmät projektit kannattaa toteuttaa ketterämmillä menetelmillä. Erityistä huomiota on osoitettava projektin vaatimusten hallinnalle. Yleisesti ketterät menetelmät vaativat kehittävän tiimin olevan sijoitettuina samaan rakennukseen ja mahdollisesti samaan avokonttoriin.

Laitteiston vaatimukset

Jos laitteiston vaatimukset vaihtuvat useasti, on projektin toteutuksen mallin pystyttävä reagoimaan siihen ripeästi ja hallitusti. Tämä on luonnollisesti vaikeinta plan-driven-malleissa. Laitteiston lisävaatimukset antavat tiettyä ehdottomuutta projektille. Tämä tarkoittaa mallia valitessa sitä, että suunnitteluvaiheessa joudutaan tekemään tärkeitä päätöksiä, jotka vaikuttavat koko projektin loppuun asti. Jos kriittisiin vaatimuksiin projektissa kohdistuu muutospaineita, on koko projekti lopetettava ja aloitettava alusta. Näiden kriittisten laitteistovaatimusten olemassaolo painottavat plan-driven-mallien käyttöä. Lisäksi on olemassa sellaisia vaatimuksia laitteistolle, joita ei voida ohittaa, kuten esimerkiksi sähköturvallisuus tai laitteen maksimipaino. DSDM voi tuoda tähän ongelmaan ratkaisun, mutta se vaatii koko organisaation täydellistä sitoutumista metodiin. DSDM painottaa kuitenkin alun huolellista suunnittelua, mutta kuitenkin sillä kriittisellä erolla, että asiakkaan uusia vaatimuksia voidaan käsitellä. Voidaankin sanoa kirjallisuuden perusteella ja omiin kokemuksiin perustuen, mitä suurempi paino laitteistolla on projektissa, sitä staattisempi tulisi koko projektin vaatimusten ja henkilöstön vaihtuvuus.

Laitteiston ja ohjelmiston suhde

Jos sulautettujen järjestelmien projektin painopiste on enemmän ohjelmistotuotannossa, eli esimerkiksi laitteisto on ostettu ulkopuolelta, toimivat perinteiset ohjelmistotuotannon menetelmät paremmin kuin ketterät menetelmät. Tämän jälkeen mallin valinta on helppoa, mutta silti tässä projektissa on otettava huomioon laitteiston rajat sekä mahdolliset fyysisten viiveiden olemassa olo. Näin voidaan sanoa, että riippumatta siitä toteuttaako projekti ollenkaan laitesuunnittelua, on projektin otettava huomioon sen ohjelmiston toteuttavan laitteiston vaatimukset. Nämä rajoitukset eivät kuitenkaan enää muutu juurikaan ohjelmiston tuotantovaiheessa, joten ne eivät luo juuri ongelmia projektin toteuttavan mallin valinnassa. Jos laitteiston vaatimukset ja kustannukset kattavat melkein koko projektin budjetin ja tarvittava ohjelmisto on vähäinen, voidaan projekti jopa toteuttaa vesiputousmallilla. Tämä ei kuitenkaan tarkoita sitä, että selvittäisiin yhdellä iteraatiolla, vaan vesiputouksia voidaan tehdä monia peräkkäin. Tällöin inkrementaalinen tai evoluutiomalli toimii vesiputousta paremmin. Se milloin ketterä menetelmä toimii parhaiten laitteiston ollessa painopisteenä on, kun laitteisto rakennetaan esimerkiksi niin, että se on vahvasti modulaarinen. Näin saadaan valmiiksi vähässä ajassa eri komponentteja toimitettavaksi ja niiden avulla saadaan monta eri mahdollista lopputulosta riippuen asiakkaan tarpeista. Esimerkkinä voisi toimia tiimi, joka rakentaa erilaisiin teollisuuden tarpeisiin toimintoja käyttäen Siemens:n Simatic S7-1200 -logiikkaa.

Laitteistoläheinen, välitaso- ja sovellusten ohjelmointi

Erilaiset tasot eivät sinänsä aiheuta ongelmia projektin mallin valinnassa, mutta näiden eri tasojen tiedonvälittäminen on todella kriittistä projektin onnistumisen kannalta. Tämä voidaan toteuttaa kaikissa eri menetelmissä menestyksekkäästi. Kommunikoinnin laatu ja sen vastaanottaminen on ketterissä menetelmissä kaikista parhaiten onnistunutta, mutta tämä on erityisen pakollista juuri ketterässä menetelmässä sen muuttuvien vaatimusten vuoksi. Lisäksi erilaiset Extreme Programin mukana tuomat hyödyt, kuten pariohjelmointi voivat supistaa laitteisto-ohjelmoijan ja laitteistoriippumattoman ohjelmoijan välistä kuilua. Voidaankin sanoa, että kommunikaatio hyvin toteutuessaan toimii kaikissa eri metodeissa teoriassa yhtä hyvin. Käytännössä ketterän menetelmän päivittäiset Scrum-kokoontumiset levittävät tietoa erilaisista tapahtumista ja kun ihmiset ovat kerääntyneet kaikki koolle, voi sellaisiakin

asioita tulla esille, mitä ei osattu edes ennakoida. Tätä ei voida kuitenkaan pitää erilaisten metodien ja mallien valinnan kriteerinä, sillä esimerkiksi vesiputousmallissa ei ole edes tarpeen tietää, miten laitteistoläheisen ohjelmoinnin ongelmat ovat ratkaistu, sillä ne voidaan lukea projekti suunnitelmasta.

Asiakas

Kun projektia suunnitellaan ja erityisesti kun sen vaatimuksia mietitään, on usein asiakas tai tilaaja mukana. Eri mallit eroavat siitä miten asiakas tai tilaaja on mukana suunnittelu dokumentaation tekemisen jälkeen. Yleisesti tunnettu tosiasia projektin alkusuunnittelussa on se, että asiakas ei aina edes tiedä mitä hän haluaa eikä ainakaan sitä, miten hän sen haluaa. Näin käytännössä kaikki projektit kokevat muutoksia. Miten asiakas pidetään mukana projektissa, riippuu täysin siitä millainen malli valitaan. Sulautettujen järjestelmien tapauksessa asiakkaan vaatimukset voivat poiketa rajusti siitä, mitä voidaan toimittaa. Asiakaan vaatimukset voivat olla fyysisesti mahdottomia. Tätä erikoispiirrettä ei ole puhtaassa ohjelmistotuotannon projektissa. Sulautettujen järjestelmien suunnittelussa asiakkaalle on pystyttävä sanomaan: Ei. Ohjelmistotuotannon kannalta kaikki projektin lisävaatimukset voidaan toteuttaa, mutta ne vaativat lisää aikaa ja rahaa. Sulautettujen järjestelmien tapauksessa laitteen fyysinen olemus antaa ehdottomia rajoituksia, kuten näppäinten koko tai se, että sen tulee pysyä tiettyjen lämpötilanormien sisällä. Näin asiakkaan kannalta plan-driven toteutukset voivat luoda paremman ja realistisen kuvan lopullisesta tuotteesta.

Yhteenvetona voidaan todeta kuitenkin se, että kaikilla perinteisesti ohjelmistotuotannon menetelmillä on vaikeuksia suoriutua laitteiston ja ohjelmiston yhteissuunnittelusta. Ohjelmistotuotannon menetelmät näkevät laitteiston enemmän rajoitteena ja haasteena kuin mahdollisuutena. Tämä käy ilmi erilaisista yrityksistä toteuttaa niin sanottuja yleisiä ratkaisuja eli platform ajattelua. Nokia Oy jätti monta innovaatiota käyttämättä omissa suunnitelmissaan vain saavuttaakseen yhtenevän koodikannan S60 tuoteperheessä. Tämä lähestyminen heikentää käytettävissä olevan laitteiston tehokasta käyttöä sekä aiheuttaa tuntemattoman määrän erilaisia ongelmia laitteiston ajaessa sille optimoimatonta ohjelmistoa. Lisäksi rajapintoja tulee lisää ja dokumentaatio monimutkaistuu.

Ketterät menetelmät ja etenkin Extreme Programming antaa lisämahdollisuuksia tuottaa laadukkaampia ohjelmistoja, koska projektit ovat läpinäkyvämpiä ja niiden tuottamat

ohjelmistot kustannustehokkaampia, kuin esimerkiksi evoluutiomalli tai vesiputousmalli. Varsinaisen koodin tehokkuus, jos sitä ei erikseen ole määritelty tavoitteeksi projektin alussa, voi olla heikompaa, sillä esimerkiksi DSDM pitää haittana täydellisyyden tavoittelemista. Intel esimerkki kuitenkin kuvaa hyvin sen, miten koodin ei tarvitse olla tehokasta kaikkialla, vaan siellä missä sen koetaan tuovan lisäarvoa. Tuotteen mahdolliset lisäarvot erilaisista insinöörien spontaaneista innovaatioista jäävät saavuttamatta, mutta tuotteen lopullinen aikataulu ja asiakkaan tyytyväisyys saavutetaan.

Ketterien menetelmien hyödyt ovat selvät verrattuna ei-ketteriin menetelmiin, mutta tällä hetkellä ketteristä menetelmistä mikään ei suoranaisesti ota huomioon laitteiston ja ohjelmiston yhteissuunnittelua. Suunta on kuitenkin selvä. Uusia ohjelmistoja tulee erilaisten emulaatioiden ja simulaatioiden muodossa ja ne tulevat tukemaan ketterien menetelmien kaltaisia menetelmiä, jotka on nimenomaan tarkoitettu sulautetuille järjestelmille.

Lähdeluettelo

Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. (2002), ”Agile Software Development Methods – Review and Analysis,” VTT Publications, pp. 1-10.

Agilealliance.org (2013), *Manifesto for Agile Software Development*, saatavilla www.agilealliance.org, viitattu 1.1.2013

Barr, M. (2012), “Building reliable and secure embedded systems,” in *Embedded Systems Design* Apr2012, Vol. 25 Issue 3, pp. 8–10.

Cockburn, A., Williams, L. (2000), “The costs and benefits of pair programming,” in *Extreme programming examined*, pp. 223–247.

Boehm, B., W. (1988), “A spiral model of software development and enhancement,” in *Computer*, 21(5), pp. 61–72.

Greene, B. (2004), “Agile methods applied to embedded firmware development,” In *Agile Development Conference*, pp. 71–77. IEEE.

Kettunen, P., Laanti, M. (2005), “How to steer an embedded software project: tactics for selecting the software process model,” in *Information and Software Technology*, 47(9), pp. 587–608.

Thomas, A., Henzinger, T., Sifakis, J. (2007), *Proceedings of the 14th International Symposium on Formal Methods (FM)*, 2006. *Lecture Notes in Computer Science 4085*, Springer , pp. 1–15.

Jeffries, R., Andersson, A., Hendrickson, C. (2001), “Extreme Programming Installed,” pp. 172.

Jeffries, R., Lindstrom, L. (2004), “Software Development, Extreme Programming and

Agile Software Development Methodologies", pp. 41–52.

Kosunen, P. (2003), "Sulautettujen järjestelmien varhainen kehitys," *Tietojenkäsittelytieteen historia seminaari*, Helsingin Yliopisto.

Shiue, W., Chakrabarti, C. (1999), "Memory exploration for low power, embedded systems," In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference* ,pp. 140–145.

Lavagno, L., Sangiovanni-Vincentelli, A., Sentovich, E. (1999). "Models of computation for embedded system design," In *System-Level Synthesis*, pp. 45–102.

Luukko, J. (2002), "Sulautetut Prosessorijärjestelmät," Kurssimateriaali Kurssilla *08056000 Sulautetut prosessorijärjestelmät*.

Francioni, M., Kandel, A. (1988). "A software engineering tool for expert system design," in *Expert: Intelligent Systems and Their Applications*, *3*(1), pp. 33-41.

Stapleton, J. (1997). "*DSDM Dynamic Systems Development Method: the method in practice,"*

Vece, G., B., Conti, M. (2011), "Power Analysis of Embedded Systems," in *Solutions on Embedded Systems*, pp. 301–314.

Wolf, W., H. (1994), "Hardware-software co-design of embedded systems [and prolog]," in *Proceedings of the IEEE*, 82(7), pp. 967–989.