XP-ohjelmointi vs. RUP-prosessi

Pekka Helesuo

9.9.2004

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu - tutkielma

Tiivistelmä

Jokaisessa menestyvässä ohjelmistotalossa täytyy olla käytössä joku prosessi ohjelmistojen tuottamiseksi. Erilaisia prosessimalleja on runsaasti erilaisiin tarpeisiin, erilaisille tiimeille ja erilaisiin projekteihin. XP-ohjelmointi on kevyt ja tehokas prosessi pienille tiimeille ja pieniin projekteihin. Se lukeutuu ns. notkeisiin prosesseihin. RUP-prosessi on muodollisempi ja tarkemmin määritelty prosessi. Tässä tutkielmassa esitellään nämä kaksi ohjelmistotuotantoprosessia, selvitetään minkälaisiin olosuhteisiin ne soveltuvat sekä esitetään yhtäläisyyksiä ja eroja niiden välillä.

ACM-luokat (ACM Computing Classification System, 1998 version): D.2, K.6.3

Avainsanat: XP-ohjelmointi, RUP-prosessi, ohjelmistotuotantoprosessi

Sisältö

1 Johdanto **1**

2 XP-ohjelmointi **3**

2.1 XP-ohjelmoinnin ydinarvot 3

2.1.1 Kommunikaatio 3

2.1.2 Yksinkertaisuus 4

2.1.3 Palaute 4

2.1.4 Rohkeus 4

2.2 XP-ohjelmoinnin 13 käytäntöä 4

2.2.1 Kokonainen tiimi 5

2.2.2 Vertauskuva 7

2.2.3 Suunnittelupeli 8

2.2.4 Yksinkertainen suunnitelma 10

2.2.5 Pienet julkistukset 11

2.2.6 Asiakastestit 12

2.2.7 Pariohjelmointi 13

2.2.8 Testauslähtöinen kehitys 14

2.2.9 Refaktorointi 15

2.2.10 Koodin yhteisomistus 15

2.2.11 Jatkuva integrointi 16

2.2.12 Kestävä työtahti 16

2.2.13 Ohjelmointistandardi 17

2.3 XP-projektin vaiheet 17

2.3.1 Tutkimusvaihe 18

2.3.2 Suunnitteluvaihe 18

2.3.3 Iteraatiovaihe 21

2.3.4 Tuotteistamisvaihe 23

2.3.5 Ylläpitovaihe 23

3 RUP-prosessi **24**

3.1 Prosessin kuvaus 24

3.1.1 Roolit 25

3.1.2 Toiminnot 25

3.1.3 Tuotokset 26

3.1.4 Käytännöt 27
3.1.5 Työnkulut 28
3.1.6 Prosessin lisäelementit 29
3.2 Iteratiivinen kehitys 31
3.3 Arkkitehtuurikeskeinen prosessi 33
3.4 Käyttötapauspohjainen prosessi 35
3.4.1 Tapahtumavuo 35
3.4.2 Skenaariot 37
3.4.3 Käyttötapausmalli 37
3.4.4 Käyttötapausten kehittäminen 37
3.4.5 Käyttötapausten organisointi 38
3.4.6 Käyttötapaukset prosessissa 38
3.5 Prosessikäytännöt 39
3.5.1 Projektinhallintakäytäntö 39
3.5.2 Liiketoimintamallinnuskäytäntö 41
3.5.3 Vaatimuskäytäntö 42
3.5.4 Analysointi- ja suunnittelukäytäntö 43
3.5.5 Toteutuskäytäntö 45
3.5.6 Testauskäytäntö 46
3.5.7 Kokoonpanon- ja muutostenhallinnan käytäntö 47
3.5.8 Ympäristökäytäntö 49
3.5.9 Käyttöönottokäytäntö 50

4 XP-ohjelmoinnin ja RUP-prosessin vertailu 52
4.1 Prosessin määrittely 52
4.1.1 Arvot ja käytännöt 54
4.1.2 Tiimit 55
4.1.3 Roolit 55
4.1.4 Toiminnot 56
4.1.5 Tuotokset 56
4.1.6 Iteratiivisuus 57
4.2 Suunnittelu 58
4.2.1 Vaatimukset 59
4.2.2 Arkkitehtuuri 60
4.2.3 Mallinnus 61
4.3 Toteutus 61

4.4 Testaus 63
4.5 Skaalautuvuus 63
4.6 Yhteenveto 66

5 Loppusanat **67**

Viitteet **68**

1 Johdanto

Ohjelmistotuotantoprosesseille on esitetty monenlaisia määritelmiä, esimerkiksi Pressman (2000), Pfleeger (2001) sekä Firesmith & Henderson-Sellers (2002). Perinteinen vesiputousmalli sisältää peräkkäiset vaiheet, jolloin edellinen vaihe tuottaa aina tuotoksen seuraavalle vaiheelle. Ideana on myös, että koko projekti suunnitellaan etukäteen.

Arlowin & Neustadtin (2002) lyhyen määritelmän mukaan ohjelmistotuotantoprosessi määrittelee *kuka* tekee *mitä*, *koska* ja *kuinka*. Ohjelmistotuotantoprosessin käyttö on minkä tahansa ohjelmistotalon menestykselle elintärkeää. Prosesseja ja prosessityyppejä on paljon, eivätkä kaikki prosessit sovellu kaikenlaisille organisaatioille tai kaikenlaisiin projekteihin. Esimerkiksi erilaiset asiakkaat, projektityypit tai liiketoimintaympäristöt voivat vaikuttaa prosessin soveltumiseen tietylle kohdealueelle. Tässä tutkielmassa esitellään kaksi nykyaikaista ohjelmistotuotantoprosessia: XP-ohjelmointi ja Rationalin Unified-prosessi.

XP-ohjelmointi (Extreme programming, XP) on kehitetty erityisesti pienien tiimien käyttöön ja pienehköihin projekteihin (esim. Steinberg & Palmer, 2004). Sen tarkoituksena on olla helposti mukautuva ja muutoksiin sopeutuva prosessi, joka tuottaa mahdollisimman laadukasta ohjelmakoodia mahdollisimman tehokkaasti dokumentoinnin ja prosessin formaaliuden jäädessä vähemmälle arvolle. Sitä ohjaavat kommunikaation ja yksinkertaisuuden perusarvot, jotka ovat läsnä kaikessa toiminnassa.

Rationalin Unified-prosessi (Rational Unified Process, RUP) (IBM, 2004) on muodollisemmin ja tarkemmin määritelty prosessi, joka perustuu Jacobsonin & al. (1999) määrittämään Unified-prosessiin. Se määritellään roolien, toimintojen, tuotosten, työnkulkujen ja käytäntöjen avulla. Sen lähestymistapa pohjaa hyvin paljon mallintamiseen ja arkkitehtuuriin. RUP-prosessi on paitsi prosessi, myös prosessikehys. Tämä tarkoittaa, että se on muokattavissa erilaisiin organisaatioihin ja tarpeisiin sopivaksi.

Luvuissa 2 ja 3 esitellään prosesseihin liittyvät asiat käytössä olevan kirjallisuuden perusteella. Tarkoituksena on erityisesti tuoda esille tarkasteltaviin prosesseihin keskeisesti liittyvät asiat ja jättää yleiset asiat vähemmälle tarkastelulle. Tutkielmassa oletetaan, että yleiset ohjelmistotuotannon asiat tunnetaan. Tällaisia yleisiä asioita ovat esimerkiksi RUP:iin olennaisesti sisältyvä UML-kieli (OMG, 2004), vaiheistus ja vesiputousmalli sekä riskienhallinta (esim. Pressman, 2000) ja mittaaminen (esim.

Fenton & Pfleeger, 1997).

Toisaalta tutkielmassa ei oleteta XP-ohjelmoinnin ja Rationalin Unified-prosessin tuntemusta. Prosesseista kuvataan oleelliset piirteet, jotta luvun 4 vertailu olisi ymmärrettävää lukijalle. Vertailuun on käytetty prosessien kvalitatiivisia piirteitä. Tällaisia ovat esimerkiksi erilaiset lähestymistavat arkkitehtuurin, etukäteissuunnittelun ja prosessikehyksen suhteen. Eräiltä osin vertailua on suoritettu paitsi näiden kahden prosessin kesken, myös verrattuna vesiputousmallin mukaiseen ohjelmistotuotantoprosessiin erityisesti sellaisissa kohdissa ja toiminnoissa, joissa kahden esiteltävän prosessin iteratiivinen luonne on määräävässä roolissa.

2 XP-ohjelmointi

Beckin (2000) mukaan XP-ohjelmointi on kevyt, tehokas, pieniriskinen, joustava, ennustettavissa oleva, tieteellinen ja hauska tapa kehittää ohjelmistoja (Steinberg & Palmer, 2004). Se kuuluu ns. *notkeisiin prosesseihin* (agile processes). Se sopii erityisesti pienille tiimeille ja pieniin tai pienehköihin projekteihin, joissa vaatimukset ovat hämäriä ja muuttuvat alituisesti.

Tämän luvun kohdassa 2.1 esitellään XP-ohjelmoinnin ydinarvot, jotka ohjaavat kaikkia kohdassa 2.2 esiteltyjä käytäntöjä. Kohta 2.3 esittää XP-ohjelmoinnin vaiheistetun mallin, joka havainnollistaa miten XP-projekti viedään läpi.

2.1 XP-ohjelmoinnin ydinarvot

XP-ohjelmoinnin 13:a käytäntöä, jotka esitellään kohdassa 2.2, ohjaa neljä ydinarvoa, jotka esitellään tämän kohdan alakohdissa. Esiteltävät arvot ovat perusta kaikelle XP-projektissa tapahtuvalle toiminnalle.

2.1.1 Kommunikaatio

Useimmiten kaatuneiden ohjelmistoprojektien syyt voidaan tavalla tai toisella johtaa puutteelliseen kommunikaatioon. Tästä syystä XP-ohjelmoinnissa kommunikaatio on erittäin tärkeässä roolissa. Kommunikaation halutaan tapahtuvan paitsi kehitystiimin sisällä, myös kehitystiimin ja asiakkaan kesken (Newkirk, 2002). Se aika, mikä kenties perinteisemmässä ohjelmistotuotantoprosessissa menisi päätösten ja suunnitelmien dokumentoimiseen, käytetään XP-ohjelmoinnissa pikemminkin ryhmän sisäiseen, epämuodollisempaan, suoraan kommunikaatioon. Maurerin & Martelin (2002) mukaan tällaisesta lähestymistavasta on hyötyä etenkin silloin, kun kehitystiimi on suhteellisen pieni ja voidaan olettaa, että suoralla kommunikaatiolla voidaan saavuttaa koko tiimi. Suora kommunikaatio on myös nopeampaa kuin päätösten dokumentointi. Dokumentoitaessa aika kuluu tulevan ennakointiin ja sen ennustamiseen, mitä dokumentin lukijaa mahdollisesti hyödyttäisi tietää. Jos nämä ennustukset menevät pieleen, voidaan katsoa niiden tekemiseen kuluneen ajan menneen hukkaan. Samoin, jos kehitettävän järjestelmän suunnitelmat muuttuvat voimakkaasti, on dokumentointiin käytetty aika

kulunut hukkaan.

2.1.2 Yksinkertaisuus

Yksinkertaisuuden arvon mukaisesti kaikki suunnitelmat ja kaikki koodi on pidettävä aina mahdollisimman yksinkertaisina. Ohjelmakoodi kirjoitetaan nykyhetken vaatimusten mukaan sen sijaan, että yritettäisiin ennustaa tulevaisuuden vaatimuksia. Tulevaisuuden vaatimusten ennustaminen olisi vaikeaa joka tapauksessa, joten ennustamiseen käytetty aika katsotaan, varsinkin ennustuksen osuessa pieleen, usein hukkaan heitetyksi (Newkirk, 2002).

2.1.3 Palaute

XP-ohjelmoinnin käytännöt toimivat siten, että palautetta kehitettävästä järjestelmästä saadaan kehityksen alusta lähtien, usein ja paljon. Käytännöt, kuten *lyhyt kehityssykli*, *jatkuva integrointi* ja *testauslähtöinen kehitys* mahdollistavat aikaisen ja jatkuvan palautteen. Tällainen palaute mahdollistaa projektin etenemisen vakaasti (Newkirk, 2002).

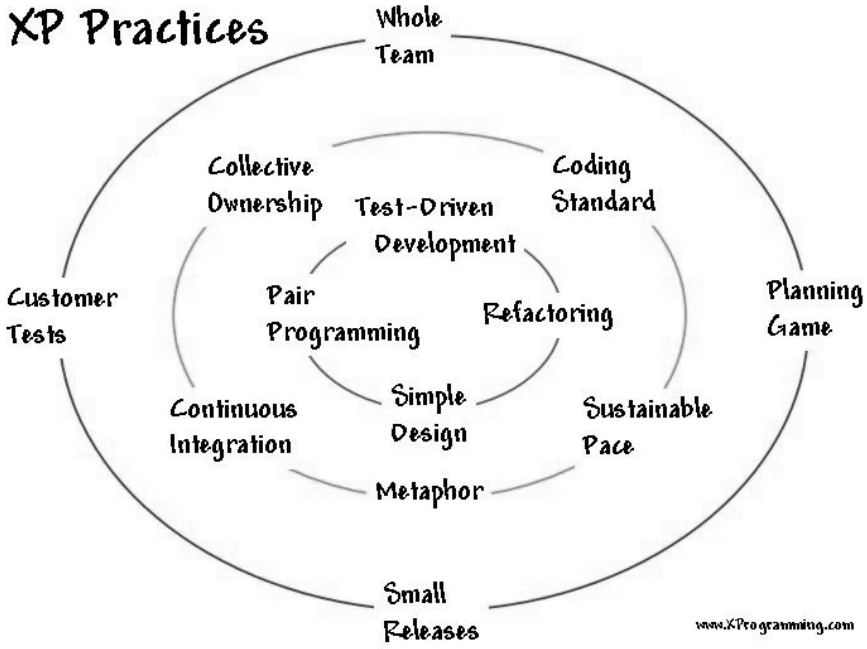
2.1.4 Rohkeus

Monet prosessit ovat defensiivisiä, varmistelevia. Toisin sanoen etukäteen tapahtuvassa suunnittelussa pyritään varautumaan kaikkiin mahdollisiin tulevaisuuden skenaarioihin. Newkirkin (2002) mukaan XP-ohjelmoinnissa rohkeudella tarkoitetaan sitä, että uskalletaan lopettaa suunnittelutyö ja aloittaa järjestelmän toteuttaminen, vaikka kaikkia tulevaisuuteen liittyviä asioita ei ole ratkaistu.

2.2 XP-ohjelmoinnin 13 käytäntöä

XP-ohjelmoinnissa on 13 käytäntöä, joita ohjaavat kohdassa 2.1 esitellyt perusarvot. Kaikki kehitystyö XP-projektissa tapahtuu näiden käytäntöjen avulla. Käytännöt kuvataan yksityiskohtaisesti tämän kohdan alakohdissa. Kuvassa 1 on kuvattu XP-ohjelmoinnin käytäntöjen nivoutuminen kerroksittain. Waken (2002) mukaan käytän-

nöt voidaan jakaa kuuluviksi kolmeen kategoriaan: yksilön, tiimin ja koko organisaation suorittamiin käytäntöihin (Astels & al., 2002). Kuvan 1 uloimmalla kehällä voidaan katsoa olevan koko organisaation, keskimmäisellä tiimin ja sisimmällä yksilön toteuttamat käytännöt.



Kuva 1: XP:n 13 käytäntöä (XProgramming.com, 2004).

2.2.1 Kokonainen tiimi

Kokonainen tiimi (whole team) luo yhteyden asiakkaan ja kehitystiimin välille. Asiakas on olennainen ja suuri osa kehitystiimiä, jolla on suuri vastuu ja valta tehdä päätöksiä kehitystyön suunnasta (Steinberg & Palmer, 2004). Tiimin kanssa toimivan asiakkaan edustajan tulisi olla yksi heistä, jotka järjestelmää lopulta käyttävät. Toiseksi, hänellä pitäisi olla kyky tuoda julki myös se, mitä hänen toverinsa järjestelmältä haluavat (Astels & al. , 2002). Maurer & Martel (2002) puhuvat tästä käytännöstä termillä *läsnäoleva asiakas* (on-site customer).

Tietyn tyyppisiä järjestelmiä kehitettäessä on vaikeaa saada vaatimukset lukittua, ennen kuin itse järjestelmää on toteutettu. Tällaisia projektityyppejä voivat olla esimerkiksi verkkopohjaiset järjestelmät (Maurer & Martel, 2002), joiden vaatimukset muuttuvat alituiseen, tai ylläpitoprojektit (Poole & Huisman, 2001). Tällöin on käytännöl-

listä, että asiakas on jatkuvasti kehitystiimin lähellä valmiina vastaamaan kehitystyön lomassa esille tulleisiin kysymyksiin, selventämässä vaatimuksia, asettamassa prioriteetteja vaatimuksille tai esittämässä täysin uusia vaatimuksia. Asiakkaan läsnäolo nopeuttaa myös ongelmanratkaisua, sillä kehitystiimi voi jatkuvasti tukeutua asiakkaaseen ongelmien ilmestyessä.

Vaikka XP-ohjelmoinnissa on pyritty häivyttämään perinteisten ohjelmistotuotantoprosessien tiimijakoa kokonaisen tiimin käsitteellä, käsittelevät Astels & al. (2002) kuitenkin projektin osanottajia kahtena tiiminä: *asiakas*- ja *kehitystiiminä*.

Asiakastiimin jäsenillä on todennäköisesti kokemusta kohdealueesta, jolle järjestelmää kehitetään, joten he määräävät projektin laajuuden, prioriteetit ja toimitusten sisällön. Asiakastiimi päättää mitä toiminnallisuutta järjestelmään tulee ja testaa toiminnallisuuden hyväksymistesteissä.

Asiakastiimin sisällä on eri rooleja (Astels & al, 2002), jotka kukin vaativat erilaisia taitoja. *Tarinankertojat* (storytellers) ovat henkilöitä, jotka kirjoittavat kohdassa 2.2.3 kuvattavia käyttäjätarinoita. Viime kädessä on heidän vastuullaan, että valmis järjestelmä vastaa vaatimuksiaan. *Hyväksyjät* (acceptors) varmistuvat hyväksymistestejä suorittamalla siitä, että kehitystiimin tuottama julkistus vastaa sille kirjoitettuja käyttäjätarinoita. *Kullanomistajat* (gold owners) huolehtivat resurssien, kuten rahan, henkilöstön ja laitteiden tarjoamisesta projektin käyttöön. *Suunnittelijat* (planners) aikatauluttavat ja suunnittelevat kehitystiimin kehittämien julkistusten käyttöönoton. *Iso pomo* (big boss) varmistaa, että kaikki voimat ovat tasapainossa ja ettei organisaatio aiheuta projektin etenemiselle esteitä.

Kehitystiimi on vastuussa tehtävien arvioinnista, asiakkaiden auttamisesta päätöstenteossa, kehitysprosessin mukauttamisesta ja järjestelmän toimittamisesta (Astels & al., 2002).

Valmentaja (coach) on henkilö, jolla on runsaasti kokemusta ja joka vastaa siitä, että projekti etenee hyvässä järjestyksessä. *Mittaaja* (tracker) seuraa projektin edistymisnopeutta verrattuna arvioituun etenemisnopeuteen sekä pitää kirjaa ja raportoi tiimin historiatiedoista. *Edistäjän* (facilitator) tehtävänä on asiakas- ja kehitystiimien välisen kommunikaation edistäminen ja konfliktitilanteiden ratkaiseminen. *Arkkitehdin* (architect) rooli on XP-ohjelmoinnissa erilainen kuin perinteisissä ohjelmistotuotantoprosesseissa. Arkkitehti ei suunnittele ideaalista järjestelmäarkkitehtuuria etukäteen,

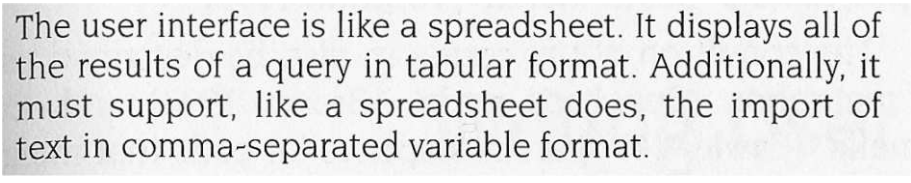
vaan tuottaa sen tarvittaessa. Rooli on välttämätön, mutta sen laajuus on muuttuva. Arkkitehdin työhön kuuluu arkkitehtuuriin liittyvien testitapausten, jotka eivät suoranaisesti sisälly käyttäjätarinoihin, suunnittelu.

Roolit eivät ole XP-ohjelmoinnissa statussymboleita, vaan pikemminkin epämuodollisia ohjenuoria siitä, mitä pitäisi tehdä ja kuinka. Roolit eivät myöskään sulje toisiaan pois. Henkilö voi siis toimia esimerkiksi valmentajana ja mittaajana samanaikaisesti. Tarkoituksena on, että henkilöt XP-projektissa toimisivat vahvuuksiensa mukaisesti. Heidän, joilla on kommunikaatio- tai ihmissuhdetaitoja kannattaa toimia edistäjinä, kun taas he, joilla on paljon kokemusta toimivat valmentajina.

2.2.2 Vertauskuva

Vertauskuvan (metaphor) eli metaforan tarkoitus on esittää tuntemattomaan kohdealueeseen sisältyvä vaikea asia ymmärrettävällä tavalla. Jokaisessa XP-projektissa pitäisi käyttää yhtä tai useampaa vertauskuvaa ohjaamaan projektia sekä tarjoamaan tiimeille yhteinen käsitteellinen konteksti (Astels & al., 2002). Vertauskuvasta puhuttaessa tarkoitetaan siis sitä, että kaikkien kehitystiimin jäsenten tulisi käyttää samanlaista kieltä ja terminologiaa puhuessaan kehitettävästä järjestelmästä. Yhteinen terminologia auttaa kaikkia tiimin jäseniä ilmaisemaan ja ymmärtämään käsitteitä samalla tavalla. Steinbergin & Palmerin (2004) esimerkki metaforasta on käsite *työpöytä* (desktop).

Pieni järjestelmä voi olla kuvattavissa yhden metaforan voimin, kun taas suurempi järjestelmä voi olla tarpeen jakaa useampiin pienempiin vertauskuviin esimerkiksi järjestelmän eri osien mukaan. Kuvassa 2 esitetään vertauskuva järjestelmän käyttöliittymälle.



Kuva 2: Vertauskuva (Astels & al., 2002).

Vertauskuva voidaan esittää esimerkiksi yhdessä käyttäjätarinassa, jossa kuvataan koko järjestelmän idea. Se antaa perusteet kaikelle tulevalle kehitystyölle. Vertauskuva ker-

too hyvin ytimekkäästi sen, minkälainen järjestelmä on tarkoitus kehittää. Maurerin & Martelin (2002) mukaan vertauskuvan voidaan katsoa olevan korkean tason kuvaus järjestelmän arkkitehtuurista. Vertauskuva voi olla vain yksinkertaistettu esitys käsitteestä tai piirteestä (Astels & al., 2002), eikä sen tarvitse olla joka suhteessa tarkka, mutta sen avulla voidaan voittaa käsitteellistämisestä ja kommunikaatiosta aiheutuvia ongelmia.

Joskus on vaikeaa erottaa mikä on vertauskuva ja mikä vaatimus, kuten kuvan 2 tapauksessa. Osa vertauskuvasta tai koko vertauskuva voi kylläkin muuttua vaatimukseksi. Silti alkuvaiheessa on tärkeintä, että asiakas onnistuu luomaan kehittäjälle jonkinlaisen mielikuvan siitä, mitä järjestelmän tulisi olla. Vertauskuvia voidaan käyttää koko järjestelmän kehittämisen ajan. Jossakin vaiheessa niitä saatetaan tiputtaa pois, jos huomataan, että ne joko on jo toteutettu tai ovat käyneet tarpeettomiksi. Ideana on, että kun on nähty jonkin muuntuvan ohjelmaksi, sen kuvausta ei enää tarvita. Toisaalta vertauskuva voidaan myös todeta vääräksi, jolloin sitä ei luonnollisestikaan enää tarvita (Astels & al., 2002).

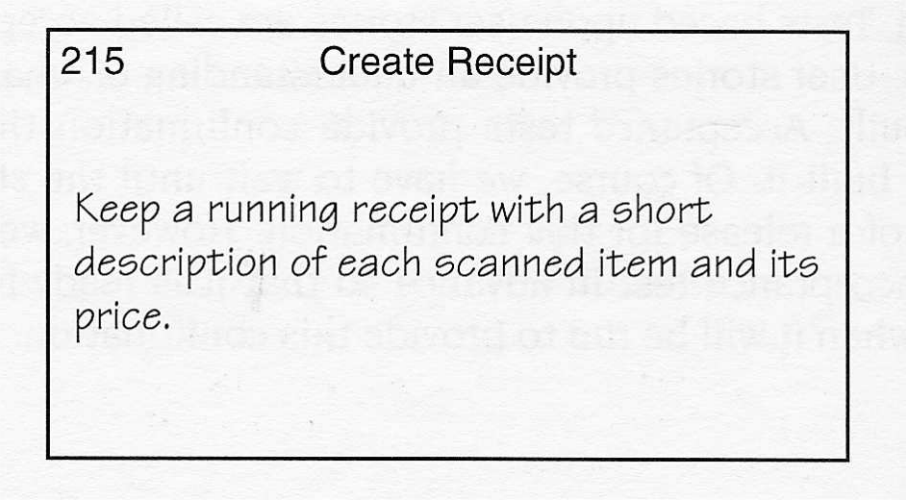
2.2.3 Suunnittelupeli

Mikä tahansa vähänkin suurempi ohjelmistoprojekti täytyy suunnitella. Suunnittelun ansiosta kaikilla osapuolilla on yhteneväinen käsitys projektin koosta ja kestosta. XP-ohjelmoinnissa tarkoituksena on saada aikaan hyvin minimaalinen suunnitelma, suuntaviivat projektin päämäärästä ja keinoista sinne pääsemiseksi sekä tiedostaa riskit. Suunnitelma on aina epätarkka ja muuttuu projektin edetessä vaatimusten muuttuessa ja uusien riskien noustessa esiin. Mitä pidemmälle tulevaisuuteen suunnitellaan, sitä epätarkempi suunnitelma on ja sitä suurempia tehtävät muutokset ovat.

Periaatteena on, että se, joka tietää ratkaisun ongelmaan, tekee suunnittelupäätökset. Asiakas tekee liiketoimintaan ja liiketoiminta-alueeseen liittyvät suunnittelupäätökset, kehitystiimi tekniseen toteutukseen (Astels & al. , 2002). XP-ohjelmoinnin suunnitteluprosessi käsittää aina seuraavan *iteraation* aikana järjestelmään lisättävät ominaisuudet. Projektin edetessä tämä antaa jatkuvasti paremman selon siitä, mitä on mahdollista toteuttaa seuraavaan julkistuksen aikarajaan mennessä. Asiakkaat ilmaisevat omat käsityksensä *käyttäjätarinoin* (user stories) (Astels & al., 2002; Steinberg & Palmer, 2004). Käyttäjätarinoissa asiakkaat kuvaavat järjestelmän yleistä käyttäytymistä. Kehitystiimi arvioi tarinoiden kustannukset ja asiakas priorisoi ne ohjaten siten projek-

tin etenemistä ja suuntaa. Aina uuden julkistuksen toimittamisen jälkeen kehitystiimi arvioi omat aiemmat arvionsa ja niiden toteutumisen. Syntynyttä tietoa käytetään hyväksi suunniteltaessa seuraavaa iteraatiota (Steinberg & Palmer, 2004).

Kuvassa 3 on lyhyt käyttäjätarina kassakuitin tulostamiseksi.



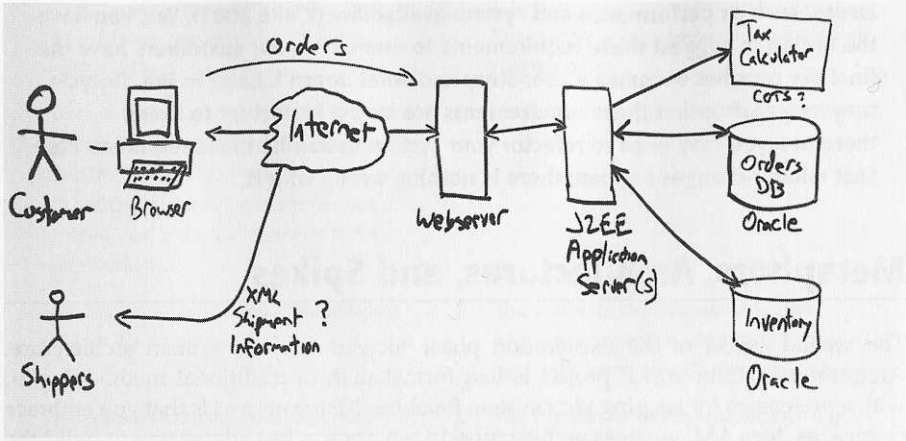
Kuva 3: Käyttäjätarina (Astels & al., 2002).

Tapa, jossa järjestelmälle asetetaan vaatimukset jo alkuvaiheessa, ennen kuin järjestelmää aletaan kehittämään, johtaa Astelsin & al. (2002) mukaan tilanteeseen, jossa lopputuote ei useinkaan kohtaa käyttäjien todellisia vaatimuksia. Vaatimusten voidaan olettaa muuttuvan ajan mukana, joten on loogista päätellä, että mitä enemmän vaatimuksia lyödään lukkoon jo projektin alkutaipaleella, sitä suurempi on todennäköisyys sille, että vaatimukset asetetaan virheellisesti. Käyttäjätarina on pienin mahdollinen tiedon määrä, jonka avulla asiakas voi kuvata jonkin toiminnon suorittamiseksi tarvittavat toimenpiteet. Käyttäjätarinoita koottaessa tulisi olla läsnä pieni määrä asiakkaan edustajia, joilla on tietämystä kohdealueeseen liittyen ja, jotka edustavat hyvin käyttäjäkuntaa. Järjestelmän kehittäjien tulisi myös olla läsnä näissä tilaisuuksissa, sillä he voivat oppia asioita järjestelmästä, jota ovat alkamassa rakentamaan (Astels & al., 2002).

Käyttäjätarinoiden kirjoittaminen on hyvä alkaa koko järjestelmän päämäärästä. Käyttäjätarina on lyhyt askelittainen kuvaus ratkaisusta päämäärään pääsemiseksi. Kukin askel voi muodostaa uuden käyttäjätarinan, joita voidaan edelleen tarkentaa. Käyttäjätarinoihin tulee sisällyttää myös mahdolliset käyttäjätarinaan liittyvät ongelmakohdat tai virhetilanteet, sekä vaihtoehtoiset tapahtumapolut. Käyttäjätarinoiden tulisi keskit-

tyä kuvaamaan järjestelmän ulkoista käyttäytymistä, sillä se on käyttäjälle näkyvä järjestelmän osa. Tarinoiden ei siis pitäisi keskittyä kuvaamaan esimerkiksi tiedostoihin kirjoittamista tai tietokantaoperaatioita. Käyttäjätarinoita voidaan lisätä, poistaa tai muokata missä tahansa projektin vaiheessa (Astels & al., 2002; Steinberg & Palmer, 2004).

Vaikka XP-ohjelmoinnissa pyritään välttämään liikaa suunnittelua, voidaan esimerkiksi UML-kieltä kuitenkin käyttää mallintamiseen (Astels & al., 2002; Ambler, 2002). Amblerin (2002) esittämä periaate *oikean tuotoksen soveltamisesta* (apply the right artifacts) on hyvä tapa UML-kaavioiden hyödyntämiseksi XP-projektissa. Periaatteen mukaan tiivistettynä kutakin kaaviotyyppiä tulisi soveltaa vain alueille, joille ne selvästi soveltuvat. Amblerin (2002) mukaan esimerkiksi käyttäjätarinoille ei kuitenkaan löydy UML-kielestä täsmällistä vastinetta, vaikkakin niiden sijalla voidaan tarvittaessa käyttää esimerkiksi käyttötapauksia. Kuvassa 4 esitetään yleinen kuvaus erään järjestelmän arkkitehtuurista.



Kuva 4: Yleiskuva järjestelmän arkkitehtuurista (Ambler, 2002).

2.2.4 Yksinkertainen suunnitelma

XP-projektin suunnitelma on pidettävä aina niin yksinkertaisena, kuin järjestelmän toiminnallinen taso sallii (Steinberg & Palmer, 2004). Ylimääräistä kompleksisuutta ei tarvita, eikä sitä toisaalta myöskään sallita. Jotta kasvava malli voitaisiin pitää yksinkertaisena, on suunnitelman käsitettävä ainoastaan seuraavan iteraation aikana lisättävät toiminnot. Kun huomataan, että koodista alkaa tulla epäselvää sitä *refaktoroidaan*, kunnes se on jälleen mahdollisimman yksinkertaista (Steinberg & Palmer,

2004).

Tapaa, jossa suunnittelua lykätään siihen hetkeen, kunnes se on välttämätöntä, Astels & al. (2002) kutsuvat *oikea-aikaiseksi suunnitteluksi* (just in time design). XP-ohjelmoinnissa pyritään suunnittelemaan kerrallaan vain niin pitkälle, kuin ehdottomasti on välttämätöntä. Tämä perustuu käsitykseen, että suurella todennäköisyydellä suunniteltuja toimintoja ei tarvita ja vielä suurempi on todennäköisyys sille, että tarvitaan jotakin täysin ennustamatonta. Kun suunnitelmia koko ajan versioidaan, parannetaan ja muutetaan vastaamaan juuri nykyhetken vaatimuksiin, ne voidaan pitää luotettavampina ja yksinkertaisina.

Beck (2000) esittää yksinkertaisen suunnitelman olevan kelvollinen, jos se suoriutuu kaikista testeistä, on selkeä ja ymmärrettävä, sisältää pienimmän mahdollisen määrän luokkia ja metodeja eikä sisällä toistoa (Astels & al., 2002). Newkirk & Martin (2001) lisäävät, että refaktorointi on avain suunnitelman pitämiseen yksinkertaisena (Astels & al., 2002).

Asiakkaan kirjoitettua käyttäjätarinat sekä niiden hyväksymistestit, kehitystiimi alkaa kehittää ratkaisua. Yksinkertaisinta ratkaisua haettaessa etsitään ensin jokin mahdollinen ratkaisu asiakkaan pulmaan, sillä ilman minkäänlaista ratkaisumallia ei voida myöskään arvioida käyttäjätarinoiden toteuttamiskustannuksia. Tiimin jäsenten pitää yhteisesti hyväksyä pohjaksi tämä ratkaisu. Jos keskusteluissa tulee ilmi useampia kuin yksi tapa ratkaista ongelma, yksinkertaisin valitaan. Tämän suunnittelusession pohjalta ei yritetä löytää järjestelmän arkkitehtuuria (Astels & al., 2002), vaan arkkitehtuuri on XP-projektissa pikkuhiljaa kasvava ilmiö.

2.2.5 Pienet julkistukset

XP-projektissa julkaistaan testattua, toimivaa koodia, jonka toiminnallisuus on asiakkaan määrittämää. Näitä julkistuksia tapahtuu usein, joten on itsestään selvää, että julkistukset ovat pieniä (Steinberg & Palmer, 2004). Sellaisten iteraatioiden päätteeksi, joissa on toteutettu riittävän suuri määrä toiminnallisuutta, että järjestelmä on järkevää toimittaa asiakkaalle, asiakas saa uuden toimivan version kehitettävästä järjestelmästä. Asiakas arvioi saamansa toimituksen ja voi sen perusteella päättää, mitä ominaisuuksia hän tahtoo mukaan seuraavaan julkistukseen, joka jälleen toimitetaan seuraavan iteraation päätteeksi. Tämä auttaa myös tarkempien suunnitelmien tekemisessä, sillä palaut-

teesta voidaan jatkuvasti ottaa oppia ja ennustaa paremmin mitä seuraavan iteraation aikana on mahdollista järjestelmään lisätä.

Vaikka julkistukset ovat pieniä ja julkistuksia tapahtuu usein, julkistuksen pitää aina olla toimiva, kokonainen ja järkevä. Vain osittaisia toimintoja toteuttavia julkistuksia ei julkaista (Astels & al., 2002).

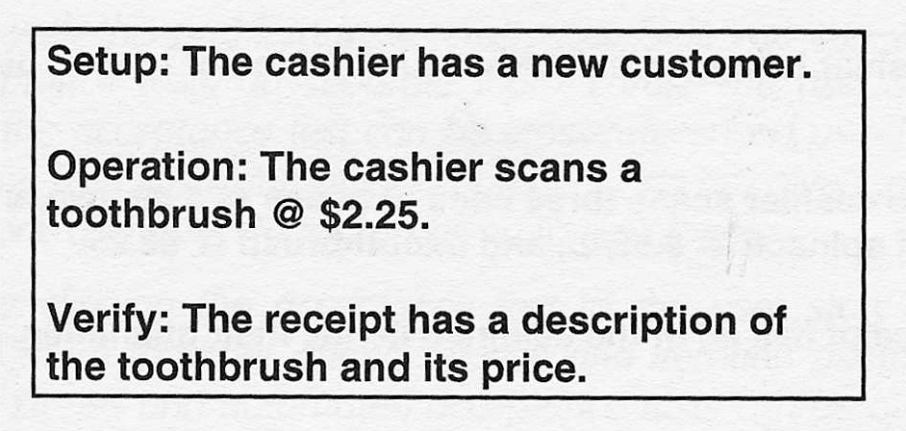
2.2.6 Asiakastestit

Asiakas suunnittelee jokaisen iteraation päätteeksi hyväksymistestin. Hyväksymisessä testataan, että toimitettu järjestelmä vastaa käyttäjätarinoissa kuvattuja tavoitteita. Asiakkaan tehtävänä on hyväksymistestien suunnittelu ja vapaamuotoinen kuvaaminen, kun taas kehitystiimi toteuttaa ja automatisoi testit sekä ajaa niitä säännöllisesti voidakseen todentaa, onko saavuttanut käsillä olevien käyttäjätarinoiden tavoitteet (Steinberg & Palmer, 2004).

Käyttäjätarina kuvaa mitä piti rakentaa ja hyväksymistesti todentaa onnistumisen. Käyttäjätarina kuvailee yleisellä tasolla järjestelmän jonkin tietyn piirteen, mutta hyväksymistesti on konkreettinen tilanne, jossa järjestelmää käytettäessä joudutaan tilanteeseen, jossa kyseinen piirre ilmenee. Siispä jokaista käyttäjätarinaa kohden pitää olla vähintään yksi hyväksymistesti, joka osoittaa, että kuvatunlainen piirre toimii. Hyväksymistestien tavoitteena on saavuttaa konkreettinen, yksinkertainen ratkaisu, jonka avulla voidaan osoittaa järjestelmän tietyn piirteen toimivan. Hyväksymistestejä kirjoitettaessa voidaan myös todeta ylimääräistä kompleksisuutta käyttäjätarinoissa, jolloin niitä voidaan jakaa uusiksi käyttäjätarinoiksi (Astels & al., 2002).

Hyväksymistesti voidaan Astelsin & al. (2002) mukaan jakaa kolmeen osaan. *Esiehdot* kertovat minkälaisten olosuhteiden vallitessa testi voidaan suorittaa. Toinen osa on itse testi, jota joskus kutsutaan myös termillä *operaatio*. Kolmanneksi arvioidaan testin *jälkiehdot*, jotka osoittavat, että testi oli onnistunut. Kuvassa 5 näkyy hyväksymistesti kuvan 3 esittämälle käyttäjätarinalle.

Hyväksymistestien automatisointi voi nopeuttaa uusien julkistusten hyväksymistä huomattavan paljon, sillä jokaisen julkistuksen jälkeen voidaan ajaa automatisoitu taantumatesti. Toisin sanoen kaikki järjestelmää varten kirjoitetut hyväksymistestit ajetaan julkistuksen jälkeen. Tämä nopeuttaa myös julkistusten käyttöönottoa. Hyväksymis-



Kuva 5: Hyväksymistesti (Astels & al., 2002).

testit voi kirjoittaa monin tavoin, eikä ole niin tärkeää mitä tekniikkaa tai välinettä hyväksymistestien automatisointiin käytetään. Jotta hyväksymis- ja julkistusprosessit toimisivat juohevasti XP-ohjelmoinnissa, on kuitenkin ensiarvoisen tärkeää, että jotain automatisoitua testaustapaa käytetään (Astels & al., 2002).

2.2.7 Pariohjelmointi

Pariohjelmointi (pair programming) on tärkein XP-ohjelmoinnin käytännöistä, sen ydinkäytäntö. Jokaisen XP-projektissa syntyvän koodirivin kirjoittavat pareittain toimivat ohjelmoijat. Toinen ohjelmoijista pitää hallussaan näppäimistöä ja hiirtä sekä kirjoittaa koodia. Toinen henkilö toimii taustalla huomauttaen kirjoitusvirheistä, tehden ehdotuksia koodin suhteen, tai toimien missä tahansa tarpeellisessa roolissa. Henkilöt voivat vaihtaa paikkoja milloin vain (Steinberg & Palmer, 2004).

Sinänsä itse termi pariohjelmointi on hieman harhaanjohtava. Ohjelmointi on nimittäin vain yksi aktiviteeteista, joita pari toteuttaa. Astels & al. (2002) esittävät, että samanaikaisesti koodauksen kanssa tapahtuu muita toimintoja, kuten suunnittelua, analysointia, refaktorointia, testausta ja koodin tarkastusta. Parit eivät XP-projektissa ole kiinteitä, vaan vaihtelevat päivittäin ja tehtävittäin. Pariohjelmoinnista ei ole hyötyä vain siinä mielessä, että toinen voi tarkkailla toisen tekemiä virheitä. Koodia tuottavalle voi olla myös avuksi pakko tuottaa koodia ääneen ajatellen. *Vertaispaineella* (peer pressure) varmistetaan kurinalainen työskentely, käytäntöjen noudattaminen, vaatimusten täyttäminen ja mahdollisten ongelmien löytyminen. Vertaispaineen syntymistä edistää esimerkiksi se, että parit vaihtuvat koko ajan. Näin varmistetaan, ettei

jokin tietty pari laiminlyö työskentelytapoja.

2.2.8 Testauslähtöinen kehitys

Kun kehitystiimi on asettanut seuraavan tavoitteensa, tiimin jäsenet suunnittelevat ja kirjoittavat testitapaukset tulevalle koodille. Testitapausten pitää todentaa, että seuraavaksi kirjoitettava koodi täyttää sille asetetun tavoitteen. Nämäkin testit automatisoidaan ja lisätään testisarjaan, joka kasvaa jatkuvasti kehitystyön edetessä. Koko testisarjaa ajetaan jatkuvasti ja usein kehityksen aikana, mikä tarjoaa runsaasti palautetta kehitettävästä järjestelmästä (Steinberg & Palmer, 2004).

Koodia, jolle ei ole kirjoitettu testiä, ei voida pitää valmiina. Kirjoittamalla testit ennen koodausta voidaan saavuttaa hyötyä, koska automatisoitu testi toimii eräänlaisena määrityksenä tai dokumentaationa kirjoitettavalle koodille. Kun kirjoitetaan ensin testi, tulee testistä osa ongelman ratkaisua. Kirjoitettava koodi täytyisi joka tapauksessa suunnitella ensin. Kirjoittamalla testi suoraan käytettävällä ohjelmointikielellä saadaan samalla myös määriteltyä tulevan koodin toiminnallisuus. Tällä tavoin vältytään suurilta työmääriltä pelkän testikoodin kirjoittamisessa. Toisaalta, kun testit on suunniteltu hyvin ja ne ovat kattavia, voi ohjelmoija huoletta pitää kirjoittamaansa toimintoa valmiina ja siirtyä eteenpäin, kunhan koodi suoriutuu sille tehdyistä testeistä (Astels & al. , 2002).

Kuva 6 esittää Java-kielistä (Sun, 2004) metodia, jota käytetään luomaan kaksi testaamisessa käytettävää Customer-oliota.

```
protected void setUp() {
  dave =
    new Customer(
      "Dave Astels",
      "542-4562",
      "Wolfville");
  miro =
    new Customer("Miro Novak", "555-1111",
"Calgary");
 }
```

Kuva 6: Yksikkötestin metodi (Astels & al., 2002).

Yksikkötestit eroavat hyväksymistesteistä siten, että hyväksymistestit testaavat kokonaisia käyttäjätarinoita, kun taas yksikkötestit testaavat tehtäviä eli käyttäjätarinoiden

osia. Käyttäjätarinoiden ositus tehtäviksi on osa kohdassa 2.3.2 esitettävää suunnitteluvaihetta. Tyypillisiä yksikkötestauksen kohteita ovat Astelsin & al. (2002) mukaan esimerkiksi luokat ja metodit. Yksikkötestit ajetaan joka kerran, kun koodia muutetaan. Testitapauksia on varmempaa kirjoittaa liikaa kuin liian vähän. Vain kaikkein yksinkertaisimmat, kuten vain yhden olion yhden attribuutin arvon palauttavat metodit, voidaan jättää testaamatta. Testejä kirjoitettaessa voi myös tulla ilmi tarpeettomia metodeja tai luokkia, jotka ovat voineet käydä tarpeettomiksi esimerkiksi käyttäjätarinoiden muuttumisen seurauksena.

Järjestelmän kasvaessa myös testitapausten määrä kasvaa, jolloin koko testisarjan ajaminen vie kauemmin. Testaus ei saisi kuitenkaan hidastaa itse kehitystyötä. Testien ajamista voidaankin rajoittaa koskemaan vain niitä testitapauksia, jotka koskevat vain tiettyä kehityksen alla olevaa kohdetta. Toisaalta joissain tapauksissa, erityisesti integroitaessa uutta koodia, on välttämätöntä ajaa koko testisarja (Astels & al., 2002).

2.2.9 Refaktorointi

Kun järjestelmän koodissa huomataan puutteita, ylimäärää tai sekavuutta sitä *refaktoroidaan* (code refactoring). Tärkeää koodin refaktoroinnissa on se, ettei suunnitelmia tai koodia muutettaessa muuteta niiden ulkoista käyttäytymistä (Newkirk, 2002). Fowlerin (1999) mukaan koodin refaktorointi on systemaattista, kurinalaista ja ankaraa koodin sisäisen esityksen asteittaista parannusta (Astels & al., 2002). Tavoitteena on pitää koodi jatkuvasti niin yksinkertaisena ja tehokkaana kuin mahdollista. Koodin refaktorointia suoritetaan jatkuvasti koko projektin ajan.

Refaktorointi tapahtuu pienin askelin, joista jokaisen jälkeen on tarkistettava, ettei järjestelmän ulkoinen toiminta ole muuttunut. Jos refaktoroinnin aikana syntyy virhe, testit kertovat siitä heti, eikä virhettä todennäköisesti ole vaikeaa korjata.

2.2.10 Koodin yhteisomistus

XP-ohjelmoinnissa koodin omistamiseen on toisenlainen lähestymistapa, kuin perinteisissä ohjelmistotuotantoprosesseissa: koko tiimi omistaa kaiken järjestelmän koodin (Steinberg & Palmer, 2004). Kun useat parit voivat muokata samanaikaisesti koodin samaa osaa, on versionhallinnan käyttäminen luonnollisesti välttämätöntä. Astels &

al. (2002) lisäävät myös, että pelkkä versionhallinnan käyttäminen ei riitä. Käytettävä versionhallintajärjestelmä ei saa pohjautua kirjoitussuojauksiin, vaan kaikkien tiimin jäsenten pitää päästä käsiksi mihin tahansa koodin osaan milloin tahansa. Jotta tämä tekniikka toimisi, on välttämätöntä harrastaa jatkuvaa integrointia vähintään päivittäin, jolloin mahdolliset ristiriitaiset muutokset huomataan nopeasti ja ne ovat todennäköisesti pieniä.

2.2.11 Jatkuva integrointi

Riippumatta siitä, mikä on järjestelmän sen hetkinen toiminnallinen taso, järjestelmän pitää kääntyä, olla ajettava ja selviytyä kaikista testeistä. Tästä seuraa, että kaiken järjestelmään lisättävän koodin tulee kääntyä, olla ajettavaa ja selviytyä testeistä (Steinberg & Palmer, 2004). Jatkuvasta integroinnista seuraa, että järjestelmästä voidaan joutua ottamaan lukuisia uusia *koosteita* (build) päivässä (Steinberg & Palmer, 2004). Koosteella tarkoitetaan versiota koko järjestelmästä tietyllä hetkellä. Kaikelle lisättävälle koodille asetettujen vaatimusten mukaisesti uusimmat versionhallinnasta löytyvät tiedostot muodostavat yksiselitteisesti uusimman version koko järjestelmästä. Erityisasemassa oleva kooste on julkistus, joka julkaistaan iteraation lopuksi ja joka toteuttaa nykyisen iteraation käyttäjätarinoissa kuvatut tavoitteet.

Astels & al. (2002) muistuttavat, että integroinnin pitäisi tapahtua vähintään päivittäin. Ideaalitilanne olisi heidän mukaansa useita integrointeja päivässä, esimerkiksi jokaisen osatehtävän suorittamisen jälkeen. Testeistä suoriutuminen todistaa aina sen, että järjestelmästä voi julkaista uuden version. Kun integroidaan näin usein, minimoidaan mahdollisuus, että lisätty koodi olisi konfliktissa jonkun toisen lisäämän koodin kanssa. Vaikka konflikti olisikin olemassa, se on todennäköisesti minimaalinen. Astelsin & al. (2002) mukaan koodin integrointi on sitä tuskaisempaa, mitä pidempi aika on integrointien välissä. Näin ollen jatkuva integrointi vähentää integroinnista aiheutuvia ongelmia.

2.2.12 Kestävä työtahti

Kestävä työtahti (sustainable pace) tarkoittaa sitä, että jokaiseen iteraatioon pitäisi kulua kehitystiimiltä suunnilleen sama työpanos ja työaika. Newkirk (2002) puhuu tästä käytännöstä termillä *40-tuntinen viikko*. Ylityöt voivat johtaa stressiin, lisäylitöihin

ja väsymyksestä johtuviin virheisiin, joten pitkällä tähtäimellä säilyttämällä kestävä työtahti voidaan tuottaa parempia ohjelmia lyhyemmässä ajassa (Steinberg & Palmer, 2004). Käytäntö ei ole siinä mielessä sitova, ettei XP-projektissa koskaan tehtäisi yli 40-tuntista työviikkoa, mutta ylityöt eivät saa olla pysyvä ilmiö.

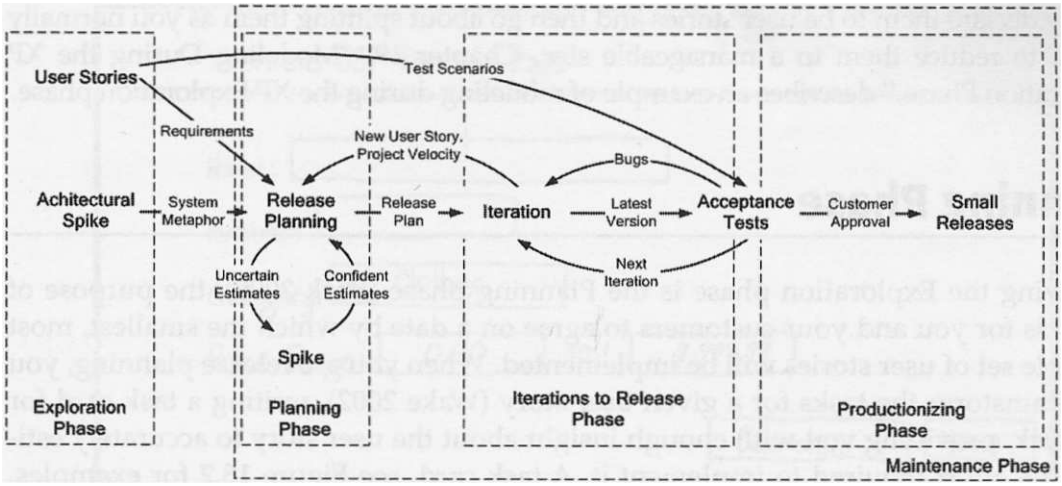
2.2.13 Ohjelmointistandardi

Kehitystiimin kaikkien jäsenten pitää käyttää samaa ohjelmointistandardia. Ohjelmointistandardi varmistaa fokuksen pysymisen itse koodin sisällössä, koska kaikki käyttävät samaa ohjelmointistandardia. Siten esimerkiksi koodin ulkoasu, kommentointikäytäntö ja muuttujien nimeäminen ovat kaikille tuttuja, joten niihin ei tarvitse kiinnittää niin paljon huomiota. Sillä, mikä tämä standardi on, ei ole niin suurta merkitystä kuin sillä, että sitä todella käytetään (Steinberg & Palmer, 2004; Astels & al., 2002). Käytäntö toteuttaa kommunikaation avainarvoa, sillä yhteiset ohjelmointistandardit auttavat koodin pitämisessä yhteneväisenä luokasta toiseen, jolloin koodin ymmärtäminen on kaikille helpompaa (Newkirk, 2002).

2.3 XP-projektin vaiheet

Vaikka XP-ohjelmointi ei suoranaisesti ole vaiheistettu prosessi perinteisessä mielessä, voidaan XP-projekti esittää myös vaiheistettuna. Tässä kohdassa havainnollistetaan XP-projektin elinkaari vaiheiden avulla. Kuva 7 esittää XP-projektin vaiheet Beckin (2000) mallin mukaan. Tämän kohdan alakohdissa käydään läpi nämä vaiheet tarkemmin. Kohdassa 2.3.1 esitellään tutkimusvaihe (exploration phase), jonka aikana luodaan projektin perusta ja alkuvaiheen vaatimukset. Kohdassa 2.3.2 esitellään suunnitteluvaihe (planning phase), jonka seurauksena päätetään julkistuksen sisällöstä. Kohdassa 2.3.3 esiteltävä iteraatiovaihe (iterations to release phase) käsittää pääosan itse kehitystyöhön liittyvistä seikoista. Kohdassa 2.3.4 esitellään tuotteistamisvaihe (productionizing phase) ja kohdassa 2.3.5 ylläpitovaihe (maintenance phase).

Ambler (2002) painottaa vielä erikseen, ettei vaiheella XP-ohjelmoinnissa ole samaa merkitystä kuin esimerkiksi vesiputousmallissa, vaan vaiheiden välillä voidaan vuorotella, kuten kuva 7 osoittaa, tai ne voivat esiintyä samanaikaisesti. Vaiheiden avulla voidaan lähinnä kuvata kehitystyön eri aspektien painoarvoja eri vaiheissa ja toisaalta havainnollistaa XP-projektin elinkaarta ymmärrettävämmin.



Kuva 7: XP-projektin vaiheet (Ambler, 2002).

2.3.1 Tutkimusvaihe

Vaiheista ensimmäinen on tutkimusvaihe (Ambler, 2002). Se käsittää alustavan arkkitehtuurin tutkimisen sekä alkuvaiheen käyttäjätarinoiden kirjoittamisen. Beck (2000) esittää, että alkuvaiheessa kerättyjen käyttäjätarinoiden tulisi sisältää mahdollisimman kattava kuvaus järjestelmästä, sillä ne määrittävät projektin *laajuuden* (scope) (Ambler, 2002). Astelsin & al. (2002) mukaan kuitenkin päätökset projektin laajuudesta tehdään vasta julkistusta suunniteltaessa, mikä kuvataan kohdassa 2.3.2.

2.3.2 Suunnitteluvaihe

Tutkimusvaihetta seuraa suunnitteluvaihe. Vaiheen tarkoituksena on päättää ensimmäisen julkistuksen päivämäärä ja tuottaa julkistussuunnitelma. Julkistuksen tulisi sisältää pienin ja asiakkaalle arvokkain joukko käyttäjätarinoista (Ambler, 2002). Astelsin & al. (2002) mukaan julkistusten suunnittelun tarkoitus on tarjota asiakkaalle kuva projektin kokonaiskestosta. Jotta projektin kokonaiskestoa ja -kustannuksia voidaan arvioida, tarvitaan jokin mekanismi käyttäjätarinoiden, eli järjestelmän vaatimusten työmäärien estimoimiseksi.

Käyttäjätarinoiden työmäärien estimointi

Estimaatio on Astelsin & al. (2002) mukaan käyttäjätarinan toteuttamiseen yhdeltä

kehittäjältä vaadittava työmäärä *tarinapisteinä* tai *ideaalisina viikkoina*. Ideaalisella viikolla tarkoitetaan, että viikon kaikki 40 työtuntia voidaan käyttää ohjelmoimiseen, mikä ei tietenkään ole mahdollista todellisuudessa. Iteraatioiden kannalta ideaalinen kesto käyttäjätarinalle olisi 1-3 tarinapistettä (ideaalista työviikkoa), jolloin se olisi melko yhtäpitävä iteraation keston kanssa, jollaisen kesto tyypillisesti on noin kaksi viikkoa (Astels & al., 2002). Käytännössä käyttäjätarinoiden koko vaihtelee suuresti. Pienimmät käyttäjätarinat, joiden kesto on alle yhden tarinapisteen, voidaan sijoittaa iteraatioiden täytteeksi, joskaan liian suuri määrä liian pieniä käyttäjätarinoita ei ole hyvä asia.

Estimoitaessa pitää Beckin & Fowlerin (2001) mukaan olettaa jokaisen tarinan olevan itsenäinen (Astels & al., 2002). Näin voidaan olettaa kaikkien käyttäjätarinaan sidoksissa olevien käyttäjätarinoiden olevan jo toteutetut. Riippuvuudet huomioidaan sekä julkistuksia suunniteltaessa että suunniteltaessa iteraatioita kohdan 2.3.3 kuvauksen mukaan, mutta ei estimoinnin aikana. Toinen olettamus on, että infrastruktuurin rakentamista käyttäjätarinoille voidaan lykätä. Tässä kontekstissa infrastruktuurilla tarkoitetaan sellaisia koodin osia, jotka eivät käy suoranaisesti ilmi käyttäjätarinoista, mutta joita järjestelmä tarvitsee toimiakseen, kuten erilaisia komponentteja ja luokkakirjastoja (Ambler, 2002). Beckin & Fowlerin (2001) mukaan käyttäjätarinan tarvitsema infrastruktuuri rakennetaan vasta, kun sen rakentaminen on välttämätöntä (Astels & al. , 2002).

Liian suuret käyttäjätarinat on mahdotonta toteuttaa lyhyessä iteraatiossa, eikä osittaista käyttäjätarinaa voida toteuttaa. Niinpä ne pitää pilkkoa pienemmiksi. Pilkkominen voidaan hoitaa esimerkiksi tekemällä käyttäjätarinan askelista omia käyttäjätarinoitaan. Käyttäjätarinan pilkkomisen seurauksena syntyy useita uusia käyttäjätarinoita, joiden tulisi asettua 1-3 tarinapisteen väliin (Astels & al., 2002).

Tiimin nopeus

Kun käyttäjätarinoiden työmäärät on estimoitu tarinapisteinä, voidaan laskea arvio projektin kokonaiskestosta. Jotta projektin kestoa voitaisiin arvioida, pitää tietää *tiimin nopeus*. Tiimin nopeudella tarkoitetaan sitä tarinapisteiden määrää, jonka tiimi voi toteuttaa yhdessä iteraatiossa (Astels & al., 2002). Tiimin nopeus muuttuu tyypillisesti kokemuksen myötä ja projektin edetessä. Mittaajan tehtävä on mitata ja seurata tiimin nopeuden muutoksia. Kun tiedetään tiimin nopeus ja projektin suorittamiseksi vaadit-

tavien estimoitujen käyttäjätarinoiden yhteenlaskettu tarinapisteiden määrä, voidaan projektin oletettu kokonaiskesto viikkoina laskea kaavan (1) mukaisesti

$$p = (k/v) * i \tag{1}$$

missä *p* tarkoittaa projektin oletettua kokonaiskestoa viikkoina, *k* tarinapisteiden kokonaismäärää, *v* tiimin nopeutta ja *i* iteraation kestoa.

Julkistussuunnitelma

Jos projekti on pieni, tiimi riittävän iso eikä ole aikarajaa, voidaan projektin kokonaiskustannukset laskea suoraan kaavan (1) mukaan. Ohjelmistoprojekteilla on kuitenkin useimmiten aikaraja, mihin mennessä asiakas tarvitsee valmiin järjestelmän. Jos osoittautuu, että kaikkien käyttäjätarinoiden toteuttaminen ei aikarajan puitteissa onnistu, asiakkaan pitää päättää mitä toiminnallisuutta voidaan karsia. Toisin sanoen asiakkaan pitää päättää projektin laajuudesta. Kun projektin laajuus on päätetty, pitää eri toiminnot priorisoida. Asiakas arvottaa järjestyksen, missä tahtoo toiminnallisuutta lisättävän. Jotkut käyttäjätarinat ovat välttämättömiä järjestelmän toiminnalle, toisia voidaan lykätä tai jättää pois. Kuvassa 8 on esimerkki julkistussuunnitelmasta. Kuvattuna ovat käyttäjätarinat, prioriteetit, estimoidut työmäärät sekä luku, joka kertoo missä iteraatiossa mikäkin käyttäjätarina on tarkoitus toteuttaa. Käyttäjätarinoiden priorisointi voidaan suorittaa yksittäisten käyttäjätarinoiden sijasta myös suurempien kokonaisuuksien osalta, jolloin priorisointi voidaan suorittaa makrotasolla (Astels & al., 2002).

Julkistuksia suunniteltaessa saatetaan projekti hylätä, ellei saavutettavan hyödyn suhde hintaan tunnu järkevältä. Astelsin & al. (2002) mukaan tämä on myös hyvä vaihe hylätä projekti, ennen kuin kukaan on sijoittanut siihen suurempia panostuksia. Iteraatiovaiheeseen siirtyminen on asiakkaan puolelta osoitus sitoutumisesta projektiin, mikä luonnollisesti on tärkeää myös kehitystiimille. Iteraatiovaiheessa tapahtuvassa iteraatioiden suunnittelussa käytetään pohjana julkistussuunnitelmaa.

Story	Priority	Estimate	Iteration
Invoices	A	4	2
Quotes	A	4	1
Associated Quote	C	1	6
Invoice Maintenance	A	3	3
Quote Maintenance	A	3	3
Customers	A	1	1
Customer Maintenance	B	2	5
Parts	A	1	1
Part Maintenance	B	2	5
Persistance	A	6	4
Print Quote	A	2	2
Print Full Invoice	A	2	5
Print Package Invoice	B	2	6
Print Difference Invoice	C	3	6

Kuva 8: Julkistussuunnitelma (Astels & al., 2002).

2.3.3 Iteraatiovaihe

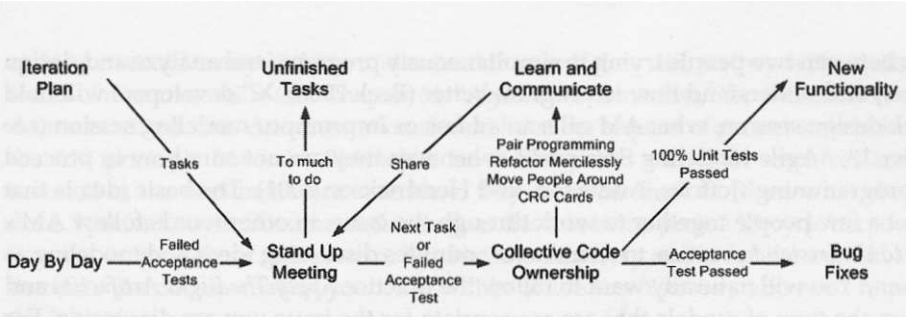
Beckin (2000) vaihemallin mukainen iteraatiovaihe on työläin XP-projektin vaiheista (Ambler, 2002). Vaiheessa tapahtuu suurin osa itse kehitystyöstä eli mallinnuksesta, ohjelmoinnista, testauksesta ja integroinnista. Kuva 9 esittää iteraation elinkaarta. Iteraatioiden suunnittelu on samanlaista toimintaa kuin julkistuksen suunnittelukin. Ero on, että iteraatioita suunniteltaessa keskitytään nykyisen iteraation käyttäjätarinoihin. Toinen ero on tarkkuus: julkistussuunnitelma käsittää estimoituja käyttäjätainoita, iteraatiosuunnitelma estimoituja *tehtäviä*, eli käyttäjätarinoiden osia. Käyttäjätarinat ositetaan tehtäviksi, jolloin käyttäjätarinoiden työmäärien estimointi helpottuu. Kehitystiimi arvioi aivoriihessä välttämättömät tehtävät käyttäjätarinan toteuttamiseksi. Yksittäisten tehtävien tulee olla lyhytkestoisia ja testattavia. Muutamia poikkeuksia lukuunottamatta tehtävän kesto ei saisi olla yli kahdeksaa tuntia. Tätä pidemmät tehtävät tulisi pyrkiä jakamaan osiin (Astels & al., 2002).

Tietyn iteraation parissa työskenneltäessä saatetaan löytää uusia käyttäjätarinoita joita ei vielä ole estimoitu. Niiden osalta tulee etsiä käyttäjätarinoihin sisältyvät tehtävät, jotta käyttäjätarinat voidaan estimoida. Uusien käyttäjätarinoiden estimoinnin jälkeen

saatetaan joitakin käyttäjätarinoita joutua lykkäämään mikäli ne eivät mahdu iteraatioon, tai toisaalta uusia käyttäjätarinoita voidaan ottaa mukaan (Ambler, 2002).

Kuva 9: Iteraation elinkaari (Ambler, 2002).

Iteraatiosuunnitelma toimii pohjana XP-projektin päiväkohtaiselle toiminnalle, jota kuva 10 esittää.



Kuva 10: Päiväkohtainen toiminta (Ambler, 2002).

Iteraation kustannukset

Iteraatioiden suunnittelussa siis julkaisusuunnitelma jaetaan useiksi aikarajatuiksi iteraatioiksi, joiden ideaalinen kesto on noin kaksi kalenteriviikkoa (Astels & al., 2002). Kun tietyn iteraation kesto on valittu voidaan käyttäjätarinat sijoittaa iteraatioihin. Iteraation kustannukset tarinapisteinä voidaan laskea kaavasta (2)

$$kustannukset = a * l/t \tag{2}$$

missä *a* tarkoittaa iteraation kalenterikestoa, *l* kehittäjien lukumäärää ja *t* todellisten ja ideaalisten työviikkojen välistä suhdetta eli sitä, kuinka monta todellista työviikkoa

tiimi joutuu tekemään yhden ideaalisen työviikon saavuttamiseksi. Siten esimerkiksi kolmeviikkoiseksi aikarajattuun, kymmenen kokematonta kehittäjää käsittävään iteraatioon voidaan valita käyttäjätarinoita kymmenen tarinapisteen edestä, jos oletetaan, että todellisten ja ideaalisten viikkojen suhde on 3:1. Asiakas- ja kehitystiimit yhdessä valitsevat iteraatioissa toteutettavat käyttäjätarinat.

Kun käyttäjätarinat on sijoitettu iteraatioihin, iteraatiot pitää kiinnittää kalenteriaikoihin. Mittaajien ja suunnittelijoiden tulee varmistaa, että molempien tiimien kaikki jäsenet tiedostavat aikarajat ja toimivat niiden mukaan. Eräs tärkeimmistä seikoista iteratiivisessa prosessissa on aikarajojen noudattaminen. Toisin sanoen, jos jokin toiminnallisuus on myöhässä, se siirretään seuraavaan iteraatioon. Julkistuspäivämääränä toimitetaan siitä huolimatta julkistus. Tämä metodi pakottaa kehitystiimiä sisäisesti arvioimaan eri käyttäjätarinoiden tärkeyksiä ja aloittamaan tärkeimmistä (Astels & al., 2002).

2.3.4 Tuotteistamisvaihe

Tuotteistamisvaiheessa varmistetaan, että kehitettävä järjestelmä on valmis tuote. Toisin sanoen vaihe sisältää esimerkiksi suuren mittakaavan järjestelmä-, kuormitus- ja asennustestausta (Ambler, 2002). Vaiheessa järjestelmän evoluutio hidastuu, koska painoarvo on testaamisessa. Kehittyminen ei silti pysähdy. Suurin ero verrattuna iteraatiovaiheeseen on, että julkistaminen tapahtuu oikeaan ympäristöön kehitysympäristön sijasta. Ambler (2002) esittää tarpeellisen dokumentaation tuottamisen eri sidosryhmille tapahtuvan tuotteistamisvaiheesssa.

2.3.5 Ylläpitovaihe

Vaihemallin viimeinen vaihe on ylläpitovaihe (Ambler, 2002). Vaihe on XP-projektin normaalitila ja se on eräänlainen paketti joka sisältää suunnittelu-, iteraatio- ja tuotteistamisvaiheet, kuten kuva 7 osoittaa. Yleensäkin kaikki aktiviteetit mitkä perinteisessäkin prosessissa kuuluvat ylläpidon piiriin, kuuluvat siihen myös XP-ohjelmoinnissa.

3 RUP-prosessi

Tässä luvussa esitellään toinen tässä tutkielmassa käsiteltävistä ohjelmistotuotantoprosesseista, Rationalin Unified-prosessi eli RUP.

Kruchten (2004) määrittelee RUP:in kurinalaiseksi lähestymistavaksi työtehtävien ja vastuiden asettamiseen ja hallinnoimiseen ohjelmistoyrityksessä. RUP toteuttaa monia hyville ohjelmistotuotantoprosesseille kuvatuista käytännöistä. Koska RUP on paitsi prosessituote myös prosessikehys, se tuottaa käytännöt muodossa, jossa ne ovat räätälöitävissä erilaisille projekteille ja organisaatioille. RUP:in päämääränä on varmistaa, että tuotetaan korkealaatuisia ohjelmistotuotteita, jotka kohtaavat asiakkaiden vaatimukset ja joiden toteuttamisaikataulu ja -budjetti ovat ennustettavissa.

Kohdassa 3.1 on kuvaus RUP:in kokonaisuudesta. Kohta 3.2 esittelee iteratiivisen kehityksen periaatteet RUP:issa, kohta 3.3 esittelee arkkitehtuurikeskeisen lähestymistavan, kohta 3.4 käyttötapausten käytön järjestelmän pohjana ja lopuksi kohdassa 3.5 kuvataan RUP:in prosessikäytännöt.

3.1 Prosessin kuvaus

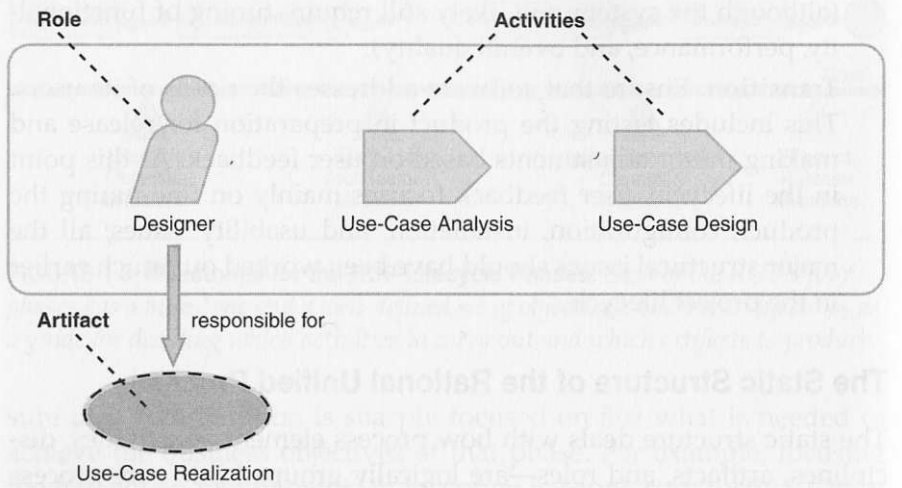
Arlowin & Neustadtin (2002) mukaan ohjelmistotuotantoprosessin mallin tulee kuvata *kuka* tekee *mitä*, *kuinka* ja *milloin*, kuten johdannossa mainitaan. RUP:in kuvaamiseen voidaan käyttää viittä peruselementtiä (Kruchten, 2004; Hull & al., 2002) taulukon 1 mukaisesti.

Taulukko 1: RUP:in elementit.

Elementti	*Merkitys*
Roolit (roles)	Kuka
Tuotokset (artifacts)	Mitä
Toiminnot (activities)	Kuinka
Työnkulut (workflows)	Milloin
Käytännöt (disciplines)	"Säiliöitä" muille elementeille

Kuvassa 11 kuvataan näistä kolme ensimmäistä ja niiden yhteys toisiinsa. Kuvasta näkyy, että roolissa toimiva tekijä suorittaa toimintoja, joiden seurauksena syntyy tuo-

toksia.



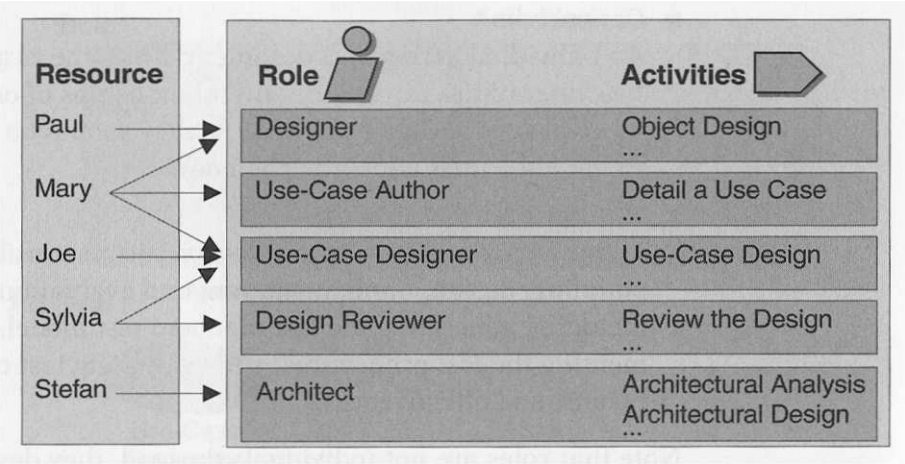
Kuva 11: RUP:in roolit, toiminnot ja tuotokset (Kroll & Kruchten, 2003).

3.1.1 Roolit

Prosessin keskeisimpiin käsitteisiin lukeutuu roolin käsite (Pollice & al., 2004; Kruchten, 2004). Rooli määrittää yksittäisen työntekijän tai kokonaisen tiimin käyttäytymismallin ja vastuut. Roolin käyttäytyminen ilmaistaan sen kautta mitä toimintoja rooli suorittaa. Tietyn roolin vastuut kuvataan yleensä suhteessa tuotoksiin, joita rooli tuottaa, muokkaa tai kontrolloi. Rooleja ei käsitellä yksittäisinä henkilöinä, mutta ei toisaalta myöskään työnimikkeinä (Kruchten, 2004; Hull & al., 2002). Roolituksesta eri vaiheissa päättää projektipäällikkö projektia suunniteltaessa. Roolit tulisi jakaa työntekijöiden reaalisten ominaisuuksien mukaan. Tällainen jaottelu mahdollistaa eri henkilöiden toimimisen useissa eri rooleissa ja toisaalta roolien jakautumisen eri henkilöille, kuten kuva 12 osoittaa.

3.1.2 Toiminnot

Rooleille määrätyt toiminnot määrittävät työn, jota rooli suorittaa (Pollice & al., 2004). Toiminto on tietyssä roolissa toimivan henkilön suorittaman työn yksikkö. Sen tuloksena syntyy tuotos, joka on merkityksellinen projektin etenemiselle. Toiminnolla on selkeä päämäärä, joka yleensä koskee tuotosten, kuten malli, luokka tai suunnitelma, luomista tai muokkaamista (Kruchten, 2004). Yksittäinen toiminto kestää tyypillisesti



Kuva 12: Henkilöt eri rooleissa (Kruchten, 2004).

muutamasta tunnista muutamaan päivään. Se kuuluu yleensä yhdelle tietyssä roolissa toimivalle ihmiselle ja käsittää vain yhden tai muutamia pieniä tuotoksia. Liian pienet toiminnot voidaan hylätä ja liian suuret pilkkoa osiksi (Hull & al., 2002; Kruchten, 2004). Toimintoja voidaan kohdistaa samaan tuotokseen useaan kertaan varsinkin iteraatiosta toiseen, kun järjestelmä tarkentuu ja laajentuu. Toistuvia toimintoja suorittaa sama rooli, muttei välttämättä sama yksilö (Kruchten, 2004).

3.1.3 Tuotokset

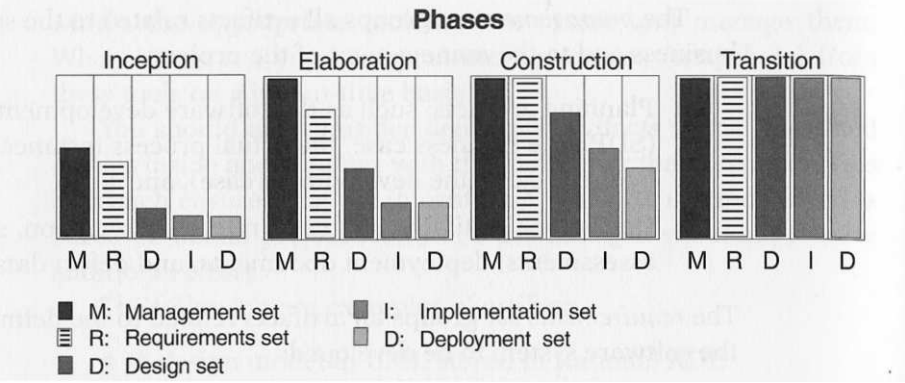
Tuotokset ovat tietoa, jota prosessissa tuotetaan, muokataan tai käytetään. Kruchtenin (2004) mukaan tuotoksia ovat esimerkiksi kaaviot, kuten käyttötapaus- tai suunnittelukaaviot tai niiden yksittäiset elementit, kuten yksittäiset käyttötapaukset. Lisäksi esimerkiksi dokumentit, lähdekoodi sekä ajettava järjestelmä lukeutuvat tuotoksiin. Tuotokset voivat koostua aiemmista tuotoksista. Esimerkiksi suunnittelumalli (design model) voi sisältää useita luokkia. Tuotokset ovat useimmiten versionhallinnan sekä kokoonpanonhallinnan alaisia. Joskus ei voida versioida jotakin yksittäistä tuotosta, kuten esimerkiksi yksittäistä luokkaa, jolloin versioidaan säiliö, johon tuotos sisältyy, esimerkiksi suunnittelumalli (Kruchten, 2004). Tuotokset voidaan jakaa kuuluviksi viiteen kategoriaan taulukon 2 mukaisesti (Hull & al., 2002; Kruchten, 2004).

Monissa prosesseissa annetaan ylisuuri painoarvo dokumenteille ja eritoten paperidokumenteille. RUP ei kannusta paperidokumenttien tuottamiseen. Kruchtenin (2004) mukaan tehokkain ja käytännöllisin tapa projektin tuotosten hallinnoimiseen on se, et-

Taulukko 2: RUP:in tuotosjoukot.

Tuotosjoukko	*Merkitys*
Johtamisjoukko (management set)	Liiketoimintaan ja johtamiseen liittyvät tiedot
Vaatimusjoukko (requirements set)	Järjestelmän määritykset
Suunnittelujoukko (design set)	Järjestelmän kuvaus
Toteutusjoukko (implementation set)	Lähdekoodi, ajettava järjestelmä
Käyttöönottojoukko (deployment set)	Toteutusjoukkoon liittyviä tuotoksia, kuten käyttäjille suunnatua dokumentaatiota

tä ne ovat automaattisesti luotavissa joillakin työvälineillä ja ne tuotetaan vain tarvittaessa. Iteratiivisessa kehitysprosessissa taulukon 2 tuotosjoukot kehittyvät ja kasvavat koko kehityssyklin ajan. Kuvassa 13 on kuvattu tuotosjoukkojen kasvua yhden syklin eri kehitysvaiheiden aikana.



Kuva 13: Tuotosjoukkojen kasvu (Kruchten , 2004).

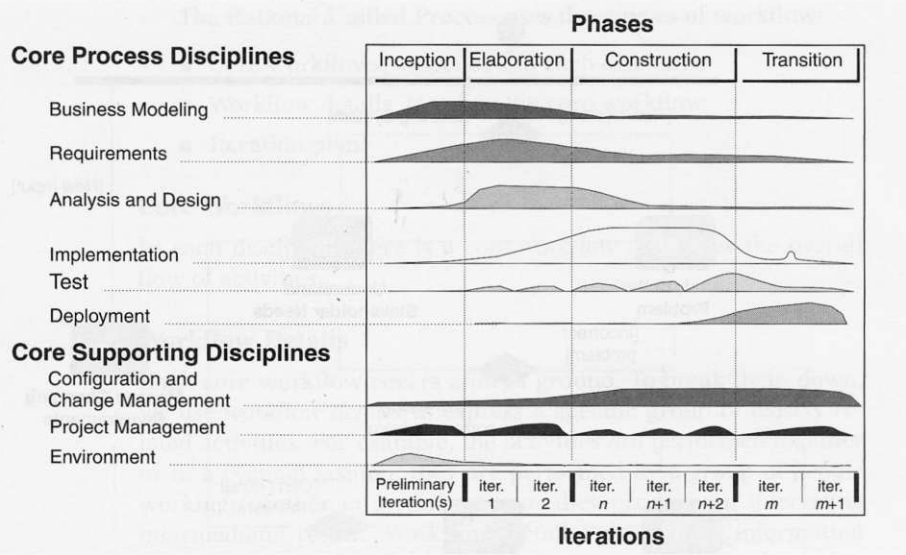
3.1.4 Käytännöt

Käytäntöjä tarvitaan organisoimaan projektin toiminnot. RUP:issa on yhdeksän pääkäytäntöä (Kroll & Kruchten, 2003), jotka esittävät roolit ja toiminnot loogisina ryhminä. Nämä yhdeksän käytäntöä Kruchten (2004) jakaa edelleen kuuteen tekniseen ja kolmeen avustavaan käytäntöön taulukon 3 mukaisesti.

Taulukko 3: RUP:in käytännöt.

Tekniset	*Avustavat*
Liiketoiminnan mallinnuskäytäntö	Projektinhallintakäytäntö
Vaatimuskäytäntö	Kokoonpanon- ja muutosten-hallinnan käytäntö
Analyysi- ja suunnittelukäytäntö	Ympäristökäytäntö
Toteutuskäytäntö	
Testauskäytäntö	
Käyttöönottokäytäntö	

Vaikka tekniset käytännöt muistuttavat nimiltään vesiputousmallin mukaisia peräkkäisiä vaiheita, niin iteratiiviseen prosessin peräkkäiset vaiheet ovat erilaisia ja käytännöt sekä niiden sisältämät työnkulut ovat jatkuvan muutoksen alla koko elinkaaren ajan (Kruchten, 2004). Kuvassa 14 kuvataan ydinkäytännöt sekä niiden työmääriä eri vaiheissa ja eri iteraatioissa.



Kuva 14: Yhdeksän ydinkäytäntöä (Kruchten, 2004).

3.1.5 Työnkulut

Työnkulku on sarja toimintoja, joiden seurauksena syntyy jokin tarkasteltavissa oleva tulos. Työnkululla esitetään myös roolien keskinäistä vuorovaikutusta. UML-kielellä

työnkulku voitaisiin kuvata sekvenssi-, yhteistyö- tai toimintokaaviona (Kruchten, 2004; Hull & al., 2002).

RUP määrittelee kolmenlaisia työnkulkuja. *Ydintyönkulut* (core workflows) liittyvät tiettyyn käytäntöön. Lisäksi ovat *työnkulun yksityiskohdat* (workflow details), jotka tarkentavat ydintyönkulkua sekä *iteraatiosuunnitelmat* (iteration plans), joiden tarkoituksena on kuvata prosessi tyypillisessä iteraatiossa tapahtuvien asioiden näkökulmasta (Kruchten, 2004).

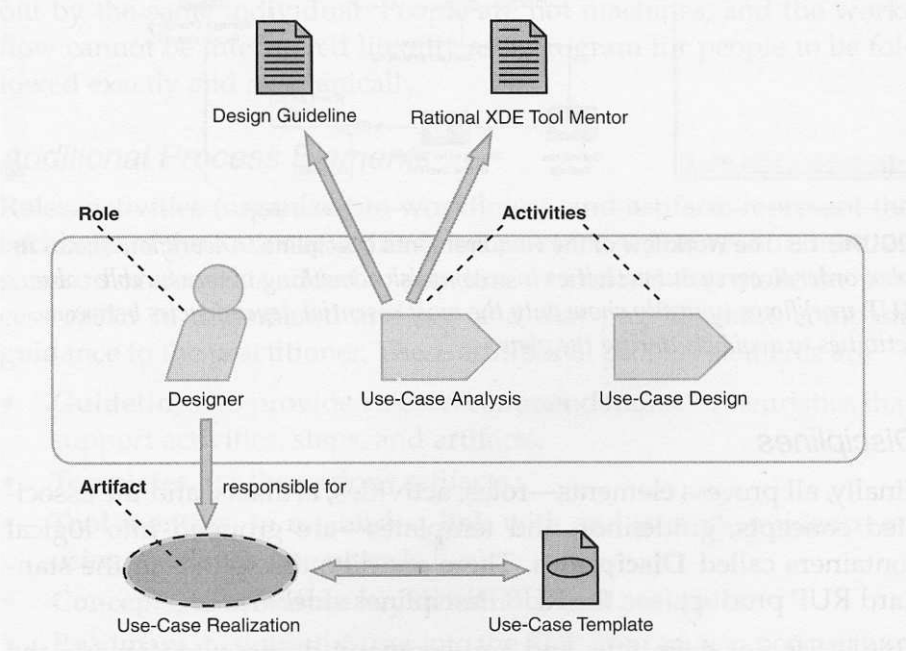
Jokaiseen käytäntöön liittyy yksi ydintyönkulku, joka käsittää yleisen toimintovuon. Kukin ydintyönkuluista on hyvin laaja. Työnkulun yksityiskohtia käytetään pilkkomaan ydintyönkulut osiin ja esittämään tietty ryhmä toisiinsa jollakin tavalla liittyviä toimintoja. Työnkulun yksityiskohdat käsittävät myös toimintoihin liittyviä tietovirtoja, joilla esitetään toimintojen vuorovaikutusta yhteisten tuotosten kautta. Iteraatiosuunnitelmia voidaan tarkastella prosessin ilmentyminä yhden tietyn iteraation osalta, joissa valitaan tietyt iteraation aikana suoritettavat toiminnot. RUP sisältää muutamien tyypillisten iteraatiosuunnitelmien kuvaukset valmiina (Kruchten, 2004).

3.1.6 Prosessin lisäelementit

Prosessin ymmärtämisen ja käyttämisen helpottamiseksi toimintoihin ja tuotoksiin tarjotaan lisäelementtejä. Näitä tarvitaan, jotta voitaisiin antaa paremmin ymmärrettävä opastus prosessin käyttöön. Lisäelementtejä ovat (Hull & al., 2002; Kroll & Kruchten, 2003) *ohjeet* (guidelines), *kaavaimet* (templates), *työvälinementorit* (tool mentors) sekä *käsitteet* (concepts).

Kuvassa 15 on kuvattu kaavaimien, työvälinementoreiden ja ohjeiden suhde rooleihin, toimintoihin ja tuotoksiin.

Ohjeet liittyvät toimintoihin, askeliin tai tuotoksiin. Ne ovat sääntöjä, suosituksia tai heuristiikkoja (Kroll & Kruchten, 2003), jotka auttavat askelien ja toimintojen suorittamisessa. Ne kuvaavat hyvinmuodostettuja tuotoksia keskittyen niiden erityisominaisuuksiin, kuten hyvän luokan tai käyttötapauksen tunnistamiseen. Ohjeet kuvaavat myös tekniikoita, joilla voidaan valmistaa jokin tietty tuotos, muuntaa tuotos muodosta toiseen tai käyttää UML:ää. Ohjeita voidaan käyttää myös tuotosten laadun arviointiin tarkastuslistoina. Lisäksi ohjeita voidaan käyttää toimintojen arvioimiseen (Kruchten,



Kuva 15: Kaavaimet, työvälinementorit ja ohjeet (Kroll & Kruchten, 2003).

2004).

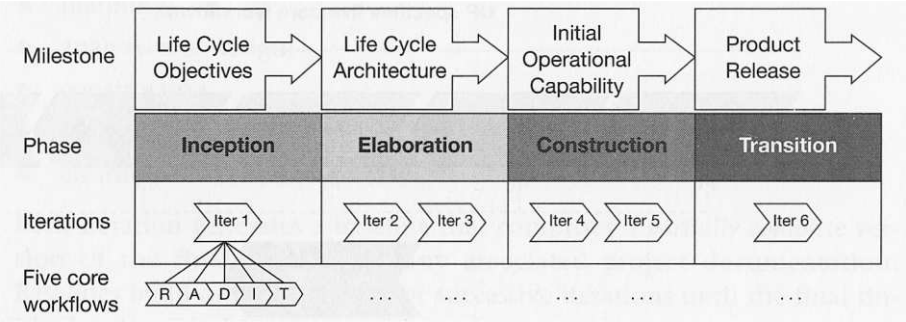
Kaavaimet ovat tuotosten malleja tai prototyyppejä (Kruchten, 2004). Tuotoksen kuvaukseen saattaa liittyä kaavaimia, joita voidaan käyttää luomaan kyseinen tuotos. Kaavaimet on sidottu käytettävään työkaluun. Kaavaimiin voi lukeutua esimerkiksi MS-Word pohjia dokumenteille ja raporteille tai HTML-pohjia eri käyttötarkoituksiin. RUP tarjoaa valmiita kaavaimia, mutta niiden muokkaaminen on tavallista. Bergströmin & Råbergin (2004) mukaan niihin lisätään yleensä vähintään organisaation logo.

Työvälinementorit osoittavat kuinka jokin tietty toiminto suoritetaan tietyllä työvälineellä (Kruchten, 2004). Käsitteitä ovat esimerkiksi iteraatio, vaihe ja tuotos.

Tässä kohdassa esitellyt asiat muodostavat *prosessikehyksen* (process framework). Rooleja, tuotoksia, toimintoja, ohjeita, käsitteitä ja mentoreita lisäämällä tai poistamalla voidaan muokata tai sopeuttaa prosessi organisaation tarpeisiin (Kruchten, 2004).

3.2 Iteratiivinen kehitys

RUP:in yksi kehityssykli koostuu neljästä *vaiheesta* (phase) (Jacobson & al., 1999; Hull & al., 2002; Kruchten, 2004). Vaiheet ovat kuvan 16 mukaisessa järjestyksessä *aloitus* (inception), *kehittely* (elaboration), *rakentaminen* (construction) ja *siirtyminen* (transition). Jokaisen vaiheen päättää suuri *virstanpylväs* (milestone). Jokaisen virstanpylvään kohdalla projektin johto voi tehdä päätöksiä projektin jatkamisesta, lopettamisesta tai suunnasta (Arlow & Neustadt, 2002; Kruchten, 2004).



Kuva 16: Vaiheet ja virstanpylväät (Arlow & Neustadt, 2002).

Aloitusvaiheessa selvitetään oleelliset vaatimukset järjestelmän rajaamiseksi ja toteutettavuus sekä projektin kannattavuus ja kriittiset riskit. Toteutettavuuden selvittäminen saattaa vaatia prototyypin rakentamisen. Aloitusvaiheen päättävästä virstanpylväästä käytetään nimitystä *elinkaaritavoitteet* (Life Cycle Objectives, LCO)

Kehittelyvaiheessa luodaan suorituskelpoinen arkkitehtoninen perustaso (baseline) sekä tarkennetaan resurssisuunnitelmaa ja riskien arvioimista. Lisäksi määritellään laatuattribuutit, kuten sallittava puutetiheys. Tavoitteena on myös arkkitehtuurin sisältämän toiminnallisuuden määrittäminen käyttötapausten avulla sekä rakentamisvaiheen suunnitelman laatiminen. Vaiheen päättävästä virstanpylväästä käytetään nimitystä *elinkaariarkkitehtuuri* (Life Cycle Architecture, LCA).

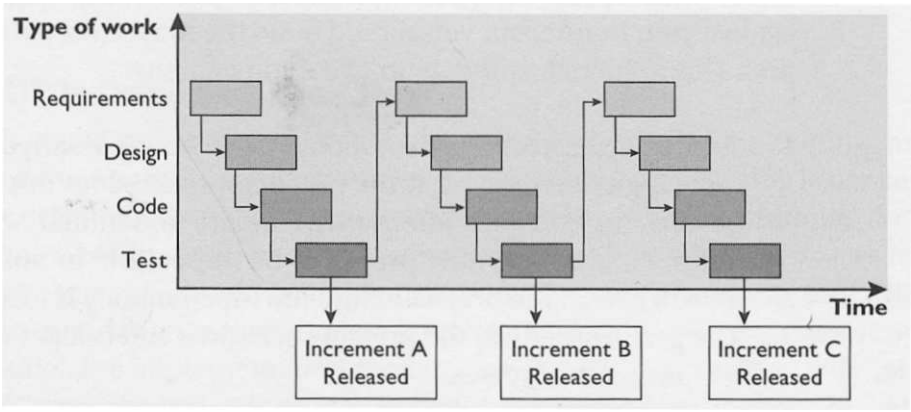
Rakentamisvaiheessa toteutetaan kaikki vaatimukset ja kehitetään arkkitehtonista perustasoa lopulliseksi järjestelmäksi. Vaiheen päättävästä virstanpylväästä käytetään nimitystä *lähtötoimintakyky* (Initial Operational Capability, IOC).

Siirtymisvaihe alkaa, kun beta-testaus on valmis ja puutteet korjattu. Ohjelmisto sijoitetaan käyttäjien koneille, räätälöimällä tarvittaessa. Vaiheeseen kuuluu myös mahdollisesti rinnakkainen toiminta vanhan järjestelmän kanssa, operatiivisten tietokan-

tojen muuntaminen ja käyttäjien konsultointi. Vaiheen päättävästä virstanpylväästä käytetään nimitystä *tuotejulkistus* (Product Release, PR)

Edellä mainitut neljä vaihetta muodostavat Kruchtenin (2004) mukaan *kehityssyklin* (development cycle) ja niiden tuloksena syntyy *ohjelmistosukupolvi* (software generation). Järjestelmä tuotetaan *alkukehityssyklissä* (initial development cycle). Seuraavia syklejä kutsutaan *evoluutiosykleiksi* (evolution cycles) ja myös ne koostuvat edellä luetelluista neljästä vaiheesta (Kroll & Kruchten, 2003; Kruchten, 2004). Evoluutiosykleissä tosin eri vaiheiden painotukset saattavat olla erilaiset, kuin alkukehityssyklissä. Jokaisen toteutuneen evoluutiosyklin seurauksena syntyy uusi ohjelmistosukupolvi. Käytännössä eri syklit voivat olla osittain päällekkäisiä. Seuraavan syklin aloitus- ja kehittelyvaiheet voivat käynnistyä jo edellisen syklin siirtymävaiheen ollessa vielä käynnissä (Kruchten, 2004). Eri vaiheet saattavat puolestaan eri sykleissä olla hyvinkin eripituisia. Olosuhteet, joissa kutakin projektia viedään läpi, vaikuttavat eri vaiheiden kestoihin.

Kukin vaihe koostuu kuvan 14 mukaisesti yhdestä tai useammasta iteraatiosta. Kukin iteraatio tuottaa *lisäyksen* (increment) kuvan 17 periaatteiden mukaisesti. Lisäys on kahden iteraation tuottamien sisäisten tai ulkoisten *julkistusten* (release) välinen ero.



Kuva 17: Iteraatiot ja julkistukset (Bergström & Råberg, 2004).

Kunkin iteraation työnkulku voi siis sisältää vaatimusten määrittelyä ja analysointia, suunnittelua ja toteutusta sekä integrointia ja testausta (Hull & al., 2002). Iteraatiosta toiseen eri toimintotyyppien painoarvo kuitenkin muuttuu. Aloitusvaiheessa pääpaino on vaatimusten ymmärtämisessä ja kehityspanosten kattavuudessa, siirtymävaiheessa taas sen varmistamisessa, että järjestelmän laatu täyttää vaatimukset (Kruchten, 2004).

3.3 Arkkitehtuurikeskeinen prosessi

Jacobsonin & al. (1999) mukaan järjestelmän arkkitehtuuri voidaan ajatella yhteiseksi visioksi, joka kaikkien työntekijöiden on hyväksyttävä. Arkkitehtuuri antaa selkeän perspektiivin koko järjestelmään, mikä on välttämätöntä järjestelmän rakentamisen kontrolloimiseksi. Arkkitehtuuria tarvitaan kuvaamaan ne mallin alkiot, jotka ovat kehittäjille kaikkein tärkeimmät: jotkin alijärjestelmät, riippuvuudet, rajapinnat, yhteistyöt, solmut ja aktiivit luokat eli prosessit tai säikeet. RUP määrittelee arkkitehtuurin Garlanin & Shaw'n (1996) mukaisesti käsittävän tärkeitä päätöksiä seuraavista (Kruchten, 2004; Jacobson & al., 1999):

- ohjelmistojärjestelmän organisointi
- järjestelmän sisältämien rakenteellisten elementtien ja niiden rajapintojen valinta, elementtien välisen yhteistyön määrittämä käyttäytyminen
- elementtien ryhmittäminen alijärjestelmiksi
- arkkitehtoninen tyyli, joka ohjaa organisointia.

Ohjelmistoarkkitehtuuriin liittyy rakenteen ja käyttäytymisen lisäksi myös erilaisia rajoitteita käytön, toiminnallisuuden, suorituskyvyn, joustavuuden, uudelleenkäytettävyyden, käsitettävyyden, taloudellisuuden ja teknologian suhteen.

Taulukko 4: Arkkitehtoniset näkymät.

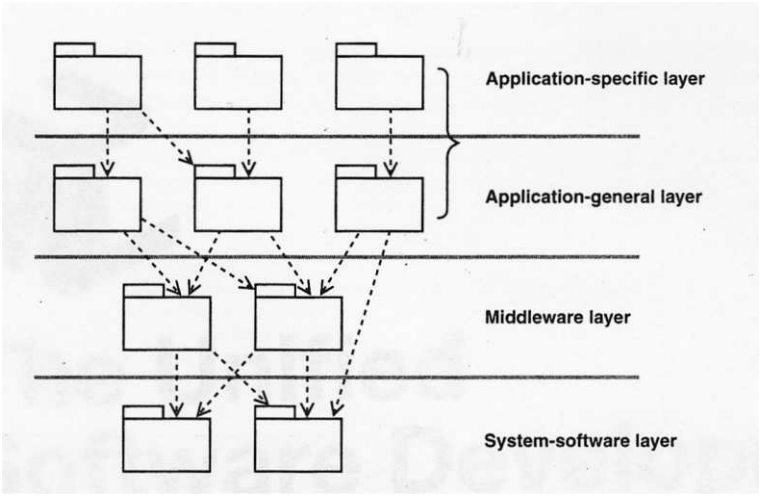
Näkymä	*Selitys*
Looginen näkymä	Järjestelmän toiminnalliset vaatimukset: paketit, alijärjestelmät, luokat, rajapinnat
Prosessinäkymä	Järjestelmän suorituskyky ja skaalattavuus
Toteutusnäkymä	Järjestelmän kokoonpano ja kokoonpanonhallinta
Käyttöönottonäkymä	Järjestelmän topologia, jakelu, toimitus ja asennus
Käyttötapausnäkymä	Käyttötapaukset, "sitoo" yhteen muut näkymät

Järjestelmän ymmärtämistä edesauttavat erilaiset näkökulmat järjestelmän malleista. RUP sisältää viiden arkkitehtonisen näkökulman lähestymistavan mallinnukseen taulukoiden 4 ja 5 mukaisesti.

Taulukko 5: Mallien ja näkymien suhde (Kruchten, 2004).

Malli	*Näkymä*
Suunnittelumalli	Looginen näkymä
Suunnittelumalli (prosessimalli monimutkaisille järjestelmille)	Prosessinäkymä
Toteutusmalli	Toteutusnäkymä
Käyttöönottomalli	Käyttöönottonäkymä
Käyttötapausmalli	Käyttötapausnäkymä

Kun kaikki järjestelmää eri näkökulmista kuvaavat mallit kootaan yhteen, saadaan järjestelmän *arkkitehtuuri*. Mitä monimutkaisempi on projekti, sen tarpeellisempi hyvin määritelty arkkitehtuuri on. Mitä paremmin arkkitehtuuri on määritelty, sitä helpompaa on reagoida muutoksiin järjestelmässä ilman, että koko järjestelmä täytyy rakentaa alusta asti uudestaan. Kruchten (2004) väittää huonon arkkitehtuurin, joka usein yhdistyy huonoon prosessiin, olevan usein syynä ohjelmistoprojektien kaatumiselle. Myös Bergström & Råberg (2004) myöntävät suurimpien teknisten riskien kohdistuvan nimenomaan arkkitehtuuriin. Pollicen & al. (2004) ja Jacobsonin & al. (1999) mukaan arkkitehtuuri voi kuitenkin olla asteittain tarkentuva ja kehittyvä. Tyyliltään kerrostettu arkkitehtuuri järjestää järjestelmät alijärjestelmäkerroksiin kuvan 18 mukaisesti.



Kuva 18: Kerrostettu arkkitehtuuri (Jacobson & al., 1999).

RUP tukee *komponenttipohjaista kehitystä* (component-based development). Tällä tarkoitetaan sitä, että toisaalta rakennetaan järjestelmiä, jotka koostuvat pienemmistä

komponenteista ja toisaalta itse komponentteja. Komponenttipohjaisen kehittämisen pohjana on tarve kehittää laadukkaita järjestelmiä nopeasti. Tähän päästään helpoiten kokoamalla järjestelmät valmiista osista, mieluummin kuin valmistamalla jokainen osanen yksittäin. Kruchten (2004) tuo esille myös arkkitehtonisten mallien, kuten MVC (Model-View-Controller) ja ORB (Object Request Broker), käytön.

3.4 Käyttötapauspohjainen prosessi

Ongelman mallintamiseen ja vaatimusten ilmaisemiseen on olemassa monia vaihtoehtoisia ratkaisutapoja. RUP suosittelee ongelman mallintamiseen käytettävän *käyttötapausten mallintamista* (Bergström & Råberg, 2004; Kruchten, 2004). Ongelman ilmaiseminen käyttötapausten avulla auttaa sen ymmärtämisessä kaikkia sidosryhmiä.

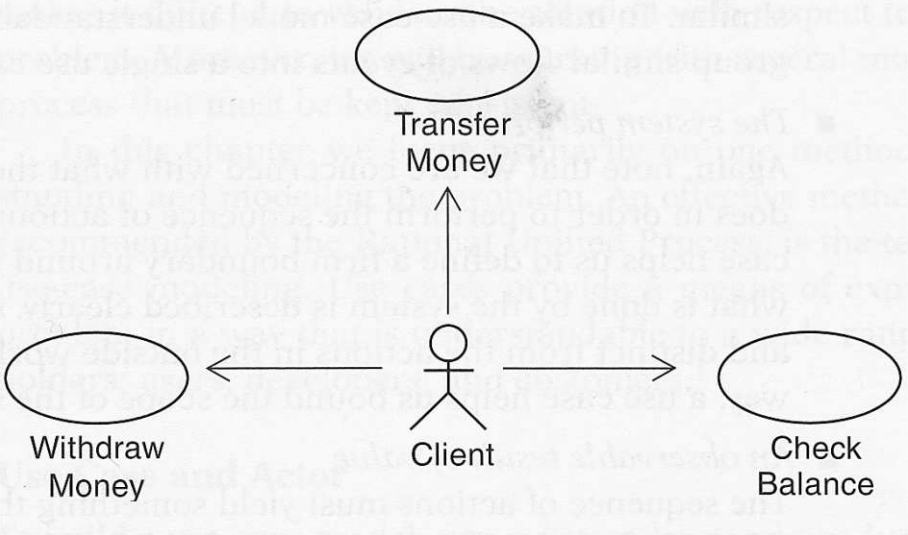
Käyttötapausten mallintamisessa tarvitaan kahta käsitettä: *käyttötapausta* ja *toimijaa*. Käyttötapauksella tarkoitetaan sarjaa toimenpiteitä, jotka järjestelmä suorittaa saavuttaakseen vaaditun tuloksen, jolla on käyttöarvoa toimijalle. Toimijan puolestaan määritellään olevan joku tai jokin järjestelmän ulkopuolinen, joka on vuorovaikutuksessa järjestelmän kanssa (Bergstöm & Råberg, 2004; Kruchten, 2004).

Käyttötapauksen kuvaus kertoo, mitä järjestelmä tekee, kun tietty käyttötapaus suoritetaan. Järjestelmän toiminnallisuuden määrittelee joukko käyttötapauksia, joista kukin esittää yhtä tiettyä tapahtumavuota. Kuva 19 esittää pankkiautomaatin ja toimijan väliset käyttötapaukset, joissa toimija voi nostaa tai siirtää rahaa tai tarkistaa tilin katteen.

3.4.1 Tapahtumavuo

Vaatimuskäytännön osalta käyttötapauksen tärkein osa on *tapahtumavuo* (flow of events). Se kertoo mitä toimenpiteitä tapahtuu järjestelmän ja toimijan välillä. Se kirjoitetaan luonnollisella kielellä käyttäen yksinkertaista kieltä ja yhteisesti hyväksyttyä terminologiaa. Kruchten (2004) tarjoaa esimerkin tapahtumavuoksi kuvan 19 esittämälle käyttötapaukselle rahan nostamiseksi seuraavasti:

1. Käyttötapaus alkaa, kun asiakas syöttää kortin pankkiautomaattiin. Järjestelmä lukee ja validoi kortin tiedot.



Kuva 19: Käyttötapaukset ja toimija (Kruchten, 2004).

2. Järjestelmä pyytää PIN-koodia. Asiakas syöttää koodin. Järjestelmä validoi PIN-koodin.
3. Järjestelmä pyytää asiakasta valitsemaan suoritettavan toiminnon. Asiakas valitsee rahan nostamisen.
4. Järjestelmä kysyy nostettavan summan määrää. Asiakas syöttää summan.
5. Järjestelmä kysyy tilin tyyppiä. Asiakas valitsee tilin tyypin.
6. Järjestelmä kommunikoi pankkiautomaattiverkon kanssa ja validoi tilin ID:n, PIN-koodin ja ettei pyydetty summa ylitä tilin saldoa.
7. Järjestelmä kysyy asiakkaalta haluaako tämä kuitin. Tämä toiminto suoritetaan vain, jos automaatissa on paperia kuitin tulostamista varten.
8. Järjestelmä pyytää asiakasta poistamaan kortin. Asiakas poistaa kortin. (Pyyntö on turvallisuustekijä, jotta asiakkaat eivät jättäisi korttejaan koneeseen.)
9. Järjestelmä antaa pyydetyn rahasumman.
10. Järjestelmä tulostaa kuitin, jos sitä pyydettiin. Käyttötapaus päättyy.

3.4.2 Skenaariot

Käyttötapauksen tapahtumavuon kukin haarautuma generoi erillisen skenaarion eli polun käyttötapauksen läpi (Arlow & Neustadt, 2002). Polkujen ryhmittely määrittelee *käyttötapausluokan*. Käyttötapausluokan ilmentymä on siis skenaario. Toisin sanoen skenaario on yksi vaihtoehtoisista käyttötapauksen läpi johtavista poluista. Yleisesti katsotaan helpommaksi viitata käyttötapausluokkaan nimellä käyttötapaus ja käyttötapauksen ilmentymään nimellä skenaario (Kruchten, 2004).

Skenaarioita käytetään prosessissa jonkin tietyn käyttötapauksen läpi etenevän tietyn toimenpidesekvenssin tai "säikeen" korostamiseksi ja erittelemiseksi. Testitapauksia määriteltäessä tämä on erittäin käyttökelpoista. Samoin yritettäessä etsiä käyttötapauksia projektin alkuvaiheissa, on usein helpompaa aloittaa hyvin ominaisesta käyttötapauksen kuvauksesta ja laajentaa sitä käsittämään useampia tapahtumapolkuja, kunnes se on laajennettu käsittämään kaikki mahdolliset polut (Kruchten, 2004).

3.4.3 Käyttötapausmalli

Käyttötapausmalli koostuu kaikista järjestelmän tai sen osan käyttötapauksista sekä toimijoista. Se kuvaa koko järjestelmän toiminnan. Se toimii mallina järjestelmän toteutettavista toiminnoista ja ympäristöstä, jossa järjestelmä toimii. Sitä voidaan myös käyttää sopimuksena asiakkaan ja kehittäjien välillä. RUP:issa käytetään käyttötapausmallin kuvaamiseen kuvan 19 mukaista UML-kielen käyttötapauskaaviota sekä toimintokaavioita (Kruchten, 2004). Toimintokaaviolla voi korvata tekstimuotoisen tapahtumavuon.

3.4.4 Käyttötapausten kehittäminen

Käyttötapauksen suunnittelu pitäisi aina aloittaa laatimalla askelmuotoinen käyttötapausluonnos menemättä tarkempiin yksityiskohtiin. Kehittelyvaiheen aikaisissa iteraatioissa vain muutamat, arkkitehtonisesti tärkeimmät käyttötapaukset kuvataan yksityiskohtaisesti. Usein käyttötapausmalli sisältää käyttötapauksia, jotka ovat niin yksinkertaisia, ettei niiden kuvaamiseen tarvita yksityiskohtaista tapahtumavuon kuvausta. Tällöin riittää, että kuvataan käyttötapauksen läpivienti pääpiirteittäin askel askeleelta (Kruchten, 2004).

3.4.5 Käyttötapausten organisointi

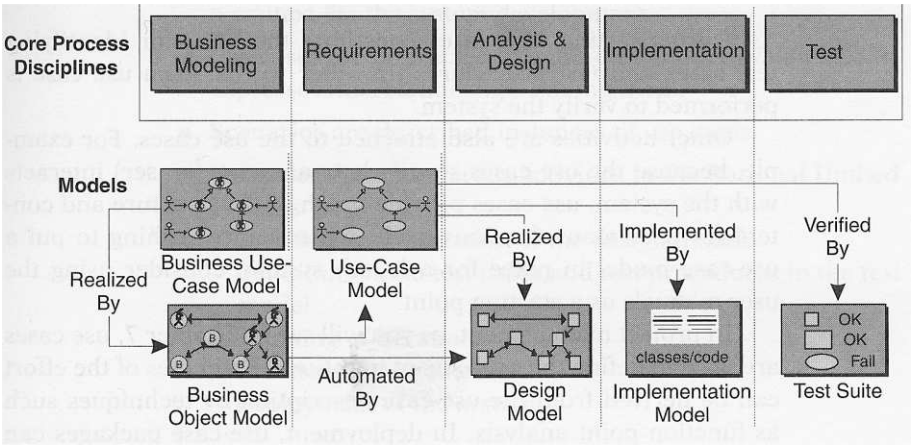
Pieni järjestelmä voidaan helposti kuvata vain muutamien käyttötapausten ja toimijoiden avulla. Isompien järjestelmien ollessa kyseessä, on kuitenkin syytä kiinnittää huomiota rakenteellisuuteen ja järjestykseen. Muuten käyttötapausten määrä ja organisoimattomuus saattaa johtaa siihen, että vaatimuksista tulee hankalia ymmärtää mikä vaikeuttaa muita toimintoja, kuten suunnittelua, priorisointia ja estimointia (Kruchten, 2004).

Käyttötapauspaketti on ensimmäinen rakenteellisuuteen vaikuttava käsite. Siinä toisiinsa liittyvät käyttötapaukset ryhmitellään yhdeksi kokonaisuudeksi. Käyttötapausten polkuja analysoimalla organisointia voidaan tarkentaa sisältyvyyden, laajentamisen ja erikoistamisen avulla (Kruchten, 2004). Eri käyttötapauksissa voi olla yhteisiä polkuja, jotka voidaan kuvata näihin käyttötapauksiin sisältyvänä käyttötapauksena. Olemassaolevaan käyttötapaukseen voidaan liittää myös laajennuskäyttötapaus tiettyjen ehtojen täyttyessä. Olioparadigman tapaan on mahdollista myös erikoistamalla tuottaa uusia käyttötapauksia.

3.4.6 Käyttötapaukset prosessissa

Käyttötapaukset ovat koko kehitysprosessin perusta kuvan 20 mukaisesti (Jacobson & al., 1999; Kruchten, 2004). Ne toimivat lähtökohtana käyttötapausten realisoinneille, jotka kuvaavat kuinka käyttötapaukset toteutetaan vuorovaikutteisten olioiden kesken suunnittelumallissa. Oliot ja luokat löytyvät todennäköisesti helpoiten tutkimalla käyttötapauksia. Tämä tekniikka varmistaa, että kaikki vaadittu käyttäytyminen ja toiminnallisuus on esitetty suunnittelumallissa.

Toteutuksen aikana suunnittelumalli toimii toteutuksen määrityksenä. Koska käyttötapaukset toimivat suunnittelumallin pohjana, ne toteutetaan suunnitteluluokkina. Testauksen aikana käyttötapaukset muodostavat pohjan testitapauksille ja testausmenetelmille. Myös muut käytännöt ovat sidoksissa käyttötapauksiin. Koska käyttötapaukset määrittävät suurelta osin käyttäjien toiminnan järjestelmän piirissä, niitä voidaan käyttää pohjana käyttäjille suunnatun dokumentaation, kuten käyttöohjeen laatimisessa. Projektin johtamisessa käyttötapauksia käytetään hyödyksi iteraatioiden sisällön suunnittelemisessa. Tekniikoiden, kuten toimintopisteanalyysin (esim. Fenton & Pfleeger, 1997) avulla työmääriä voidaan arvioida käyttötapausten pohjalta (Kruch-



Kuva 20: Käyttötapaukset prosessissa (Kruchten, 2004).

ten, 2004).

3.5 Prosessikäytännöt

Roolit, toiminnot ja tuotokset ovat RUP:in peruselementtejä. Prosessikäytännöt, jotka esitellään tämän kohdan alakohdissa, puolestaan määrittävät, mitkä roolit osallistuvat mihinkin toimintoon ja mitä tuotoksia tuotetaan. Käytännöt ovat siis roolien, toimintojen, tuotosten ja muiden tarpeellisten elementtien loogisia ryhmiä (Kroll & Kruchten, 2003), jotka tarjoavat kontekstin toiminnalle.

3.5.1 Projektinhallintakäytäntö

RUP:in projektinhallintakäytäntö tarjoaa neuvontaa ja ohjenuoria projektin hallintaan, kuten miehitykseen, suorittamiseen, seurantaan sekä riskienhallintaan liittyen. Käytäntö ei kata kaikkia projektinhallintaan liittyviä näkökohtia, vaan keskittyy iteratiivisen prosessin luonteenomaisiin piirteisiin, kuten iteratiivisen projektin koko elinkaaren sekä yksittäisen iteraation suunnitteluun, riskienhallintaan ja projektin edistymisen seurantaan ja mittaamiseen (Kruchten, 2004).

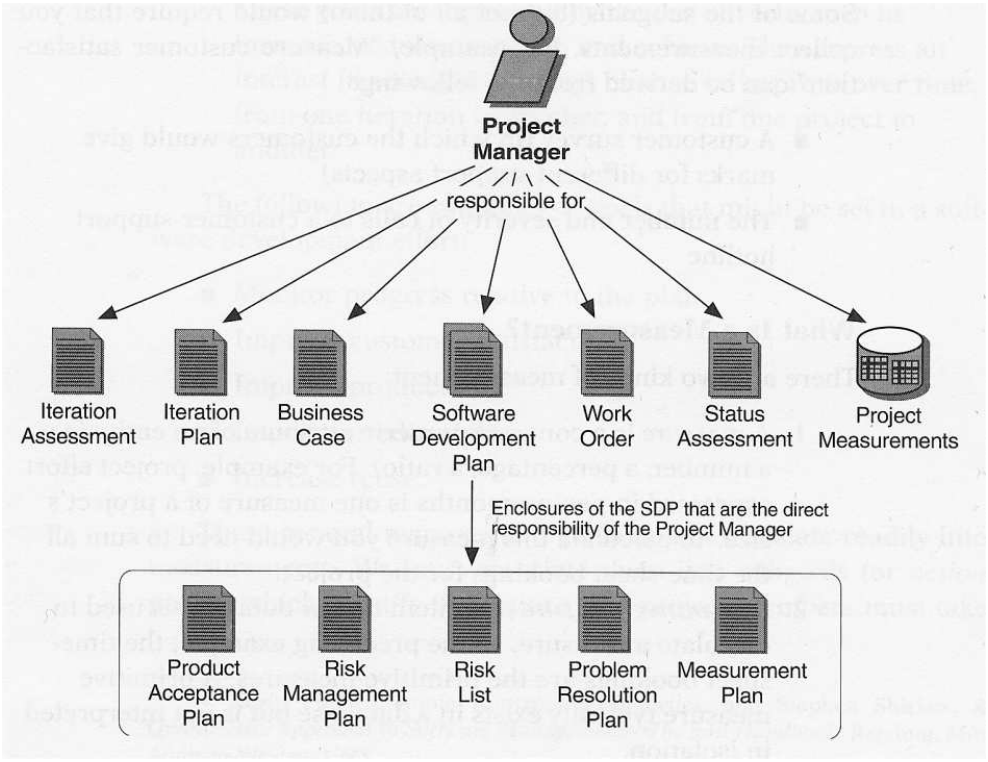
Projektinsuunnittelun päämäärinä ovat esimerkiksi tehtävien ja vastuiden jakaminen henkilöille, edistymisen tarkkaileminen ja mahdollisten ongelmakohtien tunnistaminen. Iteratiivisen projektin suunnitelman tulisi Kruchtenin (2004) mukaan olla kaksitasoinen sisältäen karkearakeisen *vaihesuunnitelman* ja sarjan hienorakeisia *iteraatio-*

suunnitelmia. Vaihesuunnitelma eli projektisuunnitelma, joita on yksi kutakin projektia kohden, sisältää suurien virstanpylväiden päivämäärät, henkilöstön vastuut ja tehtävät eri ajanhetkillä sekä pienempien virstanpylväiden päivämäärät. Vaihesuunnitelma toteutetaan hyvin varhaisessa vaiheessa aloitusvaihetta.

Iteraatiosuunnitelma on hienorakeisempi suunnitelma, jollaisia projektissa on tyypillisesti kaksi aktiivisena yhdellä kertaa. Työn alla olevaa, nykyistä iteraatiota käsittelevä iteraatiosuunnitelma, jota käytetään edistymisen seurantaan, sekä seuraavan iteraation suunnitelma, jonka on oltava valmis nykyisen iteraation loppuun mennessä. Iteraatiosuunnitelma rakennetaan RUP:issa perinteisiä tekniikoita ja työvälineitä, kuten esimerkiksi Gantt-kaaviota käyttäen ja siinä osoitetaan tehtävät ja vastuut yksilöille ja tiimeille.

Riskit vaikuttavat moniin iteratiivisen prosessin elinkaaren aikana tehtäviin suunnittelu- ja toteutuspäätöksiin. Etenkin kehittelyvaiheessa riskien ehkäiseminen tai lieventäminen ovat avainasemassa, mutta myös rakentamisvaiheessa ne saattavat olla merkittävä tekijä. Riskienhallinta huolehtii projektin tuntemattomista uhkista. Riskit ovat joko suoria tai epäsuoria. Suorat riskit ovat sellaisia, joita kohtaan projektissa voidaan vaikuttaa ja epäsuorat sellaisia, joihin projektin kannalta ei voida vaikuttaa tai voidaan vaikuttaa vain vähän. Riskiin liittyy myös kaksi attribuuttia: *todennäköisyys* ja *vakavuus*, jotka kaksi yhdessä muodostavat riskin suuruuden. Jokaista tunnistettua riskiä kohden pitää olla tiedossa, mitä tehdä riskin sattuessa.

Mittauksia suoritetaan RUP:issa projektin hallitsemiseksi, jotta tiedettäisiin, kuinka lähellä tai kaukana ollaan projektin tavoitteista valmistumisen, laadun ja vaatimusten täsmäävyyden suhteen. Toisaalta mittaustulosten perusteella voidaan saavuttaa parempia estimointeja seuraavissa projekteissa. Mittauksella pitää olla selkeät päämäärät. Ainoastaan sellaista tietoa, joka auttaa näiden päämäärien saavuttamisessa, on syytä kerätä. Tavoitteet voivat Kruchtenin (2004) mukaan olla joko tietämyksen lisääminen tai muutosten saavuttaminen. Tietämyksen lisäämisen tavoitteet ilmaisevat tarvetta ymmärtää kehitysprosessia paremmin. Muutosten saavuttaminen edellyttää edistymisen seurantaa tiettynä ajanjaksona, iteraatiosta toiseen ja projektista toiseen. Kuva 21 kuvaa projektinhallintakäytännön tärkeimmän roolin eli *projektipäällikön* sekä tärkeimmät projektinhallintakäytännön tuotokset.



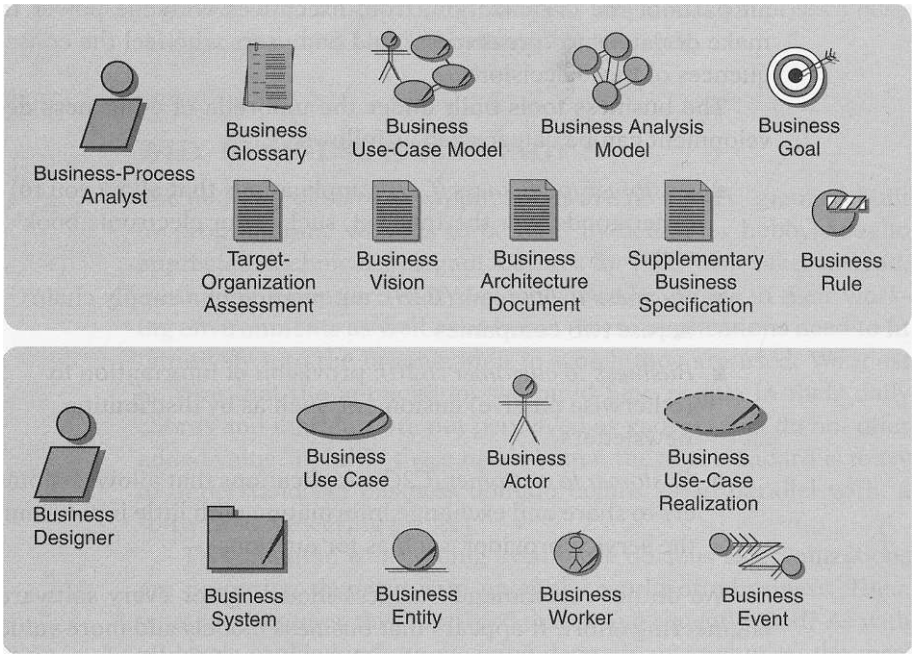
Kuva 21: Rooli ja projektinhallintakäytännön tuotokset (Kruchten, 2004).

3.5.2 Liiketoimintamallinnuskäytäntö

Liiketoimintamallinnuskäytäntö tähtää Kruchtenin (2004) mukaan kohdeorganisaation rakenteen, dynamiikan ja ongelmien ymmärtämiseen sekä parannusmahdollisuuksien tunnistamiseen. Lisäksi tavoitteena on varmistaa, että asiakkailla, loppukäyttäjillä ja kehittäjillä on yhdenmukainen kuva kohdeorganisaatiosta ja lopuksi johtaa kohdeorganisaatiota tukevat järjestelmävaatimukset. Liiketoiminnan mallinnusta Kruchten (2004) ei suosittele jokaiseen ohjelmistoprojektiin. Sellaisten järjestelmien osalta, joilla on useita käyttäjiä ja, jotka käsittelevät suuria määriä tietoa, liiketoimintamallinnuksella saavutetaan suurin hyöty. Myös liiketoimintamallinnuksen kattavuus saattaa vaihdella projektikohtaisesti. Käytettäessä liiketoiminnan ja järjestelmän mallintamisessa samanlaisia tekniikoita helpottuu kuvan 20 ilmaisemien yhteyksien hahmottaminen liiketoimintamallin ja järjestelmän mallin välillä.

Kuvassa 22 on kuvattu liiketoimintamallinnuskäytännön roolit ja tuotokset.

Liiketoimintamallinnuskäytännön rooleja ovat *liiketoimintaprosessin analysoija*, joka johtaa ja koordinoi liiketoimintakäyttötapausten mallintamista hahmottelemalla ja ra-



Kuva 22: Liiketoimintamallinnuskäytännön roolit ja tuotokset (Kruchten, 2004).

jaamalla mallinnettavaa organisaatiota. Toinen rooli on *liiketoimintasuunnittelija*, joka luo yksityiskohtaisen liiketoiminnan käyttötapausmallin ja liiketoiminnan oliomallin, joista voidaan johtaa järjestelmän käyttötapausmalli (Kruchten, 2004).

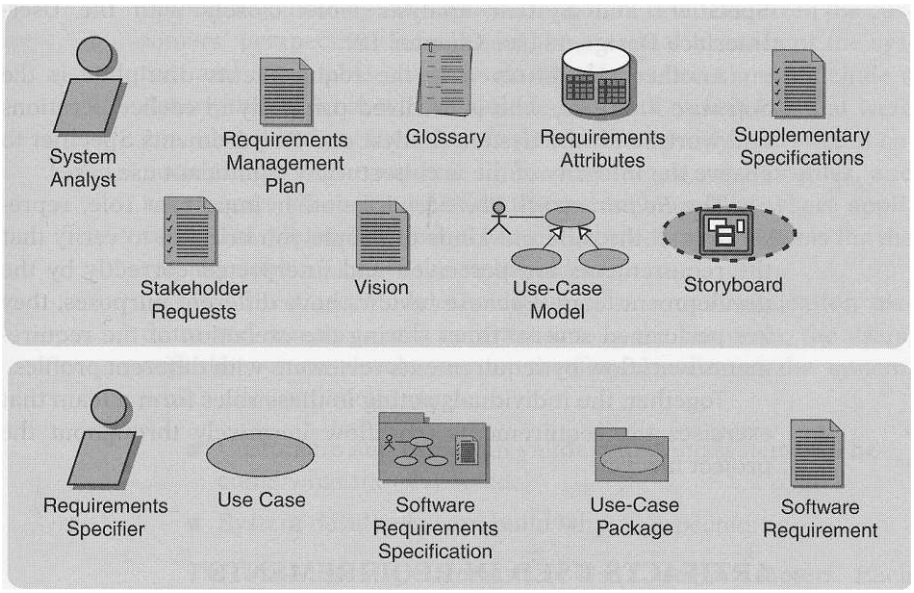
3.5.3 Vaatimuskäytäntö

Vaatimuskäytännön päämäärä on luoda ja säilyttää konsensus asiakkaiden ja kehittäjien välillä järjestelmän vaatimuksista. Samoin se määrittää järjestelmän rajat, tarjoaa perustan iteraatioiden suunnittelulle sekä perustan projektiin kuluvan ajan ja kustannusten arvioimiseksi.

Vaatimukset voidaan jakaa kahteen kategoriaan (Kruchten, 2004): *toiminnalliset vaatimukset* ja *ei-toiminnalliset vaatimukset*. Toiminnallisia vaatimuksia ovat sellaiset vaatimukset, jotka kuvaavat mitä järjestelmä varsinaisesti tekee. Ne määrittävät järjestelmän toiminnan syötteiden ja tulosteiden tarkkudella. Ei-toiminnalliset vaatimukset ovat järjestelmän yleiseen laatuun liittyviä vaatimuksia. Ne voidaan jakaa neljään ryhmään: *käytettävyys-*, *luotettavuus-*, *suorituskyky-* ja *tukivaatimukset* (Kruchten, 2004).

Jotta vaatimuskäytäntöä voidaan hallinnoida tehokkaasti, on arvokasta saada tietoon

paitsi loppukäyttäjien myös muiden sidosryhmien jäsenten, kuten ostajien, alihankkijoiden, kehittäjien tai järjestelmän tulevan ylläpitäjän todelliset tarpeet (Kruchten, 2004). Kuvassa 23 on kuvattu vaatimuskäytännön tärkeimmät roolit ja tuotokset.



Kuva 23: Vaatimuskäytännön roolit ja tuotokset (Kruchten, 2004).

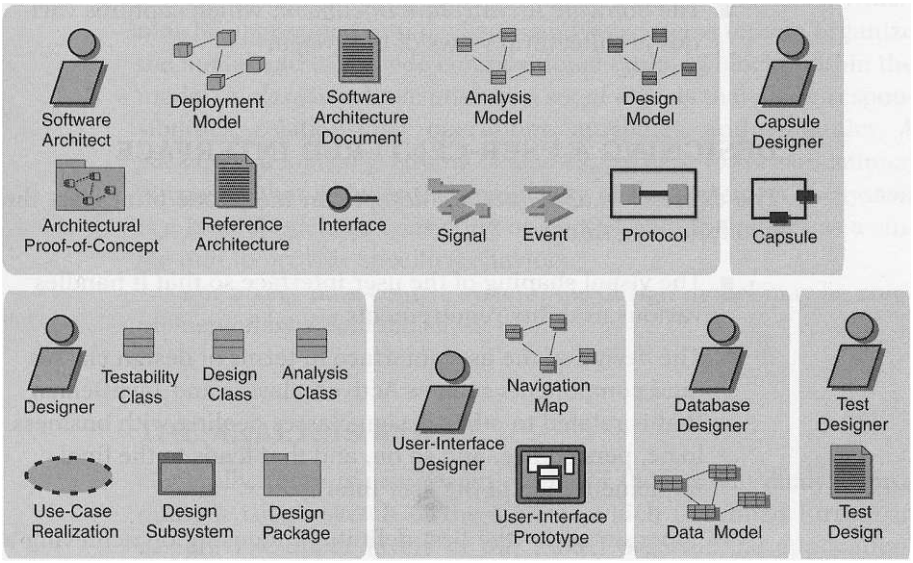
Järjestelmän analysoijan tehtäviin kuuluu vaatimusten keruun ja käyttötapausten mallintamisen johtaminen ja koordinoiminen hahmottelemalla järjestelmän toiminnallisuutta ja rajaamalla järjestelmää. *Vaatimusten määrittelijä* mallintaa yksityiskohtaisesti koko järjestelmän tai osan sen toiminnallisuudesta kuvaamalla käyttötapaukset. Vaatimuskäytäntöön voi roolina liittyä myös *ohjelmistoarkkitehti* (software architect) ja *vaatimusten tarkastaja* (requirements reviewer). Ohjelmistoarkkitehti toimii pääasiassa alkuvaiheen iteraatioissa. Hän toimii yhteistyössä vaatimusmäärittelijöiden ja järjestelmän analysoijan kanssa varmistaakseen ohjelmiston arkkitehtuurin onnistumisen. Vaatimusten tarkastaja on rooli, johon asetetaan kaikki henkilöt, jotka osallistuvat vaatimusten arviointiin. Nämä neljä roolia yhdessä muodostavat tiimin, joka suorittaa vaatimusten työnkulkua iteratiivisesti projektin elinkaaren ajan (Kruchten, 2004).

3.5.4 Analysointi- ja suunnittelukäytäntö

Käytännön tavoitteena on muuntaa järjestelmän vaatimukset määrittelyksi, jonka perusteella järjestelmä voidaan toteuttaa. Tähän pääsemiseksi on ymmärrettävä vaatimukset sekä valittava paras toteutusstrategia. Projektin alkuvaiheessa pitää perustaa

raakaversio arkkitehtuurista, jotta voidaan rakentaa järjestelmä, joka on helppo ymmärtää, toteuttaa ja kehittää (Kruchten, 2004).

Analysoinnin tarkoituksena on muuntaa vaatimukset esimerkiksi luokiksi ja alijärjestelmiksi käyttötapausten ohjaamana. *Suunnittelun* tarkoituksena sen sijaan on sopeuttaa analyysin tulokset ei-toiminnallisten vaatimusten, kuten kehitysympäristön, aiheuttamiin rajoitteisiin. Suunnittelu on siis analysoinnin tarkennus (Kruchten, 2004). Suunnittelun pitää määrittää järjestelmä vain niin tarkasti, että järjestelmä voidaan yksiselitteisesti toteuttaa. Tarkkuus kuitenkin vaihtelee projektikohtaisesti. Mitä tarkempi suunnitelma on, sen helpompaa järjestelmän toteuttaminen on ja toteutusvirheen riski pienempi. Kuvassa 24 kuvataan käytännön tärkeimmät roolit ja tuotokset.



Kuva 24: Analysointi- ja suunnittelukäytännön roolit ja tuotokset (Kruchten, 2004).

Ohjelmistoarkkitehti vastaa ohjelmistoarkkitehtuurin muodostamisesta. Tähän kuuluu luokkien ryhmittely paketeiksi ja alijärjestelmiksi sekä näiden välisten rajapintojen määrittely. Hänellä on siis keskeinen vastuu käytännön päätuotoksesta eli suunnittelumallista, jonka yleistyksenä voidaan tuottaa myös analyysimalli (Kruchten, 2004).

Suunnittelija määrittelee luokkien operaatiot, vastuut, attribuutit sekä suhteet muihin luokkiin ja päättää kuinka niitä pitää sopeuttaa toteutusympäristöön. Lisäksi suunnittelija voi olla vastuussa yhdestä tai useammasta suunnittelupaketista tai suunnittelualijärjestelmästä mukaanlukien luokat, joita paketit tai alijärjestelmät omistavat (Kruchten, 2004).

Analysointi ja suunnittelu voivat vaihtoehtoisesti sisältää muita rooleja. *Tietokan-*

tasuunnittelija toteuttaa järjestelmän vaatiman tietokannan, jos järjestelmä sellaista vaatii. *Arkkitehtuurin tarkastaja* ja *suunnittelun tarkastaja* (design reviewer) arvioivat työnkulun päätuotoksia eli *suunnittelumallia* ja *ohjelmistoarkkitehtuuri*-dokumenttia (Kruchten, 2004).

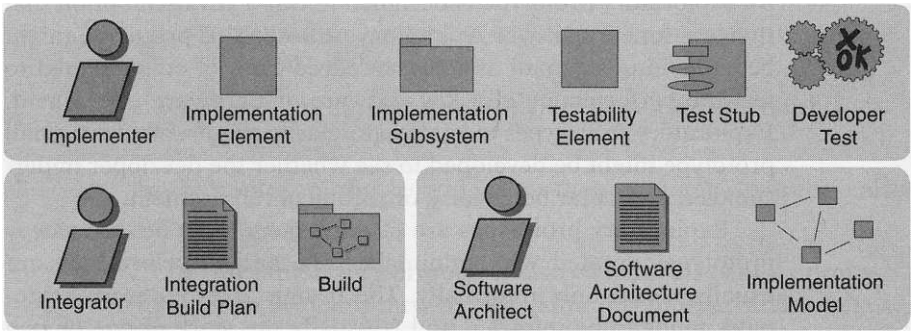
3.5.5 Toteutuskäytäntö

Toteutuskäytännön tarkoituksena on koodin organisointi kerroksittaisiksi toteutusalijärjestelmiksi, sekä luokkien ja olioiden toteuttaminen komponentteina, esimerkiksi lähdekooditiedostoina. Lisäksi tarkoituksena on toteutettujen komponenttien yksikkötestaus sekä kehittäjien työn tulosten integroiminen järjestelmään. Testauksen kattavuus toteutuskäytännössä rajoittuu yksittäisten komponenttien yksikkötestaukseen, sillä toteuttaja on vastuussa yksikkötestauksesta (Kruchten, 2004).

Toteutus RUP:issa sisältää kolme avainkäsitettä: *koosteet* (build), *integrointi* ja *prototyypit*. Kooste on toimiva versio järjestelmästä tai sen osasta, joka demonstroi osan lopullisen järjestelmän toiminnoista. Iteratiivisen kehityksen lomassa syntyy useita koosteita, joista jokainen tarjoaa aikaisia tarkastuspisteitä ja auttaa näin hahmottamaan ongelmia mahdollisimman aikaisin. Kukin kooste asetetaan kokoonpanonhallinnan alaisuuteen, jotta on mahdollista perua myöhemmät, epäonnistuneet muutokset. Koosteita otetaan usein ja säännöllisesti, esimerkiksi kerran päivässä (Kruchten, 2004).

RUP:in lähestymistavan mukaan uutta koodia integroidaan *lisäävästi* (incrementally). Tämä tarkoittaa, että koodia kirjoitetaan pienissä osissa, jotka lisätään toimivaan kokonaisuuteen yksi kerrallaan. Integrointi tapahtuu vähintään kerran jokaisessa iteraatiossa. Iteraatiosuunnitelma määrittää käyttötapaukset, joiden perusteella suunnitellaan luokat. Integrointistrategia keskittyy päättämään missä järjestyksessä luokat tulee toteuttaa ja yhdistää (Kruchten, 2004).

Prototyyppejä käytetään riskien vähentämiseen. Niitä voidaan käyttää avuksi esimerkiksi yritettäessä ymmärtää vaatimuksia tai vaikka kokeiltaessa käytettävyyttä. Prototyyppejä voi olla eri tyyppisiä riippuen käyttötarkoituksesta. Kruchten (2004) luettelee tyypeiksi *käyttäytymis-*, *rakenteelliset*, *tutkivat* ja *kehittyvät* prototyypit. Kuva 25 näyttää toteutuskäytännön pääroolit ja tuotokset. Pääroolit ovat *toteuttaja* ja *järjestelmäintegroija*.



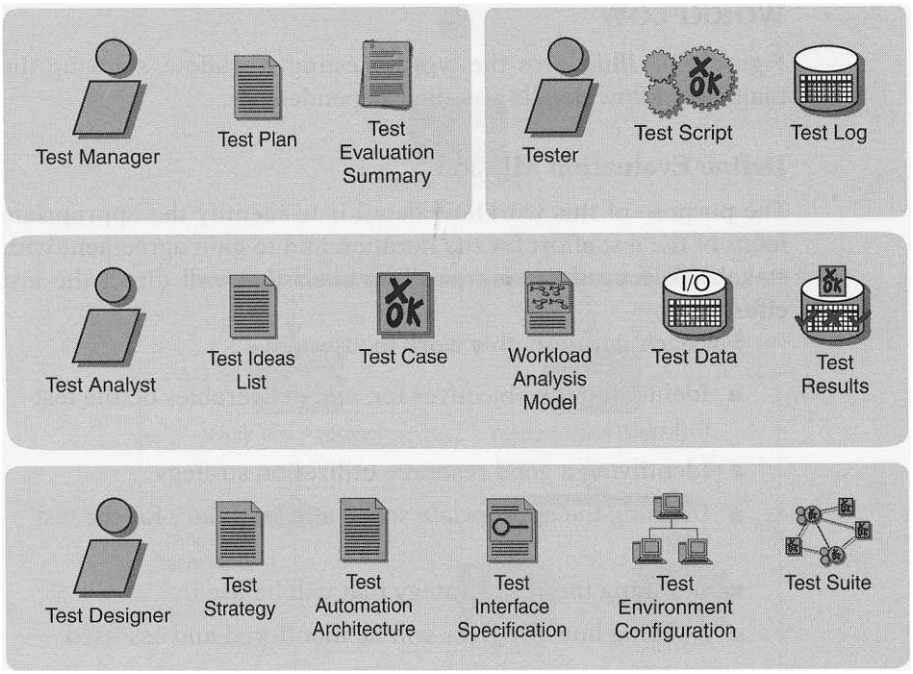
Kuva 25: Toteutuskäytännön roolit ja tuotokset (Kruchten, 2004).

Toteuttaja kehittää komponentit ja niihin liittyvät tuotokset sekä suorittaa yksikkötestauksen. Järjestelmäintegroija suorittaa koosteen rakentamisen. Lisäksi mukana on ohjelmistoarkkitehti, joka määrittää toteutusmallin rakenteen kerroksineen ja jakautuneena alijärjestelmiin. Lisäksi on rooli *koodin tarkastaja*, joka tarkastaa koodin laadukkuuden ja täsmäävyyden projektin standardin kanssa (Kruchten, 2004).

3.5.6 Testauskäytäntö

Testauskäytäntö toimii palveluna muille käytännöille (Kruchten, 2004), painottuen pääasiassa tuotteen laadun arvioimiseen. Testaus iteratiivisessa prosessissa ei ole yksittäinen toiminto eikä toisaalta projektin vaihe. Testausta tapahtuu koko projektin elinkaaren ajan, joten RUP:issa testaus tarjoaa palautetta välittömästi, kun jotakin on mennyt pieleen, jolloin puutteiden korjaamiseen on enemmän aikaa ja resursseja. Kuva 26 esittelee testauskäytännön tärkeimmät roolit ja tuotokset.

Testauspäälliköllä on kokonaisvastuu testauksen onnistumisesta. Rooliin kuuluu laadun ja testauksen valvonta, resurssien suunnittelu ja hallinta ja testausta estävien ongelmien ratkaisu. *Testausanalysoija* on vastuussa vaadittujen testien tunnistamisesta ja määrittelemisestä, yksityiskohtaisesta testauksen edistymisen ja tulosten seurannasta kussakin syklissä ja kokonaisvaltaisen laadun arvioinnista testauksen osalta. *Testaussuunnittelija* määrittelee testauksen lähestymistavan ja on vastuussa sen onnistuneesta toteuttamisesta. Rooliin kuuluu testien toteuttamiseen sopivien tekniikoiden, työvälineiden ja ohjeiden valitseminen. *Testaajan* vastuulla on järjestelmätestien suorittaminen. Tähän sisältyy testauksen järjestäminen ja suoritus, testauksen suorittamisen arviointi, virheistä toipuminen, testaustulosten arvioiminen sekä muutospyyntöjen hallinta (Kruchten, 2004).



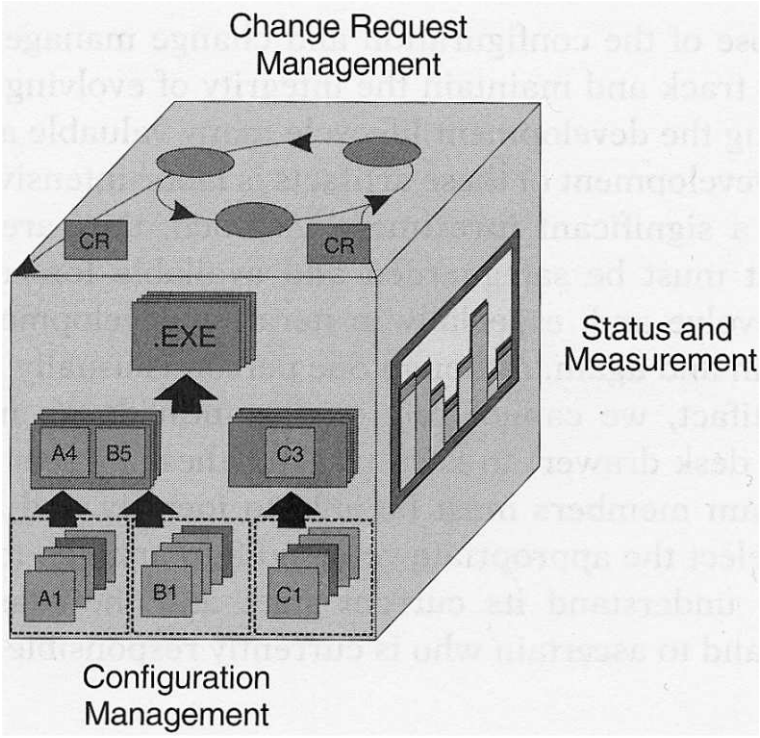
Kuva 26: Testauskäytännön roolit ja tuotokset (Kruchten, 2004) .

3.5.7 Kokoonpanon- ja muutostenhallinnan käytäntö

Käytännön tarkoituksena on seurata ja ylläpitää kehittyvän, kaikki tuotokset sisältävän projektivarannon eheyttä (Kruchten, 2004). Tiimin jäsenten täytyy pystyä tunnistamaan ja löytämään tuotoksia, valitsemaan sopiva versio tuotoksista, nähdä niiden historia ja ymmärtää niiden nykytila sekä se, miksi ne on saatettu nykyiseen tilaansa sekä kuka niistä on nykyisin vastuussa. Samaan aikaan tiimin pitää pystyä seuraamaan tuotteen kehitystä, vastaanottaa ja hallita muutospyyntöjä ja toteuttaa muutoksia useisiin tuotoksiin. Lisäksi käytäntö tarjoaa projektinhallintakäytännön tueksi tietoa joidenkin avaintuotosten tilasta, sekä kokoaa mittauksia liittyen niiden muutoksiin (Kruchten, 2004). Kokoonpanon- ja muutostenhallinnan käytännöllä on siis kolme itsenäistä toimintoa, jotka esitellään kuvassa 27.

Kokoonpanonhallinta käsittää tuotosten tunnistamisen, versiot ja riippuvuudet tuotosten välillä sekä erilaisten kokoonpanojen tunnistamisen.

Muutospyyntöjenhallinta liittyy prosessin rakenteeseen. Siihen sisältyy muutospyyntöjen vastaanottaminen ja hallinnointi. Se käsittelee myös muutosten potentiaalisen vaikutuksen sekä seuraa muutoksen tilaa, kunnes se on toteutettu.



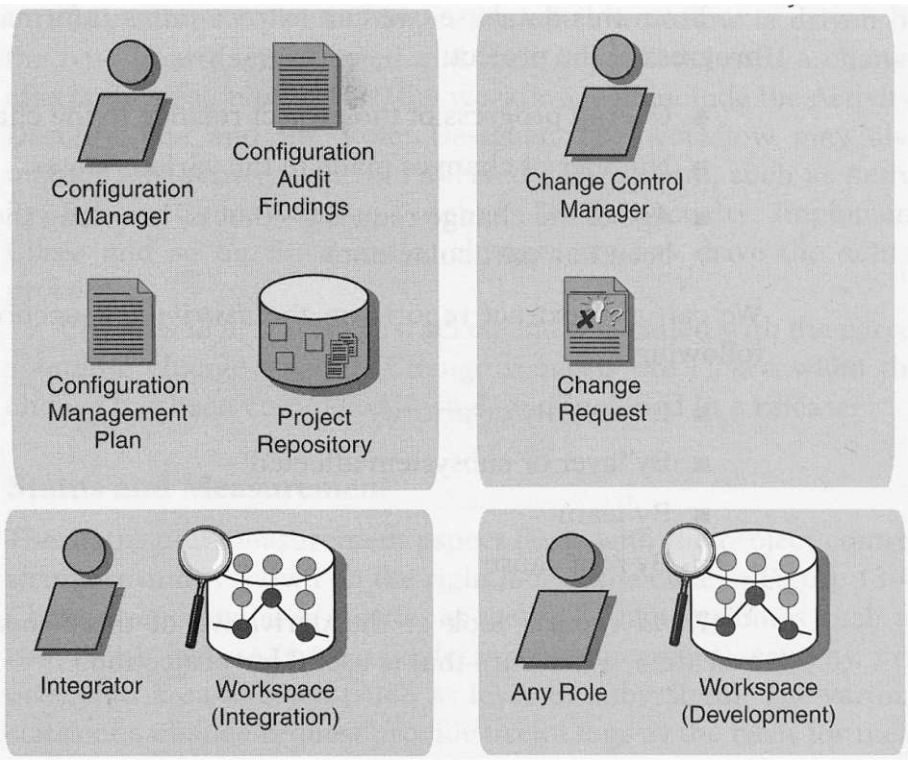
Kuva 27: CCM - kuutio (Kruchten, 2004).

Tila ja mittaus käsittää tiedon tuottamisen projektin johdolle työvälineillä, joilla kokoonpanon- ja muutostenhallintaa harjoitetaan. Esimerkiksi tuotteen tila, edistyminen, suunta, laatu, ja kulut sekä ongelma-alueet, jotka kaipaavat huomiota ovat hyödyllistä tietoa arvioimista varten (Kruchten, 2004). Kuva 28 näyttää käytännön roolit ja tuotokset.

Kokoonpanopäällikkö vastaa tuotteen rakenteen asetuksista kokoonpanonhallintajärjestelmässä ja kokoaa sopivat tila- ja mittausraportit projektipäällikölle.

Muutostenhallintapäällikkö johtaa muutostenhallintaprosessia. Tässä roolissa on yleensä muutostenhallinnan johtotiimi, jonka pitäisi koostua kaikkien sidosryhmien edustajista. Pienessä projektissa yksikin henkilö voi hoitaa roolia.

Myös ohjelmistoarkkitehti, toteuttajat ja integroijat osallistuvat tähän käytäntöön omissa rooleissaan. Lisäksi missä tahansa roolissa toimija voi toimittaa muutospyynnön ja käyttää työtiloja eli tarvittavia kokoonpanonhallinnan sisältämiä tuotosjoukkoja.



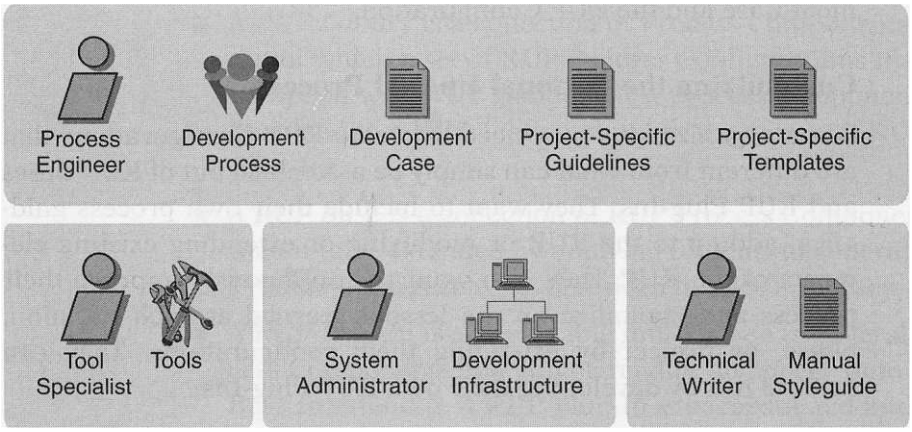
Kuva 28: Kokoonpanon- ja muutostenhallinta-käytännön roolit ja tuotokset (Kruchten, 2004).

3.5.8 Ympäristökäytäntö

Käytännön tarkoituksena on tukea kehitysorganisaatiota prosessien ja työvälineiden hallinnassa. Tähän lukeutuu työvälineiden valinta ja hankkiminen, työvälineiden kokoonpanon ja asetusten saattaminen organisaatiolle sopiviksi, prosessin konfigurointi, prosessin parantaminen sekä tekniset palvelut prosessin tukemiseksi (Kruchten, 2004).

Kaikkia RUP:in prosessikehyksen rooleja, tuotoksia ja ohjeita ei voida käyttää jokaisessa projektissa (Kruchten, 2004). Näin ollen kutakin projektia varten pitää valita sopiva RUP:in alijoukko, prosessikonfiguraatio, joka on sopiva tietylle projektille. Projektiryhmän pitää päättää käytettävä prosessikonfiguraatio, eli esimerkiksi roolitus ja käytettävät työkalut. Kuvassa 29 on kuvattu RUP:in ympäristökäytännön roolit ja tuotokset.

Käytännön päärooli on *prosessisuunnittelija*, joka on vastuussa ohjelmistotuotantoprosessista. Tähän lukeutuu prosessin konfigurointi ennen projektin alkua sekä prosessin



Kuva 29: Ympäristö-käytännön roolit ja tuotokset (Kruchten, 2004).

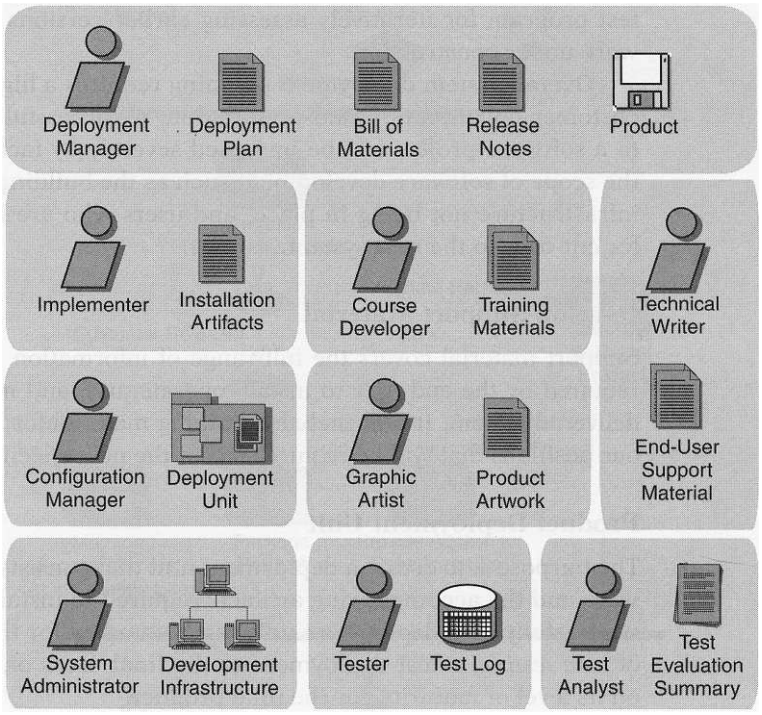
jatkuva parantaminen kehitystyön aikana (Kruchten, 2004).

3.5.9 Käyttöönottokäytäntö

Käyttöönottokäytännön tarkoituksena on saattaa valmis järjestelmä asiakkaiden käyttöön. Käytäntöön lukeutuu järjestelmän beta-testaus lopullisessa toimintaympäristössä, järjestelmän pakkaaminen kuljetusta varten, järjestelmän jakelu, asennus, järjestelmän myyjien ja loppukäyttäjien koulutus sekä tietokantojen konvertointi (Kruchten, 2004).

Käyttöönotossa voi olla variaatioita riippuen siitä, minkälainen järjestelmä on kyseessä. Käyttöönoton toiminnot ajoittuvat aina kuitenkin siirtymisvaiheeseen. Projektin onnistuminen näkyy tässä vaiheessa asiakkaan halukkuutena käyttää uutta järjestelmäänsä. Siirtymisvaiheen ja käyttöönottokäytännön päätavoite on, että asiakkaan siirtyminen tuotteen käyttäjäksi tapahtuu sulavasti. Tämän vuoksi asiakas on osallisena jo aiempien julkaisujen beta-testauksessa sekä muuten osallisena kehitysprosessissa (Kruchten, 2004). Käytännön roolit ja tuotokset näkyvät kuvassa 30.

Käyttöönottopäällikkö suunnittelee ja organisoi käyttöönoton. *Projektipäällikkö* vastaa pääasiallisesti kommunikoinnista asiakkaiden kanssa. Niinpä hän on vastuussa käyttöönoton hyväksymisestä. *Tekninen kirjoittaja* suunnittelee ja tuottaa loppukäyttäjille suunnatun dokumentaation ja *kurssien tuottaja* suunnittelee ja tuottaa koulutusmateriaalin. *Graafinen suunnittelija* (Graphic artist) tuottaa tuottteeseen liittyvän taiteellisen puolen. *Testaaja* ajaa hyväksymistestit ja on vastuussa tuotteen riittävästä testaaamisesta. *Toteuttaja* luo asennusskriptit ja niihin liittyvät tuotokset, jotka auttavat loppukäyt-



Kuva 30: Käyttöönottokäytännön roolit ja tuotokset (Kruchten, 2004).

täjää tuotteen asennuksessa (Kruchten, 2004).

4 XP-ohjelmoinnin ja RUP-prosessin vertailu

XP-ohjelmoinnin ja RUP-prosessin periaatteissa on paljon samanlaista. Molemmat prosessit perustuvat iteratiiviseen, lisäävään lähestymistapaan, jonka seurauksena on kasvava järjestelmä. Tämän lähestymistavan tarkoituksena on mukautuminen muuttuviin vaatimuksiin. Suurimmat erot prosessien välillä ovat itse prosessin määrittelyssä. RUP on formaalisti roolien, tuotosten, toimintojen, käytäntöjen ja työnkulkujen avulla määritelty prosessi, mutta XP-ohjelmointi on pikemminkin hajanainen kokoelma käytäntöjä ja periaatteita, joita soveltamalla ohjelmistoja kehitetään. English (2002) kuvailee XP-ohjelmoinnin olevan pikemminkin prosessin antiteesi, tapa tehdä prosessi tarpeettomaksi, kuin varsinainen ohjelmistotuotantoprosessi. RUP on paitsi prosessi, myös prosessikehys, mikä tarkoittaa, että RUP on elementtejä mukaanottamalla tai poistamalla modifioitavissa erilaisten projektien tarpeisiin. Näistä eroista huolimatta vertaaminen on lukuisista samankaltaisista periaatteista ja niissä esiintyvistä eroista johtuen mielekästä. Vertailu perustuu kvalitatiivisiin ominaisuuksiin. Taulukossa 6 on esitetty taulukkomuodossa ominaisuuksia, joihin tämän luvun alakohdissa esitettävät vertailut perustuvat.

4.1 Prosessin määrittely

XP-prosessi määritellään perusarvojensa ja käytäntöjensä avulla. Tietyllä tavalla arvot ja käytännöt tarjoavat prosessikehyksen, joskin väljän. Arvot ja käytännöt ovat melko pitkälle kuvailevia. Toki ne ovat myös sitovia siinä mielessä, että ilman niiden noudattamista ei oikeastaan voida puhua XP-ohjelmoinnista.

RUP määritellään formaalisti roolien, toimintojen, tuotosten, työnkulkujen ja käytäntöjen avulla. RUP:in avulla on mahdollista määrittää tarkasti projektiin osallistuvat roolit ja niiden vastuut, kunkin roolin osallisuus kuhunkin työnkulkuun, henkilön toimiminen tietyssä roolissa, tuotokset joita tuotetaan ja niin edelleen. Prosessikehyksen ansiosta prosessi voidaan määritellä sopivaksi projektikohtaisesti.

Taulukko 6: XP-ohjelmointi vs. RUP-prosessi.

Vertailukohde	*XP-ohjelmointi*	*RUP-prosessi*
Prosessin määrittely		
Arvot ja käytännöt	Kuvailevia, epäformaaleja	Muodollisia, sitovia
Tiimit	Kokonainen tiimi	Prosessikehyksen määrittämät, perinteisemmät tiimit
Roolit	Epämuodollisia, toimiminen monissa rooleissa, eivät henkilöön sidottuja	Muodolliset vastuut, toimiminen monissa rooleissa, eivät henkilöön sidottuja
Toiminnot	Toiminnon ja tuotoksen suhde ei määritelty	Tietty toiminto tuottaa tietyn tuotoksen
Tuotokset	Koodi tärkein, muuten ei ohjeistettu	Roolien, toimintojen ja tuotosten suhde formaalisti määritetty
Iteratiivisuus	Iteratiivinen, vähäinen etukäteissuunnittelu	Iteratiivinen, arkkitehtuurikeskeinen, komponenttipohjainen
Suunnittelu		
Vaatimukset	Käyttäjätarinat	Käyttötapaukset
Arkkitehtuuri	Kasvava, ei etukäteen suunniteltu, valmis kun järjestelmä on valmis	Arkkitehtuurikeskeinen, voi olla kehittyvä
Mallinnus	Käytetään tarvittaessa	Vahva sidos UML-kieleen, mallinnuskeskeinen
Toteutus	Pariohjelmointi, refaktorointi, koodin yhteisomistus	Toteutuskäytäntö (koodin organisointi, komponenttien toteuttaminen, yksikkötestaus, integrointi)
Testaus	Testauslähtöinen, jatkuvaa	Perinteisempää, jatkuvaa
Skaalautuvuus	Ei suoraan määritelty	Prosessikehys

4.1.1 Arvot ja käytännöt

Jokaisen XP-projektin taustalla ovat ohjenuorina kommunikaatio, yksinkertaisuus, palaute ja rohkeus. Kommunikaatiota arvostetaan arvoista eniten. Se korvaa suurelta osin kaikenlaisen perinteisessä ohjelmistotuotannossa dokumentointiin käytettävän ajan. Kommunikointi on läsnä kaikessa XP-projektissa tapahtuvassa toiminnassa sekä ryhmän sisällä, että ryhmän ja asiakkaan välillä. Yksinkertaisuuden arvon mukaisesti voidaan suunnittelu ja koodi tuottaa vastaamaan juuri nykyhetken vaatimuksiin ennakoimatta tulevia vaatimuksia. Koska testausta, koodausta ja integrointia tehdään jatkuvasti rinnakkain ja asiakas on mukana kaikessa toiminnassa, tulee järjestelmän kehittämisen sivutuotteena jatkuvasti paljon palautetta. Rohkeuden perusarvo sanoo, että suunnittelu on uskallettava lopettaa ja toteuttaminen aloittaa.

Melko suuressa määrin XP-ohjelmoinnissa luotetaan, että esimerkiksi kokonainen tiimi, kommunikaatio, koodin yhteisomistus, koodausstandardit, pienet tuoteversiot ja yksinkertainen suunnittelu ohjaavat projektia ilman, että tarvitaan muodollisia roolien ja vastuiden määritelmiä. Käytäntöjen määrittely XP-ohjelmoinnissa eroaa RUP:in vastaavasta siinä, että ne toimivat pikemmin kuvailevina ohjenuorina, kuin tiukkoina määritelminä siitä, kuka tekee mitä ja miten. Ne ovat abstraktimpia, koko ajan läsnä olevia tapoja suorittaa kehitystyötä.

RUP on lähtökohtaisesti arvomaailmaltaan hieman erilainen. Kommunikaatio on tietysti myös RUP:issa läsnä, mutta sitä ei erikseen koroteta itseisarvon asemaan. Sen sijaan RUP:in vahva sidos UML:ään tekee siitä pikemminkin dokumentointi- ja mallinnusorientoituneen kuin kommunikaatio-orientoituneen lähestymistavan. Etukäteissuunnittelusta, arkkitehtuurikeskeisyydestä ja komponenttipohjaisuudesta johtuen yksinkertaisuus suunnitelmissa ja koodissa ei sinänsä ole itseisarvo. Esimerkiksi yksittäisen komponentin osalta uudelleenkäytön mahdollisuus on tärkeämpi, kuin absoluuttinen yksinkertaisuus. XP-ohjelmoinnissa toteutetaan iteraatiossa yksinkertaisin mahdollinen tämänhetkiset vaatimukset toteuttava järjestelmä, kun taas RUP:issa pyritään ennakoimaan ja ottamaan huomioon tulevaisuuden vaatimuksia. Iteratiivisen prosessin ja lisäyksinä toimitettavan järjestelmän sivutuotteena niin RUP:issa, kuin myös XP-ohjelmoinnissa järjestelmän kehittäminen tuottaa paljon palautetta. Molemmissa prosesseissa palautetta hyödynnetään mittaamiseen ja seurannan suorittamiseen. Rohkeuden vastapainoksi RUP on kenties hieman defensiivisempi johtuen juuri etukäteisuunnittelusta.

Käytännöt RUP:issa ovat muodollisia roolien, toimintojen ja tuotosten pakkauksia, joita käytetään projektin toimintojen organisoimiseksi ja hallinnoimiseksi. Esimerkiksi liiketoiminnan mallintamiselle ja projektinhallinnalle on omat käytäntönsä. XP-ohjelmoinnissa käytäntöjä ei varsinaisesti jaotella näin.

4.1.2 Tiimit

XP-ohjelmoinnissa pyritään kokonaisen tiimin käsitteellä, jota esimerkiksi English (2002) kuvaa myös termillä läsnäoleva asiakas, häivyttämään kehitystiimin ja asiakkaan välistä rajaa käsittelemällä asiakasta kehitystiimin vastuullisena ja osallistuvana jäsenenä. Vaikkakin asiakas- ja kehitystiimejä myös XP-ohjelmoinnissa käsitellään kahtena eri kokonaisuutena, on asiakkaalla siitä huolimatta suuri rooli myös kehitystiimiin kuuluvana jäsenenä. Kommunikaation perusarvo toteutuu tässäkin käytännössä. Asiakas on koko kehityssyklin ajan läsnä kehitystyössä suunnitellen hyväksymistestien sisältöä, suorittaen niitä sekä ohjaamassa kehitystyötä. Tiimien sisäiset roolit on määritelty mutta ne toimivat pikemminkin ohjenuorina ja kuvailevina kertomuksina, kuin sitovina vastuiden ja toimintojen joukkoina.

RUP:issa asiakkaan osallistumista tarvitaan tietysti myös, mutta ei samassa mittakaavassa kuin XP-projektissa. Asiakkaan osallistuminen rajoittuu lähinnä alkuvaiheeseen, jossa tunnistetaan käyttötapaukset sekä iteraatioiden loppukohtiin, joissa kehitystiimi toimittaa julkistuksen. Kehitystiimin ja asiakastiimin välinen raja sekä tiimien vastuut ovat selkeämmät. Kehitystiimi itse on suuremmassa vastuussa kehitystyön suunnasta, kuin XP-ohjelmoinnissa. Roolit, niiden vastuut sekä niissä toimijat on tarkasti määritelty ja näin tiedetään aina selvästi kenen vastuulla mikäkin on.

4.1.3 Roolit

XP-ohjelmoinnissa roolit ovat väljästi määriteltyjä. Roolit eivät ole statusymboleita, vaan pikemminkin toimivat kuvailevana ohjeistuksena siitä mitä kenenkin pitäisi tehdä. Toki tietynlaiset tehtävät on selkeästi asetettu tiettyjen roolien hoidettavaksi, kuten esimerkiksi mittaajan tehtävät. Roolien osalta on kuvattu myös, minkälaisia taitoja tai ominaisuuksia tietyssä roolissa toimiva tarvitsee. Roolit eivät sulje toisiaan pois, vaan henkilö voi toimia useissa rooleissa samanaikaisesti tai vaihdella roolien välillä. Tietyt asiat, kuten koodin tuottaminen ja muu päiväkohtainen toiminta, tapahtuvat koko

tiimin toimesta eivätkä roolisidonnaisesti. Sen sijaan esimerkiksi juuri mittaukset ja asiakaskontaktien hoito on asetettu tiettyjen roolien vastuulle.

RUP:issa myös roolit ovat tarkemmin määritellyt ja muodollisemmat kuin XP-ohjelmoinnissa. Samoin roolin käsite on RUP:issa paljon suuremmassa osassa. Tietyt roolit ovat yksiselitteisesti vastuussa tietyistä toiminnoista joiden seurauksena syntyy tiettyjä tuotoksia. Myöskään RUP:issa tosin roolit eivät ole sidottuja yhteen henkilöön, eivätkä ne ole työnimikkeitä. Samoin kuin XP-ohjelmoinnissa, RUP:issa roolit tulisi jakaa henkilöille heidän reaalisten ominaisuuksiensa mukaan. Edelleen samoin kuin XP-ohjelmoinnissa, henkilö ei ole sidoksissa rooliin koko projektin ajan, vaan roolit muuttuvat projektin edetessä ja tehtäväkohtaisesti. Työnkulkuja käytetään kuvailemaan roolien vuorovaikutusta prosessin eri käytäntöjen, kuten vaatimuskäytännön, sisällä.

4.1.4 Toiminnot

XP-ohjelmoinnissa käytäntöjen toteuttamisen kautta ohjelmoijapari tuottaa suunnitelmia, koodia, dokumentteja tai muita tuotoksia. Tarkoin ei kuitenkaan ole määritelty tietyn roolin tietyn toiminnon seurauksena tuottamaa tuotosta. Tämä on loogista, sillä XP-ohjelmoinnissa ei myöskään määritellä mitä tuotoksia tuotetaan, vaan tuotoksia, kuten malleja tuotetaan tarvittaessa. Tästä huolimatta esimerkiksi Paulk (2001) pitää XP-ohjelmointia kurinalaisena prosessina ja jopa CMM-mallia (Capability Maturity Model) täydentävänä.

RUP:issa tietyn toiminnon suorittaa rooli ja sen seurauksena syntyy tuotos. Toiminnon suorittaja ja tulos ovat eksplisiittisesti määriteltyjä. Siinä missä XP-ohjelmoinnissa tulokset syntyvät, kun tiimi toteuttaa XP-prosessin määritteleviä käytäntöjä, RUP:issa tuotokset syntyvät roolin suorittaessa toimintoja.

4.1.5 Tuotokset

XP-ohjelmoinnin tapauksessa itse koodi on tuotoksista kaikista tärkein. Muita tuotoksia ei itsessään määritellä. Esimerkiksi erilaisten mallien käyttöä tai dokumentointia ei XP-ohjelmoinnissa ole ohjeistettu. Ohjenuorana toimii vain yksinkertaisuus. Jos jotakin mallia tarvitaan, sitä voidaan käyttää tai jos dokumentin kirjoittaminen koetaan tarpeelliseksi, sellainen voidaan kirjoittaa. Kaikista suurin painoarvo on kuitenkin itse

koodilla, jonka voi katsoa olevan ainut XP-projektin merkityksellinen tuotos.

RUP:issa puolestaan on tuotoksilla suuri merkitys. Se, mikä rooli laatii minkäkin dokumentin, luokan, metodin tai vastaavan minkäkin toiminnon seurauksena, on tarkasti määritelty. Selkeä hyvä puoli verrattuna XP-ohjelmointiin on se, että roolien, toimintojen ja tuotosten suhde on yksikäsitteisesti määritetty, vaikkakin se XP-ohjelmoinnin näkökulmasta saattaa tuntua jäykältä. RUP sisältää myös valmiita kaavaimia erilaisille tuotoksille, joten projektista toiseen voidaan käyttää samoille pohjille laadittuja tuotoksia.

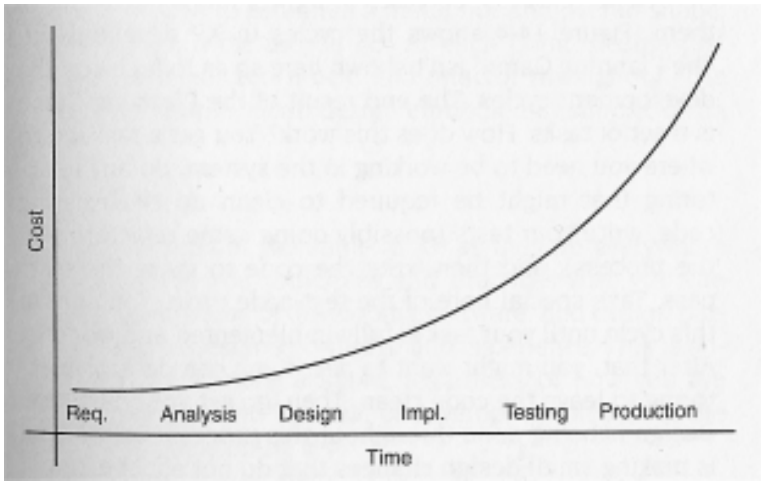
4.1.6 Iteratiivisuus

Molemmat prosessit, XP-ohjelmointi ja RUP-prosessi, perustuvat iteratiivisuuteen. Toisiinsa verrattuna ne eivät tässä suhteessa juurikaan eroa toisistaan. Verrattuna perinteiseen vesiputousmallin mukaiseen ohjelmistoprosessiin eroa on lähinnä siinä, miten suhtaudutaan etukäteissuunnitteluun, suunnitelmien lyömiseen lukkoon ja miten prosessi mukautuu muutoksiin. Iteratiivisen kehitysprosessin idea pohjautuu suuren kokonaisuuden paloitteluksi pienemmiksi osaprojekteiksi, jolloin kehitystyön mahdolliset suunnanmuutokset ovat joustavampia, edullisempia ja nopeampia toteuttaa. Molemmat prosessit ovat perinteiseen vesiputousmalliin verrattuna joustavampia, toisiinsa verrattuna suurta eroa ei ole. Iteraation sisällä tapahtuu samat asiat kuin vesiputousmallissa. Vaatimuksia määritellään, järjestelmää toteutetaan, testataan ja koodia integroidaan. Periaatteessa iteraation sisällä tapahtuu vesiputousmallin mukainen prosessi projektin osalle, kuitenkin sillä erotuksella, etteivät asiat tapahdu tiukasti lineaarisessa järjestyksessä, vaan pikemminkin rinnakkain. Vaatimuksia määritellään ja järjestelmää toteutetaan samanaikaisesti.

Pieni painotuksellinen ero prosessien välillä kuitenkin löytyy. Lukuunottamatta aivan projektin alkuvaihetta, jossa kerätään ensimmäiset käyttäjätarinat sekä luodaan vertauskuva järjestelmästä, sekä projektin loppuvaihetta, jossa testaamisella ja muilla käyttöönottoon liittyvillä toiminnoilla on suurempi painoarvo, kautta koko XP-ohjelmoinnin kehityssyklin tapahtuu samoja toimintoja suunnilleen samassa suhteessa. RUP määrittää neljä eri vaihetta iteratiivisen prosessin etenemiselle. Toki myös RUP:issa toiminnot tapahtuvat rinnakkain mutta silti prosessi etenee hieman enemmän perinteisen vesiputousmallin mukaisesti, siten että alkuvaiheessa painoarvo on selvästi vaatimusten määrittelemisessä ja kokonaisarkkitehtuurin luomisessa, kun taas loppu-

vaiheessa painoarvo on käyttöönottoon liittyvissä seikoissa.

Kuvassa 31 havainnollistetaan, miten muutoksen aiheuttamat kustannukset muuttuvat perinteisen vesiputousmallin mukaisen ohjelmistotuotantoprosesin eri vaiheissa. Kuvassa 32 puolestaan kuvataan kustannusten muutosta XP-ohjelmoinnissa, mutta kustannuksen muutosta voidaan soveltaa myös RUP:iin, joka myös on iteratiivinen prosessi.

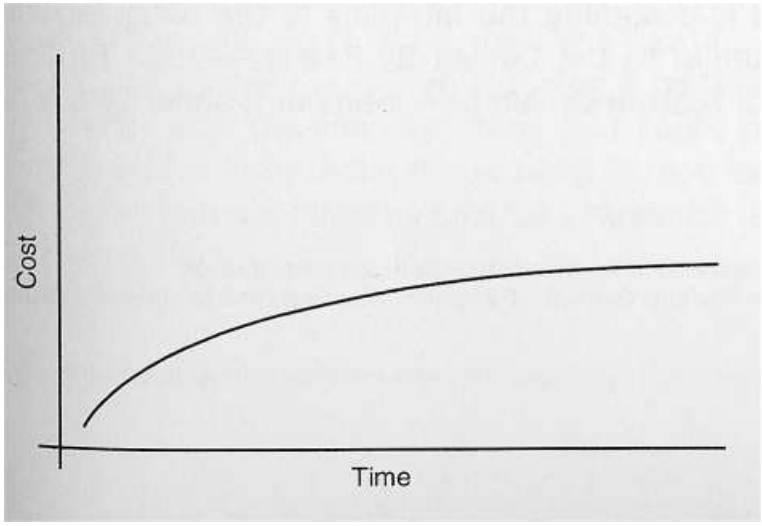


Kuva 31: Kustannusten muutokset vesiputousmallissa (Astels & al., 2002).

Kuvasta 31 huomaa, kuinka vaatimusten muuttumisen kustannukset kasvavat vesiputousmallissa sitä suuremmiksi, mitä myöhemmässä vaiheessa vaatimukset muuttuvat. Iteratiivinen prosessi auttaa kuvan 32 mukaisesti pitämään kustannusten kasvun pienenä. Tämä tarkoittaa, että vaatimusten muuttuminen missä tahansa projektin vaiheessa maksaa suunnilleen saman verran.

4.2 Suunnittelu

Suunnittelun osalta on XP-ohjelmoinnissa ja RUP:issa jonkin verran eroja. XP-ohjelmoinnissa suunnittelu on reaaliaikaista toimintaa, RUP:issa keskitytään enemmän formaalimpaan etukäteissuunnitteluun ja arkkitehtuurikeskeisyyteen. Vaatimusten keräämisessä käyttäjätarinoin XP:ssa ja käyttötapauksin RUP:issa on pieniä eroja. Myös suhtautumisessa mallintamiseen on vivahde-eroja.



Kuva 32: Kustannusten muutokset iteratiivisessa prosessissa (Astels & al., 2002).

4.2.1 Vaatimukset

XP-ohjelmoinnissa vaatimukset selvitetään joka iteraation alussa tapahtuvassa suunnittelupelissä, jossa asiakkaat kirjoittavat käyttäjätarinoita ja kehittäjät arvioivat ne. RUP:issa vaatimukset pyritään selvittämään projektin aloitusvaiheessa määrittämällä asiakkaan kanssa käyttötapaukset. Astelsin & al. (2002) mukaan asiakkaan mukanaolo ei ole yhtä intensiivistä käyttötapausten kohdalla, kuin se on käyttäjätarinoiden kanssa.

Astels & al. (2002) esittävät Jeffriesin (2001) perusteella, että myös XP-ohjelmointiin voidaan integroida käyttötapausten käyttö käyttäjätarinoiden tilalle, jos siihen on tarvetta. Sitä ei tosin suoranaisesti suositella. Kahta aluetta käyttötapauksista pitää Astelsin & al. (2002) mukaan parantaa, jos käyttötapauksia tahdotaan käyttää tehokkaasti XP-ohjelmoinnin kanssa. Ensimmäinen on käyttötapausten aikataulutus. Ne pitää voida aikatauluttaa askelten, esi- ja jälkiehtojen tasolla, jotta niitä voitaisiin käyttää XP-projektissa. Toinen alue on asiakkaiden tuki. Perinteisesti kehittäjät kirjoittavat käyttötapaukset. Jos kuitenkin tietämys kohdealueesta on vajavaista, on tärkeää, että asiakkaat ovat mukana käyttötapausten laatimisessa. Se käytetäänkö XP-projektissa käyttötapauksia vai käyttäjätarinoita, on projektikohtaisesti päätettävissä.

RUP:issa vaatimusten keruu ja määrittely hoidetaan käyttötapausten avulla. Käyttötapausten hyvä puoli verrattuna käyttäjätarinoihin on se, ettei käyttötapauksen kuvaamisen jälkeen enää tarvita asiakasosallistumista. Hyvänä puolena voidaan pitää myös sitä, että käyttötapaukset ovat selkeämmin rajattuja ja määrämuotoisempia kuin

käyttäjätarinat. Käyttötapauskaaviota ja käyttötapauksen kuvausta tarkastelemalla ohjelmistokehittäjän on helppo nähdä tietyn käyttötapauksen toimijat ja tapahtumat. UML-kielen mukainen käyttötapauskaavio on varsin universaalisti ymmärretty käsite ohjelmistotuottajien parissa. Varsinkaan huonosti kirjoitetun käyttäjätarinan osalta ei välttämättä ole näin, jolloin käyttäjätarinan tarkoitus voi jäädä joiltakin osin epäselväksi.

4.2.2 Arkkitehtuuri

Vertauskuvan voidaan katsoa olevan korkean tason kuvaus XP-järjestelmän arkkitehtuurista. Lähtökohtaisesti kuitenkin XP-ohjelmoinnissa suhtaudutaan nuivasti etukäteen suunniteltua arkkitehtuuria kohtaan, johtuen esimerkiksi kohdassa 2.2.4 kuvatusta yksinkertaisen suunnitelman käytännöstä. Järjestelmän kokonaisarkkitehtuurin ei voida katsoa olevan valmis ennen kuin itse järjestelmä on valmis. Iteratiivisen prosessin myötä kaikki suunnitelmat, koodi ja arkkitehtuuri tarkentuvat asteittain. Oikeaaikaisen suunnittelun tapa, jossa pyritään lykkäämään kaiken suunnitteleminen siihen hetkeen, kun se on välttämätöntä, hylkii etukäteen suunniteltua arkkitehtuuria. Myös arkkitehdin rooli XP-ohjelmoinnissa eroaa perinteisen ohjelmistotuotannon arkkitehdin roolista. Arkkitehti tuottaa järjestelmän arkkitehtuurin tarvittaessa, silloinkin pitäen huolta siitä, että arkkitehtuuri on mahdollisimman yksinkertainen. Samoin testitapausten joukon, joka kattaa järjestelmän arkkitehtuurin tietyn iteraation osalta, on oltava pienin mahdollinen.

RUP on arkkitehtuurikeskeinen prosessi ja iso osa RUP:ista keskittyy mallintamiseen. Perusteluna tälle on, että mallien avulla voidaan paremmin ymmärtää sekä ongelmaa että ratkaisua (Kruchten, 2004). Kun kaikki tärkeimmät järjestelmää eri näkökulmista kuvaavat mallit kootaan yhteen, on koossa jäjestelmän kokonaisarkkitehtuuri. Kokonaisarkkitehtuurin hyvä määrittely auttaa projektin hallinnassa, mikä on totta etenkin suurien projektien kohdalla. Kuten XP-ohjelmoinnissa, myös RUP:issa arkkitehtuuri kehittyy iteraatioiden myötä. Kuitenkin eroa on siinä, että RUP:issa pyritään jo projektin alussa luomaan hyvin suunniteltu pohja arkkitehtuurille. RUP tukee myös komponenttipohjaista kehittämistä, mikä on luonteva pohja arkkitehtuurikeskeiselle lähestymistavalle. Uudelleenkäytettäviä komponentteja käyttämällä on helppoa rakentaa kokonaiskuva järjestelmän arkkitehtuurista. Komponenttipohjaisen kehityksen tukeminen on RUP:issa selvästi etu verrattuna XP-ohjelmointiin.

Krollin & Kruchtenin (2003) mielestä XP-ohjelmoinnissa suosittu tapa siirtyä toteutukseen, vaikkei kokonaiskuva olisi täysin selvillä, on suurin yksittäinen ero XP-ohjelmoinnin ja RUP:in välillä. Heidän mukaansa juuri tämä seikka aiheuttaa sen, että XP-ohjelmoinnin tyypillisesti katsotaan sopivan vain pienille tiimeille, pieniin projekteihin ja tilanteisiin, joissa arkkitehtuurin tarve on joko vähäinen tai kohdealue on tuttu.

4.2.3 Mallinnus

Amblerin (2002) mukaan XP-ohjelmoinnissa voidaan hyödyntää UML-kieltä mallinnuksessa. Toiminto-, luokka-, yhteistyö- ja muut kaaviot toimivat hyvin myös XP-ohjelmoinnissa silloin, kun mallinnusta tarvitaan. XP-ohjelmoinnissa painoarvo on kuitenkin koodin tuottamisessa eikä mallintamisessa.

Arlow & Neustadt (2002) kuvailevat UML:n olevan oliosuuntautuneeseen mallinnukseen keskittynyt visuaalinen mallinnuskieli, mikä on Rumbaughin & al. (1999) mukaan UML:n alkuperäinen idea. RUP puolestaan on prosessikehys, joka määrittelee, kuinka oliosuuntautunutta analyysia ja suunnittelua toteutetaan. RUP:in arkkitehtuuri- ja mallinnuskeskeinen sekä käyttötapauspohjainen lähestymistapa ovat riittävä syy UML:n käyttämiseksi. Monet UML:n kaaviot, kuten käyttötapaus- ja luokkakaavio voivat suoraan toimia RUP:in tuotoksina, niistä kootaan järjestelmän arkkitehtuuri ja ne ovat universaalisti ymmärretty standardi (Kruchten, 2004). Jacobsonin & al. (1999) mukaan UML on olennainen osa Unified-prosessia, johon RUP pohjautuu.

4.3 Toteutus

Kaikkein merkittävin ero kahden prosessin välillä on XP-ohjelmoinnissa toteutettavat pariohjelmoinnin, refaktoroinnin ja koodin yhteisomistuksen käytännöt. Näistä varsinkin pariohjelmointi on XP-ohjelmoinnin erittäin merkittävä ominaispiirre. Pariohjelmointi käsittää kaikki ohjelmistotuotantoprosessin sisäiset aktiviteetit määrittelystä testaukseen ja koodin arviointiin, itse ohjelmoinnin ollessa vain yksi meneillään olevista aktiviteeteista. Astelsin & al. (2002) mukaan pariohjelmoinnissa on parempi todennäköisyys edetä tasaista vauhtia kuin yksin ohjelmoidessa. Toisen kokiessa ajatuskatkon on toisella todennäköisesti idea valmiina. Samoin pariohjelmointi tarjoaa XP-ohjelmoinnissa välttämätöntä jatkuvaa palautetta. Idean esittäminen joko ääneen tai koodissa antaa taustalla olevalle kumppanille mahdollisuuden välittömään palaut-

teeseen. Näin huonoja ideoita voidaan hylätä heti havaittaessa eikä vasta sen jälkeen, kun ne on jo toteutettu. Tietyllä tavalla tämän käytännön myötä koodatessa on samalla käynnissä jatkuva koodin tarkastus. Astels & al. (2002) väittävät, että pariohjelmoinnin nettotuotto on paljon suurempi kuin jos ohjelmoitaisiin yksittäin. Se tarjoaa myös kumulatiivisen vaikutuksen muihin ohjelmistotuotannon osa-alueisiin, kuten laadunvalvontaan. Suunnittelupäätökset, koodin tuottaminen, koodin arviointi, testaus, refaktorointi, parin ja koko tiimin sisäinen kommunikointi sekä parin ja tiimin opettaminen ovat asioita, joita täytyy joka tapauksessa tehdä ohjelmistotuotantoprojektissa, tehtiin ohjelmointia sitten pareittain tai yksin. Ongelmia pariohjelmoinnissa saattaa aiheuttaa Astelsin & al. (2002) mukaan yhteinen kehitysympäristö. Toisin sanoen se, että jotkut jäsenet ovat niin kiintyneitä johonkin työvälineeseen, että heidän on pakko totuttaa koko tiimi tämän välineen kannalle. Toinen ongelmien aiheuttaja voi olla jatkuva integrointi, jossa kaksi paria voi tehdä ristiriitaisia muutoksia samaan koodiin. Tällaista ei kuitenkaan Astelsin & al. (2002) mukaan tapahdu niin usein kuin pelätään tai ennakoidaan. Refaktoroinnin kautta toteutetaan ykinkertaisuuden periaatetta ja toisaalta saavutetaan joustavuutta muutosten suhteen. XP-projektissa koko tiimi omistaa kaiken koodin. Koodin yhteisomistus auttaa kehittäjiä ymmärtämään koko järjestelmää sekä aiheuttaa sen, ettei jokin järjestelmän osa jää yhden henkilön erikoisalueeksi, jolloin voi aiheutua viivästyksiä ja ongelmia esimerkiksi sairastapauksissa. Ohjelmointistandardin noudattaminen puolestaan on käytäntö, joka mahdollistaa kaikki kolme edellä mainittua käytäntöä.

RUP:issa järjestelmän toteuttaminen määritellään toteutuskäytännön avulla, jonka tarkoitus on koodin organisointi sekä komponenttien toteuttaminen, yksikkötestaus ja integrointi. Toteuttaminen RUP:issa, toisin kuin XP-ohjelmoinnissa, pohjaa ja on seurausta arkkitehtuurista. RUP:in toteutuskäytäntö eroaa XP-ohjelmoinnin vastaavasta ollen perinteisempi. Tietyn koodin osan omistaa tietty rooli. Esimerkiksi tietokantasuunnittelijan tehtävänä on tuottaa järjestelmän tietokanta. Myös koodin arvioiminen ja tarkastukset ovat RUP:issa määriteltyjä toimintoja, kun ne XP-ohjelmoinnissa oletetaan tapahtuviksi pariohjelmoinnin lomassa. Molemmissa tavoissa on hyvät ja huonot puolensa. XP:n malli antaa joustavuutta kehitystyöhön, sillä kukaan ei ole sidottu tietyn roolin vastuisiin, vaan kaikki ovat vastuussa kaikista toiminnoista. Toisaalta RUP:in malli, jossa jollakin roolilla on selkeästi vastuu jostakin tietystä toimenpiteestä, tarjoaa prosessille paremman hallittavuuden ja ennustettavuuden.

4.4 Testaus

XP-ohjelmointi on testauslähtöinen prosessi. Testausta tapahtuu jatkuvasti, minkä seurauksena myös palautetta järjestelmästä saadaan jatkuvasti. Testaaminen ennen koodausta on sellainen seikka, mikä merkittävästi erottaa XP-ohjelmoinnin vesiputousmallista ja myös RUP:ista. XP-ohjelmoinnissa testi kirjoitetaan ennen koodia, jolloin testi toimii myös määrityksenä tulevalle koodille ja osana ongelman ratkaisua. Kaikki testit automatisoidaan ja lisätään kasvavaan testisarjaan, jota ajetaan jatkuvasti. Se, pystyykö koodin testaamaan, autentikoi paitsi vaatimusten, myös koodin oikeellisuuden.

Testausta suoritetaan vähintään jokaisen integroinnin yhteydessä ja testauksen automatisointi on hyvin tärkeää integroinnin tehokkaan onnistumisen kannalta. Lisäksi jatkuva testaaminen tarjoaa runsaaasti palautetta järjestelmästä sekä paljastaa mahdolliset virheet välittömästi niiden ilmestyessä. Kunkin iteraation päätteeksi tuotettava julkistus testataan vielä erikseen hyväksymistestein, joiden sisällön asiakas on suunnitellut.

Kuten XP-ohjelmoinnissa myöskään RUP:issa testaus ei ole erillinen vaihe tai toiminto, vaan jatkuvaa muun toiminnan kanssa rinnakkain tapahtuvaa toimintaa, jonka tarkoituksena on paljastaa virheet aikaisessa vaiheessa ja tarjota palautetta kehityksen alla olevasta järjestelmästä. Merkittävin ero testauksessa XP-ohjelmoinnin ja RUP:in välillä on se, että XP-ohjelmoinnissa testit suunnitellaan ennen koodausta ja ne toimivat koodin määrityksinä ja pohjana toteutettavalle koodille. RUP:issa testaus on enemmän perinteisen vesiputousmallin mukaista yksikkö-, integrointi- ja järjestelmätesteistä koostuvaa, vaikkakin rinnakkain kehitystyön kanssa tapahtuvaa ja jatkuvaa toimintaa. RUP:issa myös testauksen suorittaminen, osallistuvat roolit sekä tarvittavat dokumentit on testauskäytännössä tarkoin määritelty.

4.5 Skaalautuvuus

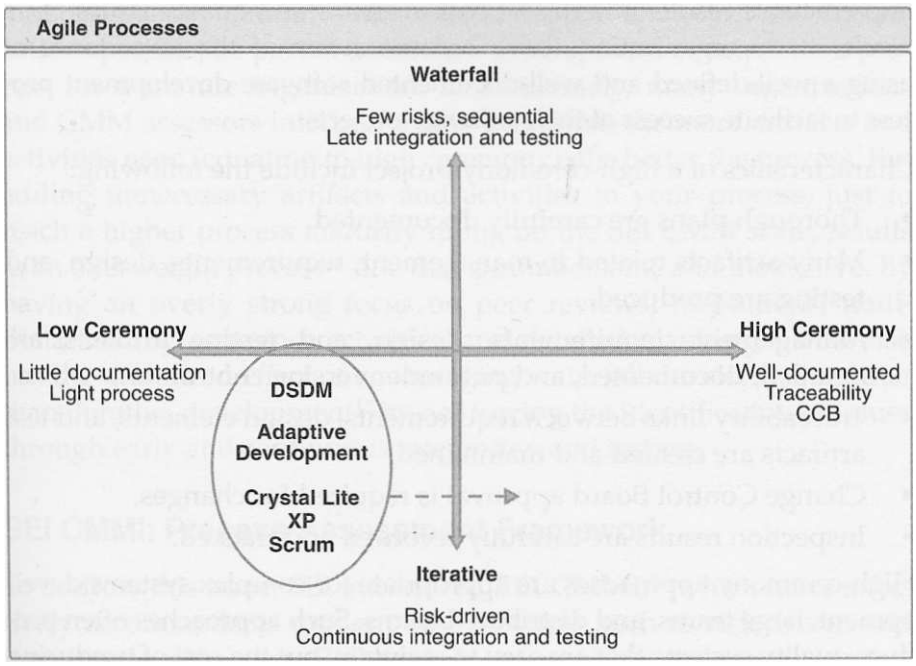
XP-ohjelmoinnin ydinarvoista suoran kommunikaation arvostaminen aiheuttaa jo yksin sen, että tiimin koon on oltava kohtuullisen pieni. Suora kommunikaatio muuttuu huomattavasti tehottomammaksi tiimin koon kasvaessa liian suureksi. Sitä, minkäkokoinen tiimi on liian suuri, ei voi suoraan sanoa, mutta esimerkiksi Maurer & Martel (2002) esittävät XP-ohjelmoinnin toimivan parhaiten 5-15 henkeä käsit-

tävissä tiimeissä. XP-ohjelmoinnin sopiminen parhaiten pieniin projekteihin johtuu myös XP-ohjelmoinnissa noudatettavasta koodin yhteisomistuksesta. Kun projektin kaikkien jäsenten tulee tuntea kaikki projektin koodi, on luonnollista, että projektin on oltava hahmotettavissa olevan kokoinen. Astels & al. (2002) esittävät kuitenkin, että myös XP-ohjelmointia voidaan skaalata. Suuret tiimit ja projektit pitää vain voida pilkkoa osiksi, joita kohdellaan itsenäisinä projekteina, joiden sisällä sovelletaan XP-ohjelmoinnin käytäntöjä. XP-ohjelmointi sinänsä ei kuitenkaan skaalaudu. Parhaimmillaan XP-ohjelmointi on sellaisissa tilanteissa, joissa vaatimukset eivät ole alussa selvillä tai vaatimusten voi odottaa muuttuvan projektin edetessä. Tällöin XP-ohjelmoinnin kyky reagoida nopeasti muutoksiin auttaa saavuttamaan hyötyä. Samoin se, ettei mallinnukseen, suunnitteluun ja dokumentointiin käytetä hukka-aikaa on hyödyksi nimenomaan silloin, kun ennustaminen olisi vaikeaa joka tapauksessa.

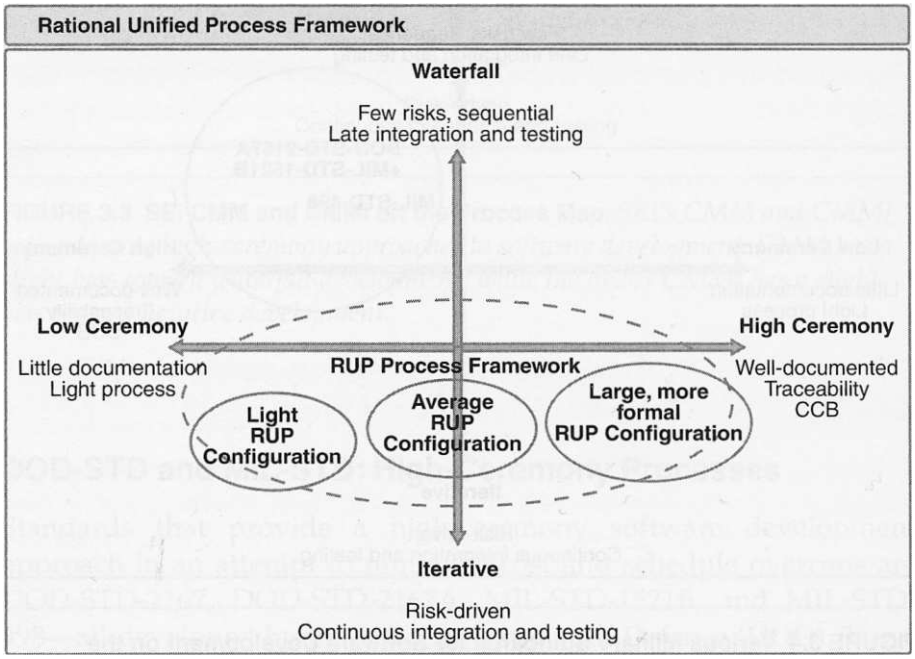
RUP on enemmän arkkitehtuuri- ja dokumentointikeskeinen prosessi kuin XP-ohjelmointi. XP-ohjelmoinnin suosima suora kommunikointi voidaan suurilta osin hoitaa dokumenttien ja mallien avulla. Nämä seikat mahdollistavat mielestäni sen, ettei tiimin ja projektin koolle tarvitse asettaa samanlaisia rajoitteita kuin XP-ohjelmoinnissa. RUP soveltuu myös pienille tiimeille (Pollice & al., 2004), mutta se ei kohtaa samanlaisia rajoitteita, kuin XP tiimin tai projektin koon kasvaessa. Roolit ja niiden vastuut, määrätyt tuotokset ja työnkulut mahdollistavat suuremmankin tiimin tai projektin pitämisen hallinnassa. RUP:in prosessikehys, mukaanotettavat tai poisjätettävät elementit mahdollistavat prosessin hyvän skaalautumisen. Myös RUP reagoi virheisiin ja muutoksiin nopeasti iteratiivisen ja lisäävän lähestymistavan ansiosta.

Kuvat 33 ja 34 kuvaavat näiden kahden prosessin erot skaalautuvuudessa ja nimenomaan RUP:in prosessikehyksen vaikutuksen tähän. Kuten kuvasta 33 nähdään, muiden notkeiden menetelmien kanssa XP asettuu diagrammissa kohtaan, jossa prossessin ja dokumentaation keveydellä on suuri painoarvo. Samoin XP asettuu vahvasti iteratiiviselle puolelle vesiputousmallin sijasta.

RUP toisaalta, kuten kuva 34 osoittaa, on skaalautuvampi prosessikehyksensä ansiosta. RUP on mahdollista konfiguroida kevyeksi, vähän dokumentaatiota sisältäväksi prosessiksi, jossa iteratiivisuudella on suuri painoarvo. Toisaalta se voidaan muokata raskaaksi, muodolliseksi, dokumentointiorientoituneeksi prosessiksi, jossa lisäksi iteratiivisuuden painoarvo voi vaihdella.



Kuva 33: XP-ohjelmoinnin sijoittuminen (Kroll & Kruchten, 2003).



Kuva 34: RUP:in sijoittuminen (Kroll & Kruchten, 2003).

4.6 Yhteenveto

XP-ohjelmoinnin tarjoamat periaatteet yksinkertaisuudesta ja kommunikaatiosta kuulostavat kauniilta ideaaleilta. Pienehkössä projektissa niiden avulla varmasti onkin mahdollista tuottaa toimivia järjestelmiä. Iteratiivinen lähestymistapa, jossa asiakas on koko ajan läsnä auttaa myös huomaamaan virheet kehitystyössä jo aikaisessa vaiheessa. Kehityksen pitämiseen oikeassa kurssissa auttavat varmasti myös muut käytännöt, kuten pienet tuoteversiot, joiden seurauksena saadaan jatkuvasti paljon palautetta. XP-ohjelmoinnin määrittely jättää kuitenkin monia kysymyksiä vastaamatta tai vastaa niihin vain osittain. Projektin hallittavuus kärsii varmasti tiimin tai järjestelmän koon kasvaessa kriittisen pisteen yli, joten ei ole suositeltavaa ottaa XP-ohjelmointia käyttöön liian suuriin puitteisiin. Toisaalta se puolustaa vamasti paikkaansa nimenomaan sellaisissa ympäristöissä, joihin se on suunniteltu. Toisin sanoen pienissä tiimeissä, pienissä projekteissa, tilanteissa, joissa etukäteissuunnittelu ja arkkitehtuuri eivät ole tärkeitä ja joissa dokumentoinnille ei aseteta suurta arvoa.

Toisaalta RUP voidaan joustavasti muokata minkälaiseen tahansa projektiin sopivaksi prosessikehyksensä ansiosta. RUP:in valmiiksi määritellyt roolit ja vastuut, toiminnot, tuotokset, käytännöt, työnkulut ynnä muut prosessin elementit tarjoavat paljon mahdollisuuksia prosessin mukauttamiseksi. Joko dokumentointiorientoituneeksi raskaaksi prosessiksi suuriin organisaatioihin ja projekteihin tai kevyeksi, minimaalisesti dokumentaatiota sisältäväksi prosessiksi. RUP:in läheinen yhteys UML-kieleen tekee siitä ja sen tuotoksista ainakin mallinnusta suorittavien ohjelmistokehittäjien piirissä varsin universaalin ja käyttökelpoisen prosessin. RUP tarjoaa kaikkeen toimintaan selviä ohjeita ja malleja, kun taas XP-ohjelmoinnissa monet asiat jätetään projekti-, henkilö- ja tilannekohtaisesti päätettäviksi. RUP:in hallittavuus ainakin teoriassa on näin ollen parempi.

Loppujen lopuksi se, mikä prosessi mihinkin projektiin olisi syytä valita, on projektikohtaisesti päätettävä seikka. Kummallakin prosessilla on hyvät ja huonot puolensa, ja riippuu täysin projektin, henkilöstön ja organisaation erityispiirteistä, mikä lähestymistapa milloinkin voi tuottaa parhaan tuloksen.

5 Loppusanat

Tässä tutkielmassa käsiteltiin kahta ohjelmistotuotantoprosessia, XP-ohjelmointia ja Rationalin Unified-prosessia. XP-ohjelmointi kuuluu notkeisiin menetelmiin, joiden tarkoituksena on tarjota kevyt ja tehokas ratkaisu ohjelmistojen tuottamiseen. Verrattuna perinteiseen ohjelmistotuotantoprosessin vesiputousmalliin XP-ohjelmointi eroaa merkittävästi siinä, ettei suurta painoarvoa ole etukäteissuunnittelulla ja dokumentoinnilla. Sen sijaan keskitytään tehokkaaseen toteutukseen muuttuvien vaatimusten ympäristössä. Iteratiivisuuteen ja lisäävään lähestymistapaan sekä jatkuvaan kommunikaatioon ja palautteeseen pohjaavana prosessina XP-ohjelmointi suhtautuu muutoksiin perinteistä ohjelmistotuotannon mallia sulavammin. Määrittelyltään XP-ohjelmointi on hyvin väljä ja epämuodollinen, hajanainen kokoelma käytäntöjä ja arvoja, jotka toimivat pikemminkin ohjenuorina kuin selvinä sääntöinä siitä, miten jonkin asian tulisi olla. Sellaisena XP-ohjelmointi sopii erinomaisesti tilanteisiin, joissa ohjelmistotuotantoprosessin ei haluta olevan liian raskas.

Myös RUP perustuu iteratiivisuuteen ja lisäävään lähestymistapaan mutta on lähempänä vesiputousmallia siinä, että etukäteissuunnittelulla, arkkitehtuurilla sekä komponenttipohjaisella kehityksellä on suuri painoarvo. Samoin kuin XP-ohjelmoinnissa, RUP:issa pyritään sopeutumaan muuttuviin vaatimuksiin joustavasti. Suuri ero verrattaesssa XP-ohjelmointiin on RUP:in tarjoama prosessikehys, joka tekee RUP:ista helpommin skaalautuvan prosessin erilaisille tiimeille ja erikokoisille projekteille.

Muulloin kuin toteutettaessa täysin triviaalia tai hyvin pientä ohjelmistotuotantoprojektia, on ohjelmistotuotantoa harjoittavan organisaation käytännössä pakko käyttää jotakin ohjelmistotuotantoprosessia. On henkilö-, tiimi-, organisaatio- ja projektikohtaista, kumpi tässä tutkielmassa esitellyistä prosesseista soveltuu paremmin, vai onko organisaation syytä käyttää jotakin muuta ohjelmistotuotantoprosessia. Loppujen lopuksi kaikilla prosesseilla on kuitenkin sama päämäärä: laadukkaiden ohjelmistojen tuottaminen tehokkaasti. Vain eri prosessien keinot päämäärään pääsemiseksi voivat olla erilaisia.

Viitteet

Ambler, S. (2002) *Agile Modeling*. John Wiley & Sons, Inc., New York.

Arlow, J., Neustadt, I. (2002) *UML and the Unified Process*. Pearson Education Limited., Great Britain.

Astels, D., Miller, G., Novak, M. (2002) *A practical guide to eXtreme Programming*. Prentice Hall PTR, Upper Saddle River, NJ 07458.

Beck, K. (2000) *Extreme programming explained*. Addison Wesley Longman, Inc., Reading, MA.

Beck, K., Fowler, M. (2001) *Planning Extreme programming*. Addison Wesley Longman.

Bergström, S., Råberg, L. (2004) *Adopting the rational unified process : a success with the RUP*. Pearson Education Inc., Boston, MA 02116.

English, A. (2002) Extreme programming: It's worth a look. *IT professional* **4**(3), 48-50.

Fenton, N., Pfleeger, S. (1997) *Software metrics: a rigorous and practical approach*. PWS Publishing Company, Boston, MA 02116-4324.

Firesmith, D., Henderson-Sellers, B. (2002) *The OPEN Process Framework*. Pearson Education Limited, Great Britain.

Fowler, M. (1999) *Refactoring: Improving the design of existing code*. Addison Wesley Longman.

Garlan, D, Shaw, M. (1996) *Software architecture - Perspectives on an Emerging Discipline*. Prentice-Hall, Upper Saddle River, NJ.

Hull, M.E.C, Taylor, P.S, Hanna J.R.P, Millar, R.J. (2002) Software development processes - an assesment. *Information and software technology* **44**(2002), 1-12.

IBM (2004) *IBM*. WWW-sivusto, `http://www.ibm.com` (1.9.2004).

Jacobson, I., Booch, G., Rumbaugh, J. (1999) *The Unified Software Development Process*. Addison Wesley Longman, U.S.A.

Jeffries, R. (2001) *Yksityinen kommunikaatio Astelsin & al. kanssa*.

Kroll, P., Kruchten, P. (2003) *The Rational Unified Process made easy:a practitioner's guide to RUP*. Pearson Education Inc., Boston, MA 02116.

Kruchten, P. (2004) *The Rational unified process : an introduction*. Pearson Education Inc., Boston, MA 02116.

Maurer, F., Martel, S. (2002) Extreme programming. Rapid development for Web-based applications. *IEEE Internet Computing* **6**(1), 86-90.

Newkirk, J., Martin, R.C. (2001) *Extreme programming in practice*. Addison Wesley.

Newkirk, J. (2002) Introduction to agile processes and extreme programming. *Proceedings of the 25th international conference on Software engineering* ACM press, New York, NY,USA, 695-696.

OMG (2004) *Object Management Group*. WWW-sivusto, `http://www.omg.org` (1.9.2004).

Paulk, M.C. (2001) Extreme programming from the CMM perspective. *IEEE Software* **18**(6), 19-26.

Pfleeger, S.L. (2001) *Software engineering : theory and practice*. Prentice Hall, Upper Saddle River, New Jersey 07458.

Pollice, G., Augustine, L., Lowe, C., Madhur, J., (2004) *Software development for small teams : a RUP-centric approach*. Pearson Education Inc., Boston, MA 02116.

Poole, C., Huisman, J.W. (2001) Using extreme programming in a maintenance environment. *IEEE Software* **18**(6), 42-50.

Pressman, R.S. (2000) *Software engineering : a practitioner's approach*. McGraw-Hill Publishing Company, England.

Rumbaugh, J., Jacobson, I., Booch, G. (1999) *The Unified Modeling Language Reference Manul*. Addison-Wesley, Reading, MA.

Steinberg, D., Palmer, D. (2004) *Extreme software engineering : a hands-on approach*. Pearson Education Inc., Upper Saddle River, New Jersey.

Sun (2004) *Java Technology*. WWW-sivusto, `http://java.sun.com` (2.9.2004).

Wake, W.C. (2002) *Extreme Programming Explored*. Addison Wesley Longman.

XProgramming.com (2004) *an Extreme programming resource*. WWW-sivusto, `http://www.xprogramming.com` (12.2.2004).