Iteratiiviset menetelmät

Jarl-Erik Malmström

Aine
Helsingin Yliopisto
Tietojenkäsittelytieteen laitos

Helsinki, 7. maaliskuuta 2013

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Jarl-Erik Malmström

Työn nimi — Arbetets titel — Title

Iteratiiviset menetelmät

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Aine	7. maaliskuuta 2013	11

Tiivistelmä — Referat — Abstract

Tiivistelmä.

Avainsanat — Nyckelord — Keywords
agile, ketterä, open source, avoin

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Tämän kirjoitelman tarkoituksena on tarkastella ohjelmistotuotannon iteratiivisia menetelmiä. Erilaisten ohjelmistotuotantomenetelmien historiaa, lähestymistapaa ohjelmistotuotantoon ja menetelmien heikkouksia sekä vahvuuksia.

Kirjoitelma käy lyhyesti läpi erilaisia lähestymistapoja ohjelmistotuotantoon. Lisäksi tarkastellaan ohjelmistotuotantoprojektien erityispiirteitä ja ohjelmistotuotannon riskien tunnistamista, sekä miten nämä vaikuttavat menetelmän valintaan.

Vaikka monet pitävät iteratiivisia ohjelmistotuotantomenetelmiä nykyaikaisina menetelminä, on niitä sovellettu ohjelmistokehityksessä 1950-luvulta lähtien. Winston Roycen ajatukset artikkelissa "Managing the Development of Large Software Systems," loivat perustan vesiputousmallille, joka vastasi valtionhallinnon sopimusten rajoitteisiin. Monet pitävät virheellisesti Roycen artikkelia lineaarisen vesiputousmallin esikuvana. Roycen artikkelin iteratiivinen ja palautteenohjaama ohjelmistokehitys, jossa onjelmisto toteutetaan kahdesti, on unohtunut useista menetelmän kuvauksista.[3]

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.[4]

Tällaisesta ohjelmistokehityksestä myös asiakkaat ovat valmiita maksamaan, sillä vaiheet pitävät sisällään aidosti luovaa työtä, joka suoraan edistää tuotettavan ohjelmiston käytettävyyttä.[4]

Suuremman ohjelmistotuotantoprojektin täytäntöönpano vaatii lisäksi muita vaiheita, jotka eivät suoraan edistä tuotettavaa ohjelmistoa ja lisäksi kasvattavat ohjelmistotuotannon kustannuksia.[4]

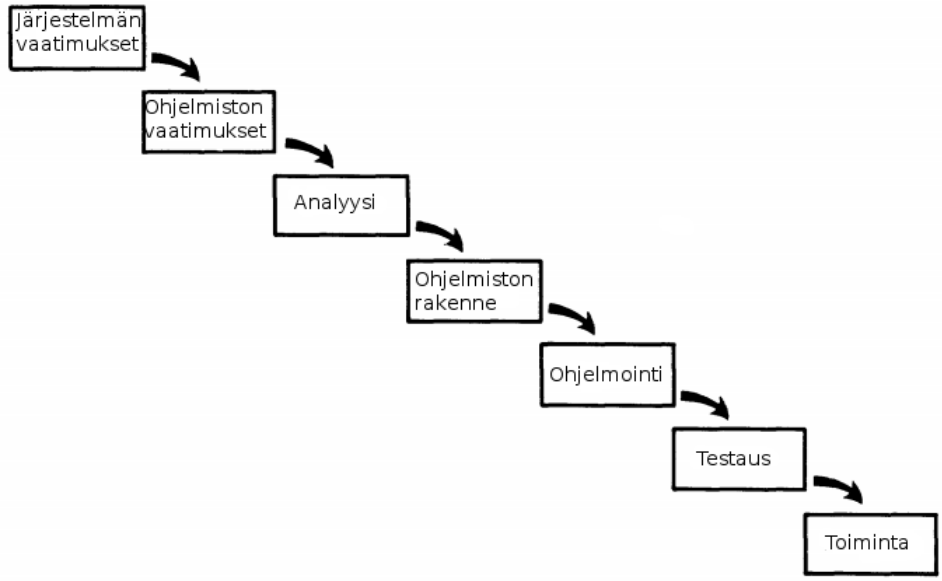
Ohjelmistotuotannon alkuaikoina käytetty "ohjelmoi ja korjaa" -mallin sisältää kaksi vaihetta. Ohjelmoidaan ensin ja mietitään vaatimuksia, rakennetta sekä testausta myöhemmin. Mallilla oli useita heikkouksia. Usean korjausvaiheen jälkeen ohjelmakoodi oli niin vaikeasti rakennettu, että oli hyvin kallista muuttaa koodia. Tämä korosti tarvetta suunnitteluvaiheelle ennen ohjelmointia.[1]

Usein hyvin suunniteltu ohjelmisto ei vastannut käyttäjien toiveita. Joten syntyi tarve vaatimusmäärittelylle ennen suunnitteluvaihetta.[1]

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.[1]

2 Ohjelmistotuotantomenetelmät

2.1 Vesiputousmalli



2.1.1 Vaiheet

1970-luvulla vesiputousmalli vaikutti suuresti lineaarisiin ohjelmistotuotannon malleihin. Vesiputousmallin lähestymistapa auttoi poistamaan monia aiemmin ohjelmistotuotantoa vaivanneita ongelmia. Vesiputousmallista tuli perusta monille teollisuuden ja hallituksen ohjelmistohankintojen standardeille. [1]

Tarkemmin Winston W. Roycen malli sisältää seuraavat vaiheet: järjestelmä- ja ohjelmistovaatimusmäärittely, analyysi, ohjelmistonrakenteen suunnittelu, ohjelmointi, testaus ja ohjelmiston käyttäminen.[4]

Perättäisten ohjelmistotuotantovaiheiden välillä on iteraatiota järjestelmän rakenteen tarkentuessa yksityiskohtaisemmaksi tuotannon edetessä. Iteraatioiden tarkoituksena on suunnitelman edetessä pitää muutosvauhti käsiteltävän kokoisena.[4]

2.1.2 Ohjelmiston suunnittelu

Lineearinen ohjelmistotuotantoprosessi sisältää huomattavan riskin. Vasta testivaiheessa, menetelmän loppupuolella, saattaa tulla esille ilmiöitä, joita ei ollut mahdollista tarkalleen analysoida aikaisemmassa vaiheessa.[4]

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%.[4]

Ongelman korjaamiseksi vaatimusmäärittelyn jälkeen - ennen analyysia - on tehtävä alustava rakenteen suunnittelu. Näin ohjelmistosuunnittelija välttää talletamiseen tai aika -ja tilavaatimuuksiin liittyvät virheet. Analyysin edetessä ohjelmistosuunnittelijan on välitettävä aika- ja tilavaatimukset sekä operatiivisiset rajoitteet analyysin tekijälle.[4]

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.[4]

2.1.3 Dokumentointi

On laadittava ymmärrettävä, valaiseva ja ajantasainen dokumentti, jonka jokaisen työntekijän on sisäistettävä.[4]

Vähintään yhden työntekijällä on oltava syvällinen ymmärrys koko järjestelmästä, mikä on osaltaan saavutettavissa dokumentin laadinnalla.[4]

Ohjelmistuotannon hyvin tärkeä sääntö on erittäin kattava dokumentointi. Ohjelmistosuunnittelijoiden on kommunikoitava rajapintojen(interface) suunnittelijoiden, ja johdon kanssa. Dokumentti antaa ymmärrettävän perustan rajapintojen suunnitteluun ja hallinnollisiin ratkaisuihin.[4]

Kirjallinen kuvaus pakottaa ohjelmistosuunnittelijan yksiselitteiseen ratkaisuun ja tarjoaa konkreettisen todistuksen työn valmistumisesta.[4]

Hyvän dokumentoinnin todellinen arvo ilmenee tuotannossa myöhemmin testausvaiheessa, ohjelmistoa käytettäessä sekä uudelleen suunniteltaessa.[4]

Hyvän dokumentin avulla esimies voi keskittää henkilöstön ohjelmistossa ilmenneisiin virheisiin. Ilman hyvää dokumenttia, ainoastaan ohjelmistovirheen alkuperäinen tekijä kykenee analysoimaan kyseessä olevan virheen.[4]

Dokumentti helpottaa ohjelmiston käyttöönottoa operatiivinen henkilöstön kanssa. Käyttöönotossa ilmenneiden mahdollisten ohjelmistovirheiden korjaamisessa selkeä dokumentti on välttämätön.[4]

2.1.4 Toinen versio

Dokumentoinnin jälkeen toinen ohjelmistoprojektin onnistumiseen vaikuttava tärkein tekijä on sen alkuperäisyys. Jos kyseessä olevaa ohjelmistoa kehitetään ensimmäistä kertaa, on asiakkaalle toimitettava käyttöönotettava versio oltava toinen versio, mikäli kriittiset rakenteelliset ja operatiiviset vaatimukset on huomioitu.[4]

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.[4]

2.1.5 Testaus

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.[4]

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.[4]

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.[4]

2.1.6 Asiakas

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.[4]

2.2 Iteratiiviset menetelmät

2.2.1 Vesiputousmallin heikkous

Lineaarisesti vaiheesta toiseen etenevän ohjelmistotuotantomentelmä ei sovi erityisesti interaktiivisiin loppukäyttäjien sovelluksiin. Suunnitelmaperustaiset standardit pakottavat dokumentoimaan yksityiskohtaisesti heikosti ymmärretyt käyttöliittymien vaatimukset.[1]

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.[1]

2.2.2 Iteratiivinen ja inkrementaalinen

NASA:n käytti iteratiivista ja inkrementaalista (IID) ohjelmistotuotantomentelmää Mercury-projektissa 1960-luvulla. Mercury-projekti toteutettiin

puolen päivän iteraatioissa. Kehitystiimi sovelsi Extreme programming-menetelmän käytänteitä tekemällä testit ennen jokaista pientä inkrementaatiota.[3]

IBM:n FSD (Federal System Division) yksikkö käytti 1970-luvulla laajasti ja onnistuneesti iteratiivisia ja inkrementaalisia menetelmiä kriittisissä Yhdysvaltain puolustusministeriön avaruus- ja ilmailujärjestelmien kehityksessä.[3]

Vuonna 1972 miljoonan koodirivin Trident-sukellusveneen komento- ja ohjausjärjestelmän kehityksessä 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.[3]

IBM:n FSD osasto kehitti Yhdysvaltain laivastolle suuren mittaluokan asejärjestelmän iteratiivisella ja inkrementaalisella mentelmällä. neljän vuoden, 200 henkilötyövuoden ja miljoonien ohjelmarivien projekti toteutettiin 45:ssä yhden kuukauden mittaisissa aikarajoitetuissa (time-boxed) iteraatiossa. Tämä oli ensimmäisiä ohjelmistoprojekteja, joka käytti nykyisten ketterien menetelmien suosittelemia iteraatiojakson pituutta.[3]

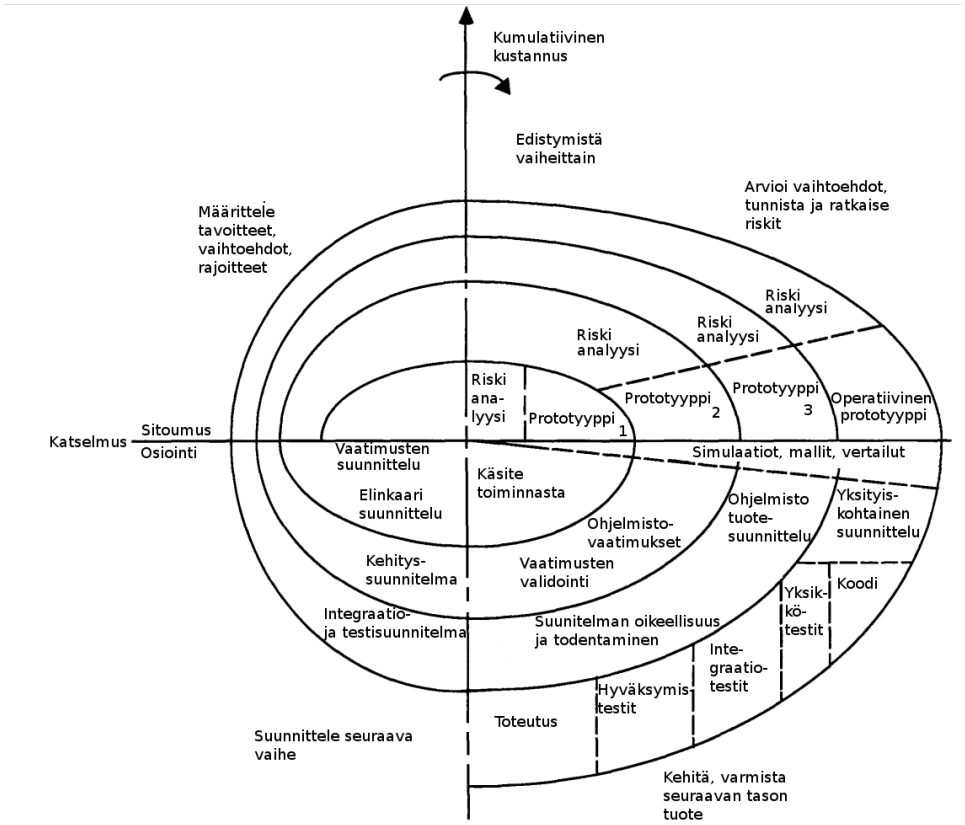
2.2.3 Spiraalimalli

Spriraalimalli on kehitetty vesiputousmallista saatujen useiden vuosien kokemuksien perusteella. Malli kuvastaa taustalla olevaa käsitettä, että jokainen vaihe sisältää saman sarjan toimenpiteitä.[1]

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).

[1]



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.[1]

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.[1]

Spriraalimalli on jaoteltu vaiheisiin riskiperusteisesti. Tämä mahdollistaa mallin mukautumaan sopivasti yhdistelemällä erittelyä (specification), prototyyppien valmistusta, simulointia tai muita lähestymistapoja ohjelmiston suunnitteluun.[1]

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.[1]

Spiraalimallin tärkeä ominaisuus on, että jokainen iteraatio päätetään katselmukseen tuotteeseen liittyvän henkilöstön tai organisaation kanssa.[1]

2.3 Ketterät kehitysmenetelmät

3 Ohjelmistojen laadun varmistus

3.1 Ohjelmistotuotannon ominaispiirteet

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.[2]

Suuret projektit onnistuvat useimmin jos projektia koordinoi henkilö, jolla on tietoa ohjelmiston toimialalta sekä ohjelmistolalta.[2]

Tällainen ideaalitilanne on usein mahdotonta suurille ohjelmistojärjestelmille, jonka koko voi olla miljoonia tai kymmeniä miljoonia ohjelmarivejä sekä projektin kesto useita vuosia.[2]

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.[2]

3.1.1 Epävarmuus

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.[2]

Lisäksi epävarmuus lisääntyy, koska ohjelmiston toimintaan liittyvät vaatimukset muuttuvat.

Muutoksia ohjelmiston vaatimuksiin esiintyy, koska ympäristö mihin ohjelmisto suunniteltiin muuttuu. Liiketoiminta, käyttäjien toiveet, tietokoneympäristö, ohjelmiston syötteet ja fyysinen maailma itsessään muuttuvat.

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. [2]

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. [2]

Tyypillisesti analyytikko vaihtelevalla toimialan tuntemuksella haastattelee asiakkaita ja käyttäjiä. Tämän jälkeen analyytikko kirjoittaa vaatimukset ohjelmistoarkkitehdeille ja -suunnittelijoille. Tässä prosessissa merkityksellistä toimialatietoa poikkeuksetta katoaa.[2]

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.[2]

3.1.2 Keskinäiset riipuvuudet

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.[2]

3.1.3 Epävirallinen keskustelu

Käytännön kokemus ja organisaatio teoria osoittavat, 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.

3.1.4 Kymmenen ohjelmistotuotannon riskiä

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.

[1]

4 Ohjelmistotuotantomenetelmän valinta

4.1 Ennustettavuus

4.2 Muuttuva liiketoimintaympäristö

4.3 Organisaation koko

4.4 Kehittäjien taidot

4.5

5

6

7 Lähteet

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

[2] 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`.

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

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