Lähteet

scale=0.7,marginratio=1:1, 1:1,ignoreall

Iteratiiviset menetelmät

Jarl-Erik Malmström

Aine
Helsingin Yliopisto
Tietojenkäsittelytieteen laitos

Helsinki, 22. maaliskuuta 2013

HELSINGIN YLIOPISTO — HELSINGFORS UNIVE

Tiedekunta — Fakultet — Faculty	
Matemaattis-luonnontieteellinen	
Tekijä — Författare — Author	
Jarl-Erik Malmström	
Työn nimi — Arbetets titel — Title	
Iteratiiviset menetelmät	
Oppiaine — Läroämne — Subject	
Tietojenkäsittelytiede	
Työn laji — Arbetets art — Level	Aika — Datum — Mont
Aine	22. maaliskuuta 2

Tiivistelmä — Referat — Abstract

Tiivistelmä.

Sisältö

1 Johdanto

Menetelmät ovat olleet ohjelmistoja tuotettaessa käytössä pitkään. Ohjelmistojen parissa työskentelevät olivat muiden alojen insinöörejä ja matemaatikkoja 1950-luvulla. Myös menetelmät olivat omaksuttu muista insinööritieteistä. Tietokoneet olivat kookkaita sekä käyttökustannukset olivat ohjelmistoja tuottavien insinöörien palkkoihin verrattuna korkeat. Koska ohjelmistojen suorittaminen tietokoneilla oli kallista, ohjelmistokehitys vaati tiukkaa suunnittelua sekä järjestelmällisiä käytäntöjä[4].

Menetelmät toivat ohjelmistokehitykseen ennakoitavuutta ja tehokkuutta. Muista insinööritieteistä vaikutteita saaneet menetelmät olivat yksityiskohtaisia prosesseja, joissa painotetaan voimakkaasti ennen toteutusta tehtävää dokumentointia, määrittelyä ja suunnittelua[5].

Vuoteen 1960 mennessä ohjelmistojen parissa toimiville ihmisille alkoi selvitä, että ohjelmistoihin liittyvät ilmiöt poikkesivat huomattavasti laitteistoihin liittyvistä ilmiöistä. Ohjelmistoja oli paljon helpompi muuttaa kuin laitteistoja eikä ohjelmistojen kopioiminen vaatinut kalliita tuotantolinjoja. Ohjelmistojen helppo muokattavuus sai monet ihmiset ja organisaatiot omaksumaan ”ohjelmoi ja korjaa” (”code and fix”) menetelmän[4].

Reaktiona 1960-luvun ”ohjelmoi ja korjaa” lähestymistapaan, laadittiin menetelmiä, joiden prosessit olivat tarkemmin organisoituja. Menetelmissä varsinaista ohjelmointia edelsi tarkka vaatimusmäärittely ja suunnitteluvaihe[4].

Ohjelmistojen hankintasopimukset, erityisesti Yhdysvaltain hallituksen kanssa, asettivat ohjelmistotuotannon menetelmille selkeät vaatimukset. Yhdysvaltain hallituksen ja puolustusministeriön vaatimien menetelmien, ohjelmistojen tuottamiseksi, tuli koostua puhtaasti peräkkäisistä prosesseista. Suunnittelua ei voitu aloittaa ennen kuin ohjelmiston vaatimukset oli täydellisesti kirjattu eikä ohjelmointia ennen suunnitelman tyhjentävää ja kriittistä tarkastelua. Yhdysvaltain hallituksen luomat standardit prosesseille aiheuttivat tulkinnan, että ohjelmistotuotannon menetelmän täytyy olla vaiheesta seuraavaan etenevä lineaarinen prosessi[4].

Winston Roycen ajatuksia, artikkelissa “Managing the Development of Large Software Systems”, pidetään vesiputousmallin (waterfall) perustana - menetelmän joka vastasi valtionhallinnon sopimusten vaatimuksiin[8].

Nykyisin vesiputousmallin mukaisia menetelmiä kritisoidaan byrokraattiksi. Menetelmien toteuttaminen ja seuraaminen vaatii paljon aikaa sekä työtä, ja tästä johtuen ohjelmistokehitysprosessi hidastuu[5].

Uusia menetelmiä on kehitetty, vesiputousmallin mukaisille, paljon suunnittelua ja dokumentaatiota vaativien menetelmien tilalle. Näitä kutsutaan yleisesti ketteriksi menetelmiksi (agile methods). Nämä uudet menetelmät ovat olleet reaktiota byrokraattisille ja raskaille menetelmille. Ketterät menetelmät pyrkivät kompromissiin, ”ohjelmoi ja korjaa”-menetelmän ja raskaan menetelmän väliltä, tarjoamalla riittävän prosessin haluttuun lopputulokseen pääsemiseksi[5].

Tässä kirjoitelmassa tarkastelemme ohjelmistokehityksen (software de-

velopment) iteratiivisia menetelmiä. Käymme läpi erilaisten ohjelmistokehityksen menetelmien (software development methodologies) historiaa sekä menetelmien heikkouksia ja vahvuuksia.

Käymme lyhyesti läpi erilaisten menetelmien lähestymistapoja ohjelmistotuotantoon. Lisäksi tarkastelemme menetelmien erityispiirteitä sekä ohjelmistotuotantoon liittyvien erilaisten riskien tunnistamista. Tutkimme myös, miten ohjelmistotuotantoprojektien riskien tunnistaminen helpottaa projektille sopivan menetelmän valinnassa.

2 Ohjelmistotuotantomenetelmät

2.1 Vesiputousmalli

Ohjelmistotuotannossa on kaksi perustavanlaatuista vaihetta: analysointivaihe ja rakennusvaihe. Nämä kaksi vaihetta riittävät ohjelmiston totetuttamiseen jos ohjelmisto on pieni ja tuotettavan ohjelmiston käyttäjät ovat itse toteuttajia. Vaiheet pitävät sisällään aidosti luovaa työtä, joka suoraan edistää tuotettavaa ohjelmistoa[10].

Suuremman ohjelmistotuotantoprojektin täytäntöönpano vaatii lisäksi muita vaiheita, jotka eivät suoraan edistä tuotettavaa ohjelmistoa ja lisäksi kasvattavat ohjelmistotuotannon kustannuksia[10].

Ohjelmistotuotannon alkuaikoina käytetty ”ohjelmoi ja korjaa”-mallin sisältää kaksi vaihetta. Ohjelmoidaan ensin ja mietitään vaatimuksia, rakennetta sekä testausta myöhemmin. Mallilla oli useita heikkouksia. Usean korjausvaiheen jälkeen ohjelmakoodi oli niin vaikeasti rakennettu, että oli hyvin kallista muuttaa koodia. Tämä korosti tarvetta suunnitteluvaiheelle ennen ohjelmointia[3].

Usein hyvin suunniteltu ohjelmisto ei vastannut käyttäjien toiveita. Joten syntyi tarve vaatimusmäärittelylle ennen suunnitteluvaihetta[3].

Ohjelmistot olivat usein kalliita korjata, koska muutoksiin ja testaamiseen oli valmistauduttu huonosti. Tämä osoitti tarpeen eri vaiheiden tunnistamiselle, sekä tarpeen huomioida testaus ja ohjelmiston muuttuminen jo hyvin varhaisessa vaiheessa[3].

1970-luvulla vesiputousmalli vaikutti suuresti lineaarisiin ohjelmistotuotannon malleihin. Vesiputousmallin lähestymistapa auttoi poistamaan monia aiemmin ohjelmistotuotantoa vaivanneita ongelmia. Vesiputousmallista tuli perusta monille teollisuuden ja Yhdysvaltain hallituksen ohjelmistohankintojen standardeille[3].

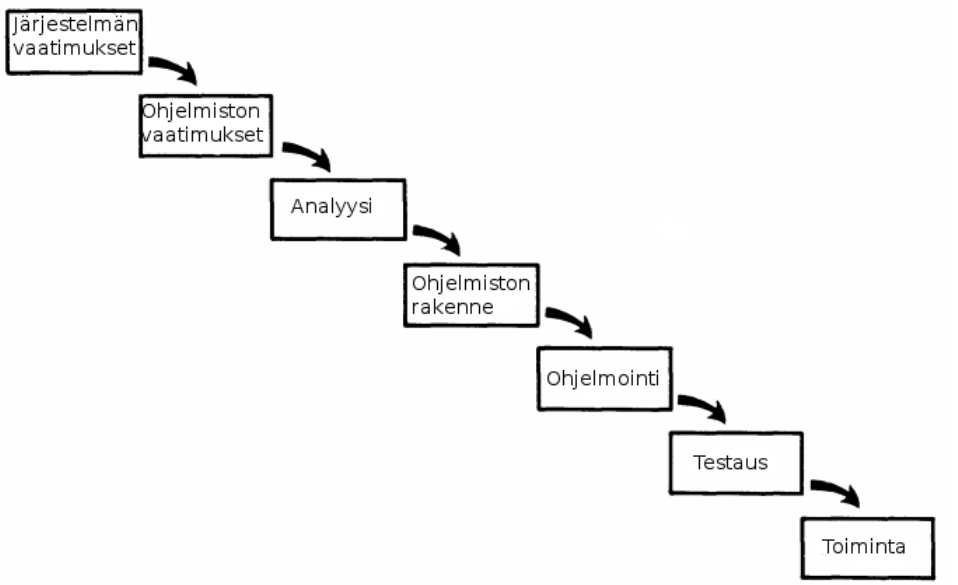
Monet pitävät virheellisesti Roycen artikkelia lineaarisen menetelmän esikuvana. Roycen artikkelin iteratiivinen ja palautteen ohjaama ohjelmistokehitys, jossa ohjelmisto toteutetaan kahdesti, on unohtunut useista menetelmän kuvauksista[8].

Tarkemmin Winston W. Roycen malli sisältää seuraavat vaiheet: järjestelmä- ja ohjelmistovaatimusmäärittely, analyysi, ohjelmistonrakenteen suunnittelu,

ohjelmointi, testaus ja ohjelmiston käyttäminen[10].

Perättäisten ohjelmistotuotantovaiheiden välillä on iteraatiota järjestelmän rakenteen tarkentuessa yksityiskohtaisemmaksi tuotannon edetessä. Iteraatioiden tarkoituksena on suunnitelman edetessä pitää muutosvauhti käsiteltävän kokoisena[10].

Kuva 1: Lineaarinen ohjelmistotuotantoprosessi



Kuvassa 1. on yleisin kuvaus lineaarisesta vesiputousmallista. Roycen kuvaama prosessi, tässä yksinkertaisessa muodossa, oli tarkoitettu vain suoraviivaisimmille projekteille. Royce oli iteratiivisten, inkrementaalisten ja kehityksellisten (evolutionary) menetelmien kannattaja. Artikkelissa "Managing the Development of Large Software Systems", 1960-1970 lukujen Yhdysvaltain hallituksen sopimusten vaatimat rajoitteet huomioiden, Royce kuvaa ohjelmistotuotannon iteratiivista prosessia[8].

Royce huomioi lineearisen ohjelmistotuotantoprosessin sisältämän huomattavan riskin. Vasta testivaiheessa, menetelmän loppupuolella, saattaa tulla esille ilmiöitä, joita ei ollut mahdollista tarkalleen analysoida aikaisemmassa vaiheessa. Ellei pieni muutos koodissa korjaa ohjelmistoa vastaamaan oletettua käytöstä, vaadittavat muutokset ohjelmiston rakenteeseen saattavat olla niin häiritseviä, että muutokset rikkovat ohjelmistolle asetettuja vaatimuksia. Tällöin joko vaatimuksia tai suunnitelmaa on muutettava. Tässä tapauksessa tuotantoprosessi on palannut alkuun ja kustannusten voidaan olettaa nousevan jopa 100%[10].

Ongelman korjaamiseksi vaatimusmäärittelyn jälkeen - ennen analyysia - on tehtävä alustava rakenteen suunnittelu. Näin ohjelmistosuunnittelija välttää talletamiseen tai aika -ja tilavaatimuuksiin liittyvät virheet. Ana-

lyysin edetessä ohjelmistosuunnittelijan on välitettävä aika- ja tilavaatimukset sekä operatiivisiset rajoitteet analyysin tekijälle[10].

Näin voidaan tunnistaa projektille varatut alimitoitetut kokonaisresurssit tai virheelliset operatiiviset vaatimukset aikaisessa vaiheessa. Vaatimukset ja alustava suunnitelma voidaan iteroida ennen lopullista suunnitelmaa, ohjelmointia ja testausvaihetta[10].

Artikkelissaan Royce painottaa kattavan dokumentoinnin tärkeyttä: On laadittava ymmärrettävä, valaiseva ja ajantasainen dokumentti, jonka jokaisen työntekijän on sisäistettävä. Vähintään yhden työntekijällä on oltava syvällinen ymmärrys koko järjestelmästä, mikä on osaltaan saavutettavissa dokumentin laadinnalla[10].

Ohjelmistosuunnittelijoiden on kommunikoitava rajapintojen(interface) suunnittelijoiden, ja johdon kanssa. Dokumentti antaa ymmärrettävän perustan rajapintojen suunnitteluun ja hallinnollisiin ratkaisuihin. Kirjallinen kuvaus pakottaa ohjelmistosuunnittelijan yksiselitteiseen ratkaisuun ja tarjoaa konkreettisen todistuksen työn valmistumisesta[10].

Hyvän dokumentoinnin todellinen arvo ilmenee tuotannossa myöhemmin testausvaiheessa, ohjelmistoa käytettäessä sekä uudelleen suunniteltaessa. Hyvän dokumentin avulla esimies voi keskittää henkilöstön ohjelmistossa ilmenneisiin virheisiin. Ilman hyvää dokumenttia, ainoastaan ohjelmistovirheen alkuperäinen tekijä kykenee analysoimaan kyseessä olevan virheen[10].

Dokumentti helpottaa ohjelmiston käyttöönottoa operatiivinen henkilöstön kanssa. Käyttöönotossa ilmenneiden mahdollisten ohjelmistovirheiden korjaamisessa selkeä dokumentti on välttämätön[10].

Dokumentoinnin jälkeen toinen ohjelmistoprojektin onnistumiseen vaikuttava tärkein tekijä on ohjelmiston alkuperäisyys. Jos kyseessä olevaa ohjelmistoa kehitetään ensimmäistä kertaa, on asiakkaalle toimitettava käyttöönotettava versio oltava toinen versio, mikäli kriittiset rakenteelliset ja operatiiviset vaatimukset on huomioitu[10].

Lyhyessä ajassa suhteessa varsinaiseen aikatauluun suunnitellaan ja rakennetaan prototyyppiversio ennen varsinaista rakennettavaa ohjelmistoa. Jos suunniteltu aikataulu on 30 kuukautta, niin pilottiversion aikataulu on esimerkiksi 10 kuukautta. Ensimmäinen versio tarjoaa aikaisen vaiheen simulaation varsinaisesta tuotteesta[10].

Testaus on projektin resursseja vaativin vaihe. Testausvaiheessa vallitsee suurin riski taloudellisesti ja ajallisesti. Loppuvaiheessa aikataulua on vähän varasuunnitelmia tai vaihtoehtoja. Alustava suunnitelma ennen analysointia ja ohjelmointia sekä prototyypin valmistaminen ovat ratkaisuja ongelmien löytämiseen ja ratkaisemiseen ennen varsinaiseen testivaiheeseen siirtymistä[10].

Testivaiheen tulee pääasiallisesti suorittaa siihen erikoistunut asiantuntija, joka ei välttämättä osallistunut varsinaiseen ohjelmointiin. Väite, että ohjelmistosuunnittelija on paras henkilö testaamaan suunnitelemansa ohjelmiston, koska ymmärtää aihealueen parhaiten, on merkki siitä että dokumentointi ei ole ollut riittävää[10].

Useimmat virheet ovat luonteeltaan ilmiselviä, jotka voidaan löytää visuaalisella tarkastelulla. Jokaisen analyysin ja ohjelmakoodin tulee tarkastaa toinen henkilö, joka ei osallistunut varsinaiseen työhön. Jokainen tietokoneohjelman looginen polku on testattava ainakin kerran[10].

Jostain syystä ohjelmiston suunnitelmaan ja aiottuun toimintaan sovelletaan laajaa tulkintaa, jopa aikasemman yhteisymmärryksen jälkeen. On tärkeää sitouttaa asiakas formaalilla tavalla mahdollisimman aikaisessa vaiheessa projektia, näin asiakkaan näkemys, harkinta ja sitoumus vahvistaa kehitystyötä[10].

Lineaaristen ohjelmistotuotantomenetelmien ongelmana oli, että tuottevien ohjelmistojen käyttäjät eivät kyenneet tyhjentävästi kertomaan mitä toiminnallisuuksia he ohjelmistolta halusivat. Asiakas saattaa muuttaa mielensä. Tai hän osaa usein sanoa, nähdessään valmiin tuotteen, mitä olisi ohjelmistolta halunnut[2]. Ongelmana ohjelmistotuotannossa on, että muutosten kustannukset kasvavat ohjelmiston elinkaaren aikana. Mitä pidemmälle projekti etenee sitä kalliimpaa muutosten tekeminen on[6].

2.2 Iteratiiviset menetelmät

Vaikka monet pitävät iteratiivisia ohjelmistotuotantomenetelmiä nykyaikaisina menetelminä, on niitä sovellettu ohjelmistokehityksessä 1950-luvulta lähtien[8].

NASA:n käytti iteratiivista ja inkrementaalista (IID) ohjelmistotuotantomentelmää Mercury-projektissa 1960-luvulla. Mercury-projekti toteutettiin puolen päivän iteraatioissa. Kehitystiimi sovelsi Extreme programming (yksi nykyisistä ketteristä menetelmistä) käytänteitä tekemällä testit ennen jokaista inkrementaatiota[8].

IBM:n FSD-yksikkö (Federal System Division) käytti 1970-luvulla laajasti ja onnistuneesti iteratiivisia ja inkrementaalisia menetelmiä kriittisissä Yhdysvaltain puolustusministeriön avaruus- ja ilmailujärjestelmien kehityksessä[8].

Vuonna 1972 miljoonan koodirivin Trident-sukellusveneen komento- ja ohjausjärjestelmän kehityksessä FSD-osasto organisoi projektin neljään noin kuuden kuukauden iteraatioon. Projektissa oli merkittävä suunnittelu- ja määrittelyvaihe sekä iteraatiot olivat nykyisen ketterän kehityksen (agile methods) suosituksia pidempiä. Vaatimusmäärittely kuitenkin kehittyi palautteen ohjaamana. Iteratiivisella ja inkrementaalisella lähestymistavalla hallittiin monimutkaisuutta sekä riskejä suuren mittakaavan ohjelmistoprojektissa. Toimittajaa uhkasi myöhästymisestä 100 000$ uhkasakko per päivä[8].

IBM:n FSD osasto kehitti Yhdysvaltain laivastolle suuren mittaluokan asejärjestelmän iteratiivisella ja inkrementaalisella menetelmällä. Neljän vuoden, 200 henkilötyövuoden ja miljoonien ohjelmarivien projekti toteutettiin 45:ssä, yhden kuukauden mittaisissa, aikarajoitetuissa (time-box) iteraatiossa. Tämä oli ensimmäisiä ohjelmistoprojekteja, joka käytti nykyisten ketterien menetelmien suosittelemia iteraatiojakson pituutta[8].

Yksi aikainen ja huomiota herättävä esimerkki iteratiivisen ja inkrementaalisen (IDD) ohjelmistotuotannon menetelmien käytöstä oli NASA:n avaruussukkulan ohjelmistojärjestelmä, minkä FSD-osasto rakensi vuosina 1977-1980. Osasto sovelsi IID:tä 17 iteraatiossa 31 kuukauden aikana, keskimäärin kahdeksan viikon iteraatioissa. Heidän motivaationaan välttää vesiputousmallia oli avaruussukkulaohjelmiston vaihtuvat vaatimukset ohjelmistokehityksen aikana[8].

Barry Boehm esittelee artikkelissaan "A Spiral Model of Software Development and Enhancement" spiraalimallin (spiral model). Mallin tarkoitus oli edistää ohjelmistotuotantoprosessia lähestymällä ohjelmistoa riskiperustaisesti. Tämä mahdollistaa mallin mukautumaan, kohdattavien riskien mukaan, sopivasti yhdistelemällä määrittelyä(specification), prototyyppien valmistusta, simulointia tai muita lähestymistapoja ohjelmiston suunnitteluun[3].

Spiraalimallissa jokainen vaihe aloitetaan tunnistamalla:

- laadittavien ohjelmisto-osien suorituskykyyn, toiminnallisuuteen sekä sopeutumiskykyyn liittyvät tavoitteet
- vaihtoehtoiset toteutustavat (ohjelmiston osto, ohjelmiston uudelleenkäyttö, vaihtoehtoiset ohjelmat)
- ohjelmiston eri vaihtoehdoille asettamien rajoitteet (rajapinnat, aikataulu, kustannukset)[3].

Kuvassa 2. Barry Boehmin kuvaama spriraalimalli on kehitetty vesiputousmallista saatujen useiden vuosien kokemuksien perusteella. Malli kuvastaa taustalla olevaa käsitettä, että jokainen vaihe sisältää saman sarjan toimenpiteitä[3].

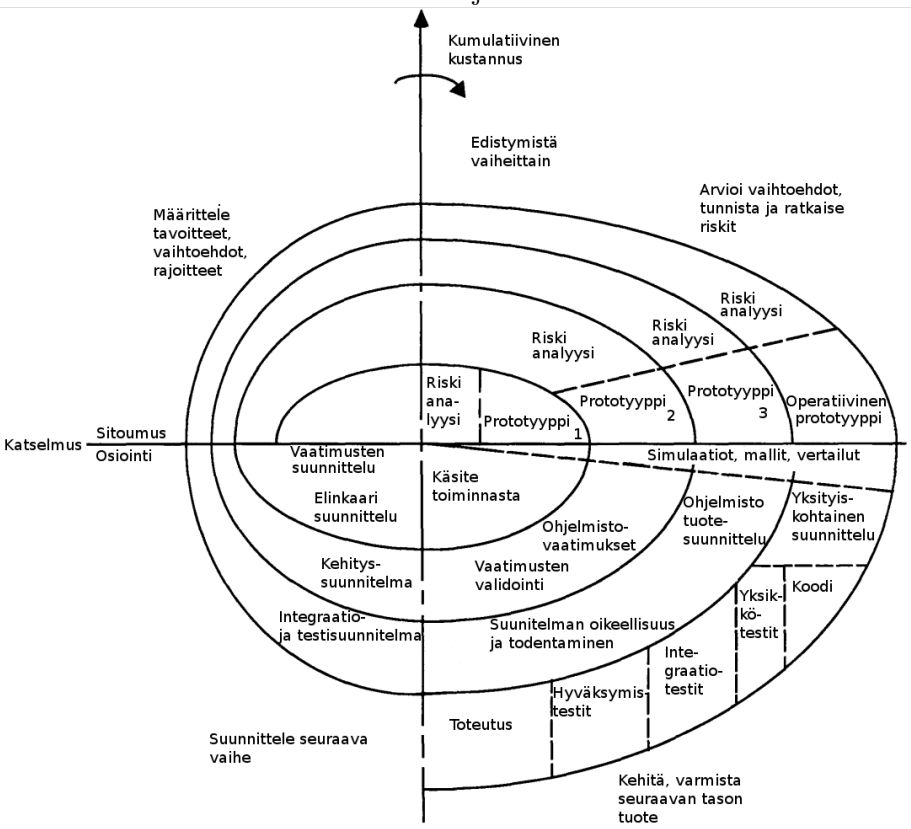
Seuraava askel on arvioida vaihtoehtoja suhteessa ohjelmiston tavoitteisiin ja rajoitteisiin. Usein tämä prosessi tunnistaa epävarmoja alueita, jotka ovat merkittäviä riskin lähteitä. Riskien löytyessä, seuraava askel pitää sisällään kustannustehokkaan strategian muotoilun riskien ratkaisemiseksi. Tähän voi liittyä prototyyppien valmistamista, simulointia, vertailuanalyysia, kyselylomakkeita, analyyttista mallinnusta, tai näiden yhdistelmiä sekä muita riskien ratkaisumenetelmiä[3].

Jos suorituskykyyn tai käyttöliittymään liittyvät riskit voimakkaasti hallitsevat ohjelman kehittämistä, seuraavassa vaiheessa kehityksellisesti (evolutionary), mahdollisimman vaivattomasti, määritellään ohjelmiston yleistä luonnetta, suunnitellaan seuraavan tason prototyyppia ja kehitetään yksityiskohtaisempaa prototyyppia riskien ratkaisemiseksi[3].

Riskinhallinta huomioiden voidaan määritellä kiinnitettävä aika ja työmäärä toiminnan suunnitteluun (planning), asetuksien hallintaan (configuration management), laadun varmistukseen(quality assurance), muodolliseen todentamiseen (formal verification) ja testaukseen[3].

Spiraalimallin tärkeä ominaisuus on, että jokainen iteraatio päätetään katselmukseen tuotteeseen liittyvän henkilöstön tai organisaation kanssa[3].

Kuva 2: Iteratiivinen ohjelmiston elinkaari



2.3 Ketterät kehitysmenetelmät

Ketterät menetelmät (agile methods) ovat saavuttaneet suosiota ohjelmistotuotannossa. Usein iteraatiivisia, inkrementaalisia sekä kehityksellisiä (evolutionary) menetelmiä pidetään modernina ohjelmistokehityksenä, mikä on korvannut vesiputousmallin. Mutta näitä menetelmiä on käytetty vuosikymmeniä[8].

Monet ohjelmistotuotantoprojektit (esimerkiksi NASA:n Mercury- ja avaruussukkula-projektit) 1970- ja 1980-luvulla käyttivät iteraatiivisia ja inkrementaalisia mentelmiä. Menetelmillä oli eroavaisuuksia iteraatioiden pituuksissa ja aikarajoitteiden käytössä (time-box). Joillakin oli merkkittävä suunnittelu- ja vaatimusmäärittelyvaihe (big design up front), jota seurasi inkrementaalinen aikarajoitettu (time-box) kehitysvaihe. Toisilla oli enemmän kehityksellisempi ja palautteen ohjaama lähestymistapa[8].

Eroavaisuuksista huolimatta, kaikilla lähestymistavoilla oli yhteistä välttää, lineaarisesta yksi vaihe kerrallaan etenevää, dokumenttiperustaista menetelmää[8].

Lineaarisesti vaiheesta toiseen etenevän ohjelmistotuotantomentelmä ei sovi erityisesti interaktiivisiin loppukäyttäjien sovelluksiin. Suunnitelmape-

rustaiset standardit pakottavat dokumentoimaan yksityiskohtaisesti heikosti ymmärretyt käyttöliittymien vaatimukset[3].

Tästä seurasi käyttökelvottoman ohjelmakoodin suunnittelua ja toteutusta. Lineaarisen ohjelmistotuotantomenetelmän vaiheet olivat tällaisille projekteille selvästi väärässä järjestyksessä. Erityisesti joillekin ohjelmistoille ei ole tarvetta yksityiskohtaiselle dokumentaatiolle ennen toteutusta[3].

Liiketoimintaympäristö muuttuu nykyisin nopeassa tahdissa. Teknologian ja liiketoiminnan vaatimusten muuttuessa tehdyt vaatimusmäärittelyt ja suunnitelmat vanhentuvat nopeasti[12]. Alkuperäisen vaatimusmäärittelyn ja suunnitelman seuraaminen ei ole ohjelmistoprojektien pääasiallinen päämäärä. Sen sijaan toimitettavan ohjelmiston tarkoitus on asiakkaan, mahdollisesti vaihtuvien, tarpeiden tyydyttäminen[6].

Internet liiketoimintaympäristönä vahvistaa ohjelmistotuotannon ongelmia korostamalla nopeutta. Asiakkaat vaativat liiketoiminnalle arvoa tuottavia ominaisuuksia nopeammassa tahdissa kuin koskaan aikaisemmin[1].

Ohjelmistojen vaatimusten muuttuminen projektin edetessä ei ole ohjelmistotuotannossa kuitenkaan uusi ongelma. Royce vuonna 1970 ja Boehm vuonna 1988 artikkeleissaan[10][3] käsittelivät ohjelmistokehityksen muuuttuvaa luonnetta.

Avaruussukkulajärjestelmän kehityksessä 70-ja 80-luvulla, aikaisempien kokemuksien perusteella, NASA ja IBM osasivat odottaa muuttuvia vaatimuksia[9]. Sukkulaohjelman ohjelmistokehityksessä muun muassa seuraavat tavoitteet otettiin huomioon:

- Toteutetaan ensimmäiseksi kehittyneimmät vaatimukset
- Maksimaalisen testauksen varmistamiseksi julkaistaan ohjelmistoa mahdollisimman nopeasti[9].

Seurauksena muuttuvista vaatimuksista useat ohjelmistoalan ihmiset ja organisaatiot kehittivät menetelmiä ja käytäntöjä, joille muutokset ovat hyväksyttyjä. Nämä uudet menetelmät toivottavat muutokset tervetulleiksi, ja ohjelmisto kehittyy uusiin vaatimuksiin ja muutoksiin mukautuen[12]. Perinteinen lähestymistapa perustui oletukselle, että aikaisella ja täydellisellä vaatimusmäärittelyllä voidaan pienentää kustannuksia vähentämällä muutoksia. Nykyään muutosten kieltäminen merkitsee reagoimattomuutta liiketoimitaympäristön kehitykselle[6].

Menetelmiä kehitettiin useita ja eri maissa:

- Taipuisa järjestelmän kehitysmenetelmä (Dynamic Systems Development) Euroopassa
- toiminnallisuus perustainen kehitysmenetelmä (Feature-Driven Development) Australiassa

- ja XP (Extreme Programming)[2], Crystal, mukautuva ohjelmistokehitys (Adaptive Software Development) ja Scrum[11] Yhdysvalloissa[12].

Helmikuussa 2001 17 menetelmien kehittäjää tapasi keskustellakseen kevyistä menetelmistä ja kokemuksiensa yhtäläisyyksistä. Huomatessaan, että heidän käytänteillään oli paljon yhteistä, ja että heidän prosessinsa tarjosivat keinoja saavuttaa merkityksellinen päämäärä: asiakkaan tyytyväisyys ja korkea laatu[12].

Osallistujat määrittelivät käytännöt ketteriksi menetelmiksi. Osallistujat kirjoittivat "Manifesto for Agile Software Development"-julistuksen, mikä kuvaa ketterän kehityksen perusarvoja:

- Yksilöt ja vuorovaikutus ennen prosesseja ja työkaluja
- Toimiva ohjelmisto ennen kattavaa dokumentaatiota
- Asiakasyhteistyö ennen sopimusneuvotteluja
- Muutoksiin vastaaminen ennen suunnitelman seuraamista[12].

Ohjelmistotuotannossa huomattiin, että ohjelmistoinsinööritieteet erosivat huomattavasti muista insinööritieteistä. Autojen kokoaminen on määriteltävä prosessi. Insinöörit voivat suunnitella prosessin, määritellä kokoonpanojärjestyksen sekä työntekijöiden, koneiden tai robottien toimenpiteet[12].

3 Ohjelmistotuotannon ongelmat

3.1 Ohjelmistotuotannon ominaispiirteet

Kunnianhimoisin tietojenkäsittelyntieteen projekti 1950-luvulla oli SAGE-järjestelmä (Semi-Automated Ground Environment) Yhdysvaltojen ja Kanadan puolustusvoimille. Projekti kokosi yhteen tutka-, viestintä- ja tietokoneinsinöörejä sekä myös ensimmäisiä ohjelmistoinsinöörejä. Ohjelmiston tuotannossa käytettävä menetelmä oli vaiheesta seuraavaan etenevä lineaarinen prosessi. Täytettyään vaatimukset ohjelmisto valmistui vuoden aikataulusta myöhässä. Ohjelmistoprojektin koettiin myöhästymisestä huolimatta onnistuneen, ja suurimpana onnistumiseen vaikuttavana tekijänä nähtiin insinööritieteistä omaksutut käytänteet[4].

Ohjelmistot merkityksen kasvaessa ihmisten ja tietokoneiden vuorovaikutus korostui yhä enemmän. Tietokonelaitteistojen merkitys väheni. Monet ohjelmistotuotantoprojektit vaativat yhä enemmän ihmisiä tuottavaan ja luovaan työhön. Muista insinööritieteistä omaksutuilla menetelmillä ei voitu arvioida ohjelmistojen luotettavuutta kattavasti. Ohjelmistotuotannon kustannukset alkoivat kasvaa. Ohjelmistoprojektien aikatauluja oli vaikea ennakoida, ja henkilöstön lisääminen aikataulun nopeuttamiseksi saattoi myöhästyttää projektia entisestään[4].

Ohjelmistotuotantoon tarvittiin enemmän insinöörejä ja matemaatikkoja kuin oli saatavilla. Ohjelmistotuotantoprojekteihin palkattiin yhä enemmän myös muiden alojen asiantuntijoita, jotka omaksuivat helposti ”ohjelmoi ja korjaa”-käytännöt insinöörimenetelmien sijasta. Suuri ero perinteisten insinöörimenetelmien ja ”ohjelmoi ja korjaa” asenteen välillä loi uutta ”hakkeri kulttuuria” merkittävien yliopistojen tietojenkäsittelylaitoksille. Nämä auktoriteetteja vastustavat luovat ”sankari ohjelmoijat” tekivät usein vaikeasti muutettavaa ja ylläpidettävää ohjelmakoodia[4].

Ennen 1960-luvun loppua NATO:n tiedekomitea järjesti kaksi ohjelmistotekniikan (software engineering) suurta konferenssia, johon osallistui monia alan ammattilaisia ja johtavia tutkijoita. Nämä konferenssit loivat vahvan pohjan ohjelmistotekniikalle ja -tuotannolle, joita teollisuus ja julkishallinnon organisaatiot käyttivät perustana vaatimuksilleen ohjelmistotuotantoprojekteissa käytettävistä menetelmistä. Oli selvää, että tarvittiin organisoituja ja kurinalaisia käytäntöjä yhä suuremmille projekteille ja tuotteille[4].

Ohjelmakoodia kirjoitettiin usein ilman taustalla olevaa suunnitelmaa. Tällainen menetelmä saattaa toimia jos tuotettava ohjelmisto on pieni, mutta järjestelmän kasvaessa uusien toiminnallisuuksien lisääminen vaikeutuu. Lisäksi virheiden korjaaminen vaikeutuu järjestelmän kasvaessa[5].

Onnistunut ohjelmistojärjestelmä vaatii erilaisten pyrkimysten koordinointia ohjelmistokehityksen aikana. Ohjelmistojärjestelmien perustavanlaatuinen ominaisuus on niiden suuri koko. Yksilöiden tai pienten ryhmien on mahdotonta luoda tai ymmärtää suuria ohjelmistoja yksityiskohtaisesti[7].

Ohjelmistolla on yleensä paljon erilaisia ohjelmakoodin polkuja (path), mitkä johtavat erilaisiin tiloihin (state). Tämä tekee tietojen määrittelystä (specification) ja ohjelmiston testaamisesta vaikeaa[4].

Suuret projektit onnistuvat useimmin jos projektia koordinoi henkilö, jolla on tietoa ohjelmiston toimialalta sekä ohjelmistolalta[7].

Tällainen ideaalitilanne on usein mahdotonta suurille ohjelmistojärjestelmille, joiden koko voi olla miljoonia tai kymmeniä miljoonia ohjelmarivejä sekä projektin kesto useita vuosia[7].

Suuren kokoluokan pyrkimykset johtavat erikoistumiseen ja työn jakamiseen. Organisaatiossa tämä johtaa toisistaan riippuvien tekijöiden jakamiseen osastoihin maantieteellisesti, organisatorisesti, sekä sosiaalisesti. Tämä vähentää mahdollisuuksia ja haluja oppia sekä jakaa tietoa etäisten työtovereiden kesken[7].

Ohjelmistotuotannon luontainen epävarmuus lisää koordinointiongelmaa. Toisin kuin teollinen valmistus, ohjelmistokehitys ei ole rutiininomainen toimi. Monet ohjelmistojärjestelmät ovat yksilöllisiä projekteja, ilman olemassa olevaa prototyyppiä, tai muokattavaa ohjelmistoa[7].

Lisäksi epävarmuus lisääntyy, koska ohjelmiston toimintaan liittyvät vaatimukset muuttuvat[7].

Muutoksia ohjelmiston vaatimuksiin esiintyy, koska ympäristö mihin ohjelmis-

to suunniteltiin muuttuu. Liiketoiminta, käyttäjien toiveet, tietokoneympäristö, ohjelmiston syötteet ja fyysinen maailma itsessään muuttuvat[7].

Muutostarpeiden ilmaantumisen todennäköisyys on suurin ohjelmistoa käytettäessä. Tällöin käyttäjät usein ymmärtävät ohjelmiston rajoitteet ja mahdollisuudet. Kun ohjelmistoa käytetään olosuhteissa, johon sitä ei ollut alunperin kuviteltu alkuperäistä suunnitelmaa tehtäessä, niin käyttäjät todennäköisesti vaativat uusia toiminnallisuuksia[7].

Ohjelmistokehitys on epävarmaa, koska vaatimukset ovat poikkeuksetta epätäydellisiä. Epätäydellisyys aiheutuu osittain rajallisista toimialan tiedoista ja ohjelmistoprojektin tyypillisestä työn jakamisesta. Liian vähällä, projektissa työskentelevillä ihmisillä, on riittävää tuntemusta toimialasta[7].

Tyypillisesti analyytikko vaihtelevalla toimialan tuntemuksella haastattelee asiakkaita ja käyttäjiä. Tämän jälkeen analyytikko kirjoittaa vaatimukset ohjelmistoarkkitehdeille ja -suunnittelijoille. Tässä prosessissa merkityksellistä toimialatietoa poikkeuksetta katoaa[7].

Kaikkia käyttäjien tarpeita analyytikko ei löydä ja jotkin tarpeet jäävät kirjaamatta vaatimusmäärittelyyn. Suuri koordinointiongelma ohjelmistokehityksessä on, että ohjelmistoarkkitehtien ja -suunnittelijoiden päätöksentekoon tarvitsema tieto ei ole saatavilla käytettävissä olevissa dokumenteissa[7].

Suuri kokoluokka ja epävarmuus olisivat vähäisempiä ongelmia, jos ohjelmisto ei vaatisi sen osajärjestelmien täsmällistä integraatiota. Ohjelmistot pääasiallisesti ovat rakennettu useista osista, jotka on kytkettävä yhteen, jotta ohjelmisto toimisi oikein[7].

Käytännön kokemus osoittaa, että aikaisemmat ohjelmistotuotannon aikaansaannokset eivät ole onnistuneet ratkaisemaan suurien ohjelmistoprojektien kooridinointiongelmia. Voidaan sanoa, että aikaisemmat ehdotetut korjaustoimenpiteet ovat lähestyneet ongelmaa seuraavasti:

- Tekniset työkalut, kuten tekstimuokkain (editor) tai korkean tason kielet
- Ohjelmiston jakaminen osiin (modularization) teknisesti, esimerkiksi olio-ohjelmoinnilla (object-oriented programming). Tai hallinnollisesti vaatimusmäärittelyn, ohjelmoinnin ja testaus toimintojen eriyttämisellä.
- Teknisillä formaaleilla menettelytavoilla,esimerkiksi versionhallinta, testisuunnitelma ja vaatimusmäärittelydokumentit[7].

ohjelmistotuotannon riskit

1. Henkilöstö vaje.
2. Epärealistinen aikataulu ja budjetti.
3. Väärien ohjelmiston toiminnallisuuksien kehitys.
4. Vääränlaisen käyttöliittymän kehitys.

5. Ohjelmiston ominaisuuksien parantelu vaatimusten täytyttyä.
6. Jatkuvasti vaihtuvat vaatimukset.
7. Puutteet ulkoisesti toimitetuissa ohjelmiston osissa.
8. Puutteet ulkoisesti suoritetuissa tehtävissä.
9. Puutteet reaaliaikaisuuden suorituskyvyssä.
10. Tietojenkäsittelytieteen valmiuksien ylikuormitus[3].

4 Ohjelmistotuotantomenetelmän valinta

4.1 Ennustettavuus

4.2 Muuttuva liiketoimintaympäristö

4.3 Organisaation koko

Viitteet

[1] Baskerville, R., B. Ramesh, L. Levine, J. Pries-Heje ja S. Slaughter: *Is "Internet-speed" software development different?* Software, IEEE, 20(6):70 – 77, nov.-dec. 2003, ISSN 0740-7459.

[2] Beck, K.: *Embracing change with extreme programming.* Computer, 32(10):70–77, oct 1999, ISSN 0018-9162.

[3] Boehm, B. W.: *A spiral model of software development and enhancement.* Computer, 21:61–72, may 1988, ISSN 0018-9162.

[4] Boehm, Barry: *A view of 20th and 21st century software engineering.* Teoksessa *Proceedings of the 28th international conference on Software engineering*, ICSE '06, sivut 12–29, New York, NY, USA, 2006. ACM, ISBN 1-59593-375-1. http://doi.acm.org/10.1145/1134285.1134288.

[5] Fowler, Martin: *The new methodology.* Wuhan University Journal of Natural Sciences, 6:12–24, 2001. http://dx.doi.org/10.1007/BF03160222.

[6] Highsmith, J. ja A. Cockburn: *Agile software development: the business of innovation.* Computer, 34(9):120 –127, sep 2001, ISSN 0018-9162.

[7] Kraut, Robert E. ja Lynn A. Streeter: *Coordination in software development.* Commun. ACM, 38:69–81, mar 1995, ISSN 0001-0782. http://doi.acm.org/10.1145/203330.203345.

[8] Larman, C. ja V.R. Basili: *Iterative and incremental developments. a brief history.* Computer, 36(6):47 –56, june 2003, ISSN 0018-9162.

[9] Madden, William A ja Kyle Y Rone: *Design, development, integration: space shuttle primary flight software system.* Communications of the ACM, 27:914–925, 1984.

[10] Royce, Winston W: *Managing the development of large software systems.* Teoksessa *proceedings of IEEE WESCON*, nide 26. Los Angeles, 1970.

[11] Schwaber, Ken: *Agile project management with Scrum.* Microsoft Press, 2009.

[12] Williams, L. ja A. Cockburn: *Agile software development: it's about feedback and change.* Computer, 36(6):39–43, june 2003, ISSN 0018-9162.