Lappeenrannan teknillinen korkeakoulu
Tietotekniikan osasto
Tietoliikennetekniikan laitos

OLIOPOHJAISEN OHJELMISTON KEHITYSPROSESSI: CASE TRADEXPRESS MAPPER

Tekijä: Jari Jauhiainen
Ohjaaja: Kari Välimäki
Tarkastaja: Prof. Jari Porras

Lappeenranta 10.02.2003

TIIVISTELMÄ

Tekijä: Jauhiainen, Jari Kalervo

Nimi: Oliopohjaisen ohjelmiston kehitysprosessi

Osasto: Tietotekniikan osasto

Vuosi: 2001

Paikka: Lappeenranta

Diplomityö, Lappeenrannan teknillinen korkeakoulu, 45 sivua, 9 kuvaa, 1 taulukko

Tarkastaja: Prof. TkT Jari Porras

Avainsanat: Oliopohjaisuus, Ohjelmistonkehitys, Prosessit, Laatu, Asiakasvaatimukset, Analysointi, Suunnittelu

Keywords Object Oriented, Software development, Processes, Quality, Customer requirements, Analysing, Designing

Työ kartoittaa yleisemmin ohjelmistonkehitysprosesseille asetettavia vaatimuksia ja selvittää yksityiskohtaisemmin niiden soveltamista ja vaikutusta uusien ominaisuuksien suunnittelussa Soneran TradeXpress Mapper ohjelmistoon. Kyseinen työkalu mahdollistaa muun muassa EDI-viestien muokkaamisen graafisesti.

Työn tuloksena syntyi selvitys ohjelmistonkehitysprosessien perusteista ja niiden soveltamisessa oliopohjaiseen systeemityöhön. Prosessin avulla voitiin ohjelma tehdä tarkemmin asiakkaan vaatimusten mukaiseksi ja uusien ominaisuuksien lisääminen onnistui helposti.

ABSTRACT

Author: Jari Jauhiainen

Title: Object-oriented software process

Department: Department of Information Technology

Year: 2001

Location: Lappeenranta

Master's Thesis, Lappeenranta University of Technology, 45 pages, 9 figures, 1 table

Supervisor: Prof. D.Sc. (Tech) Jari Porras

Keywords Object Oriented, Software development, Processes, Quality, Customer requirements, Analysing, Designing

This Master's thesis explores *software development processes* and their application to the development of graphical tool for Sonera TradeXpress-environment. This tool makes possible, for example, to edit EDI messaages graphically.

This work is a result of a research on software processes and how they affect to the object oriented software development. As a result processes were found an effective way to create the application as required.

Alkusanat

Tämä diplomityö on tehty Sonera System Software-yksikön SCP-osastolle Lappeenrannassa. Työhön kuuluva tutkimus ja suunnittelu liittyvät TradeXpress-nimisen sähköisen kaupankäynnin ohjelmiston tuotekehitykseen.

Kiitän työni ohjaajaa Kari Välimäkeä asiantuntevista neuvoista työni kuluessa, sekä osastopäällikkö Harri Tumeliusta mahdollisuudesta työni loppuun viemiseen . Lisäksi tarkastaja professori Jari Porras on myös osoittanut pitkämielisyyttä kuunnellessaan selityksiä aikataulujen venymisistä. Kiitokset siitä.

Ehdottomasti kiitoksen ansaitsee myös Kari Poutiainen, joka on kanssani jaksanut pohtia oliomallien yksityiskohtia. Lisäksi Kirsi Hietaselle kiitokset liiketoimintaprosessien selvittämisestä tietotekniikan opiskelijalle.

Termit ja lyhenteet

CMM	Capability Maturity Model
DLL	Dynamic Link Library
EDI	Electronic Data Interchange
MDI	Multiple Document Interface
MFC	Microsoft Foundation Class
ODBC	Open Database Connectivity
OLE	Object Linking and Embedding
OMG	Object Management Group
OMT	Object Modeling Technology
OOSE	Object Oriented Software Engineering
RTF	Rich Text Format
SA/SD	System Analysis/ System Design
SPI	Software Process Improvement
SPICE	Software Process Improvement and Capability dEtermination
UML	Unified Modeling Language
XML	eXtended Markup Language

SISÄLLYSLUETTELO

1. JOHDANTO

Ohjelmistonkehitysprosessit ovat tulleet suurempien kokonaisuuksien hallinnan tarpeen myötä entistä laajemmin käyttöön. Kaikkien ohjelmiston yksityiskohtien muistaminen ja hallinta on usein vaikeaa, jollei mahdotonta, yhdelle suunnittelijalle. Kokonaisuuksien ja laadun hallinta vaatii laajaa perehtymistä paitsi teknologioihin, myös prosessiin, jolla voidaan varmistaa ohjelmistojen oikeat ominaisuudet, luotettavuus, hinta ja aikataulujen pitävyys /1/, sekä mahdollistetaan useamman henkilön työpanoksen saumaton liittäminen toisiinsa ohjelmistoprojektissa.

Uudelleenkäyttö on yksi tehokkuuden lisäämisen edellytyksistä. Uudelleenkäyttö vähentää sekä varsinaisia kehityskustannuksia, että testausaikaa. Prosessit kuvaavat keinot, joilla uudelleenkäyttöä voidaan lisätä. Koodin lisäksi myös muiden työn tulosten uudelleenkäyttö on lisääntymässä. Uusia menetelmiä on tulossa uudelleenkäytön aikaistamiseksi ohjelmistoprosessissa. Tämä säästää kehityskustannuksissa. Uudelleenkäytöksi voidaan mieltää myös samanlaiset työmenetelmät projektista toiseen.

Prosessien eduiksi voidaan myös todeta projektien ennustettavuuden paraneminen. Prosessien avulla kerätyn tiedon avulla voidaan ennustaa esimerkiksi projektissa tarvittava työmäärä tarkemmin, kuin pelkästään arvailemalla.

Prosessien ominaisuuksia ovat edellisten lisäksi mitattavuus, parannettavuus ja mahdollisuus kytkeä ohjelmistojen kehitys selkeäksi osaksi liiketoimintaa. Prosessit lisäksi yleensä pienentävät kuluja, joten niiden käyttö säästää yrityksen tuotekehityskustannuksia.

Ohjelmistotekniikka käsittää koko ohjelmistonkehitysprosessin. Ohjelmiston kehitystä aina asiakkaan toiveista lopulliseen testattuun tuotteeseen asti voidaan kutsua myös ohjelmistotuotannoksi. Ohjelmistotekniikka on siinä mielessä matematiikan tapainen geneerinen tekniikan alue, ettei sillä ole mitään varsinaista sovellusaluetta. /2/ Ohjelmistotekniikka tarjoaa siis keinot käyttää kunkin alueen asiantuntijoita ohjelmistojen kehityksessä.

Sähköinen liiketietojen vaihto (electronic data interchange, EDI) on tarkoitettu yritysten välisen sähköisen kaupan välineeksi. Se on kehitetty jo 1980-luvulla, mutta on sähköisten kauppapaikkojen myötä noussut uudestaan esille. Sillä voidaan järjestää kauppapaikan ja esimerkiksi tuotteen valmistajan välinen tiedonsiirto turvallisesti ja standardin mukaisesti, välittämättä ohjelmistojen toimintaympäristöistä.

2 LIIKETOIMINTA- JA OHJELMISTONKEHITYSPROSESSIT

2.1 Prosessiajattelu liiketoiminnassa

Yksittäisistä tehtävistä ja toiminnoista koostuvaa toimintoketjua kutsutaan liiketoimintaprosessiksi. Näistä prosesseista voidaan esimerkkinä mainita mm. tuotekehitys, tilauksen tekeminen/tuotteen toimittaminen ja markkinointisuunnitelman tekeminen.

Prosesseja voidaan ryhmitellä usealla eri tavalla, mutta tässä työssä käsitellään prosesseja *ydinprosesseina*, jotka ovat yrityksen ja sen avainsidosryhmien toimintaa läpileikkaavia toimintoketjuja, ja *aliprosesseina*, joista ydinprosessit koostuvat. /3/ Ydinprosessiksi voidaan kutsua esim. tuotekehitysprosessia ja aliprosessiksi edelliselle ohjelmistonkehitysprosessia.

Liiketoimintojen prosessoimisella voidaan nähdä neljä selkeää tavoitetta, jotka ovat:

- hyvä taloudellinen tulos
- asiakkaiden tyytyväisyys
- korkea tuottavuus
- oman henkilöstön tyytyväisyys

Aikaisemmin keskityttiin kustannustehokkuuteen, mutta nyt nähdään tärkeänä myös nopeus ja joustavuus. Lisäksi asiakas ja toimittajat nähdään nyt pikemminkin yhteistyökumppaneina entisen välttämättömän pahan sijaan. Organisaation menettelyjen ja työryhmien kehittäminen on myös katsottu tärkeäksi prosessijohtamisessa. /4/

Prosessien suorituskyvyllä tarkoitetaan yrityksen (ja yksikön) kykyä tuottaa avainsidosryhmiensä odotuksia vastaavia tuloksia. Avainsidosryhmiksi

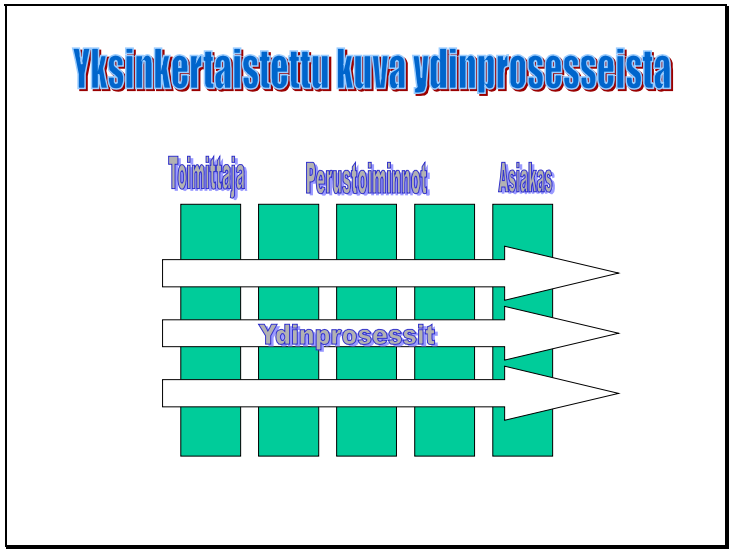
mielletään nykyisin omistajien lisäksi asiakkaat (myös sisäiset) ja oma henkilöstö. Prosessijohtamisessa on siis olennaista, että tavoitteet ja mittarit asetetaan paitsi taloudellisten arvojen perusteella, myös laadun ja henkilöstön motivaation ylläpitämiseksi. Toinen olennainen asia prosessiajattelussa on, että suorituskykyä mitataan asiakaslähtöisesti perinteisen funktionaalisen organisoinnin sijaan. /3/

Eräs avainkysymys prosessiajattelussa on, että kuinka erikoistunut asiantuntemus – yrityksen avainosaaminen – tehokkaimmalla tavalla hallitaan. Varsinainen ydinosaaminen liittyy yleensä suuressakin yrityksessä hyvin pieneen avainhenkilöiden ryhmään. Useimmat muut henkilöt eivät kasvata ydinosaamista, vaan hyödyntävät sitä työtehtävissään. Tällöin on tärkeää välittää avainhenkilöiden osaaminen mahdollisimman tehokkaalla tavalla muun organisaation käyttöön./3/ Prosessit mahdollistavat osaajien työpanoksen hyödyntämisen millä tahansa organisaation osa-alueella tarjoamansa käyttöliittymän avulla. Tämä mahdollistaa erityisten osaamiskeskusten perustamisen. Jokaisella yksiköllä ei tarvitse olla esimerkiksi omaa ohjelmistonkehitysryhmää, vaan voidaan perustaa erillinen yksikkö, joka kehittää kaikki yrityksen tarvitsemat sovellukset ja palvelut.

Prosessiajattelussa pohditaan siis sitä, miten asiat tehdään, entisen tuotekeskeisen mitä tehdään ajattelun sijaan. Tuotteen sijaan pyritään oma toiminta ajattelemaan osana asiakkaan liiketoimintaprosesseja. Tyypillisesti voidaan ajatella oma kehitystyö asiakkaan uuden liiketoiminnan tai toimintatapojen kehittämisen ja varsinaisen liiketoiminnan väliseksi osaksi asiakkaan toimintaa. Tämä mahdollistaa nopean reagoinnin markkinoiden ja asiakkaiden toiveiden muutoksiin./5/

Liiketoiminnan ydinprosessit kuvataan yleensä karkeimmalla tasolla prosessikartalla. Kartta kuvaa yrityksen ja sen sidosryhmien ydinfunktiot ja niitä läpileikkaavat ydinprosessit.

Kuvassa 2.1 on kuvattu prosessikartta ydinprosesseista, jotka leikkaavat organisaation (myös sidosryhmien) väliset rajat. Kuvassa vasemmalla on esitetty toimittajien toiminta. Ydinprosessit käsittelevät toimittajan kanssa käytävän yhteistyön ja siirtävät toimitukset organisaation käyttöön. Organisaation perustoiminnot käsittelevät sekä toimitettua, että itse kehitettyä materiaalia. Lopuksi prosessit käsittelevät tuloksen toimittamisen edelleen asiakkaalle.



Kuva 2.1 Ydinprosessit

2.2 Ohjelmistonkehitysprosessit liiketoiminnassa

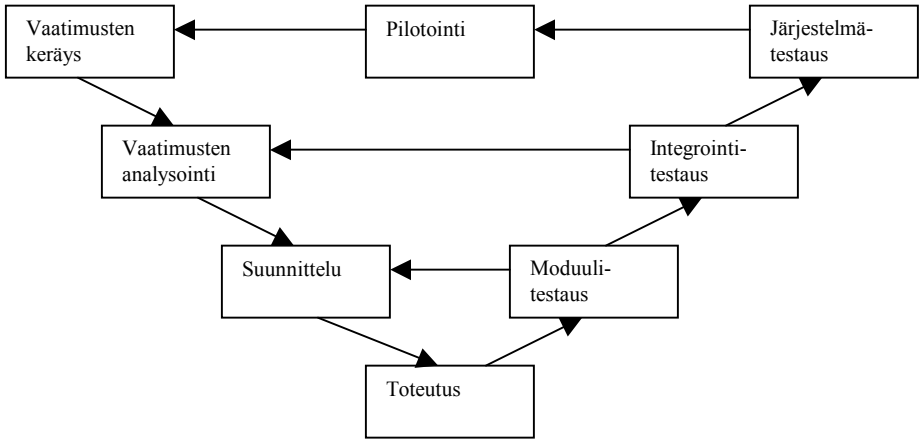
Prosessit sisältävät päätöspisteitä, joissa päätetään jatketaanko työn alla olevaa projektia, tehdäänkö aikatauluihin tai tavoitteisiin muutoksia jne. Päätöspisteet voivat sijaita esimerkiksi eri yksiköiden välisen vastuualueen rajalla, tai työn

vaiheiden välissä. Ohjelmistonkehitysprosessi sijaitsee joidenkin kahden tuotekehitysprosessin päätöspisteen välillä. Sillä itsellään on omat päätöspisteensä, jotka antavat tietoa tuotekehitysprosessille jatkoa ajatellen.

Ohjelmistoprojekti on yksittäinen prosessia hyödyntävä tehtävä. Prosessin päätöspisteet ovatkin yksittäisiä projekteja varten. Prosessi itsessään saattaa olla vaiheistettu useaksi eri projektiksi. Esimerkiksi määrittely, suunnittelu ja toteutus saattavat olla erillisiä projekteja, tai yksi alku- ja päätepisteet sisältävä tehtävä.

Prosessi määrittelee projektien toimintatavat ja mahdolliset keskeytyspisteet, jos projekti katsotaan esimerkiksi liian kalliiksi tai muuten toteutuskelvottomaksi. Lisäksi prosessin tehtäviä ovat mm. muutosten hallinta, tehtävien vaatimien roolien määrittäminen, vaadittavien resurssien määrittely yms. Prosessin tehtävä on siis ohjata projekteja.

Ohjelmistoprosessi vaiheistaa ohjelmistokehityksen. Systeemityön kannalta vaiheistus esitetään vaihejakomalleina. Näitä vaihejakomalleja ovat esimerkiksi vesiputousmalli ja erilaiset iteroivat mallit. Tässä työssä käsitellään V-mallia. V-malli on kehitetty vesiputousmallista ja siinä on testaus lisätty vesiputousmallin oikeaksi haaraksi. Jokaista kehityshaaran vaihetta vastaa yksi testausvaihe. Testausta tutkittaessa on todettu, että yhdellä testauskierroksella löydetään ohjelmistosta n. 60% jäljelläolevista virheistä. V-mallissa siis testaus määritellään jo kehityshaaran dokumenteissa. V-malli on esitetty kuvassa 2.2.



Kuva 2.2 V-malli

Tekniseltä kannalta katsottuna voidaan ohjelmistoprosessi nähdä dokumenttien tuottamisena. Itse asiassa lähdekoodikin voidaan katsoa dokumentiksi, joka kertoo kääntäjälle, millainen ohjelma sen tulee tuottaa. Ohjelmistoprosessi onkin siis vaiheittainen kuvaus järjestelmän vaatimuksista niiden toteuttamiseksi ja testaamiseksi./2/

Jokaisella tasolla tuotetaan dokumentti seuraavan tason syötteeksi. Näiden dokumenttien työstämiseksi on kehitetty useita kuvaustekniikoita, joista oliomallintamiseen voidaan käyttää esimerkiksi Unified Modeling Language-tekniikkaa (myöhemmin UML) tai Object-Oriented Analysis/Object-Oriented Design-tekniikoita. Oliopohjaisille tekniikoille on yhteistä niiden käyttötapauslähtöisyys asiakastarpeiden analysointivaiheessa, josta myöhemmin lisää.

Yhteenvetona ohjelmistonkehitysprosesseista voidaan vetää se, että kyseinen prosessi toimii kuten aiemmin käsitellyt liiketoimintaprosessit. Se saa syötteen tuotekehitysprosessilta, joka määrittelee, mitä tarvitaan. Asiakkaana sillä on sama prosessi, joka saa valmiin ohjelmiston, joka tuotteistetaan ja saadaan lähetettyä asiakkaalle tai toisen yksikön käyttöön.

2.3 Ohjelmistonkehitysprosessien tarkoitus

Ohjelmistonkehitysprosessia on usein käytännössä mahdoton täysin noudattaa. Käytäntö on osoittanut, että läheskään aina jokainen ennalta suunniteltu vaihe ei ole järkevästi perusteltavissa. Ne ovat usein intuitiivisesti valittuja, eikä vaiheiden orjallinen seuraaminen ole aina järkevää. Seuraavassa on muutamia syitä siihen, miksi prosessien liian tarkka seuraaminen saattaa aiheuttaa ongelmia./2/

- Ohjelmistolle asetettavat vaatimukset eivät ole täysin tunnettuja alkuvaiheessa.
- Vaikka vaatimukset tunnettaisiinkin, monet toteutukseen liittyvät seikat selviävät vasta projektin aikana.
- Vaikka kaikki tosiseikat olisivatkin tiedossa jo alussa, tosiseikkoja on niin paljon, että ihminen ei pysty käsittelemään niitä virheettömästi.
- Vaikka tosiseikat pystyisikin käsittelemään virheettömästi, ne voivat muuttua ulkoisista syistä.
- Ihminen takertuu aikaisemmin oppimiinsa ratkaisuihin, jolloin rationaalisesti perusteltavissa oleva ratkaisu voi jäädä huomaamatta.
- Aikaisemmin kirjoitettujen ohjelmistojen uudelleenkäyttö johtaa myös usein omituisiin ratkaisuihin.

Näistäkin syistä huolimatta ohjelmistoprosessien kehittäminen ja ylläpito on järkevää. Prosessimallin mahdollisimman tarkkaan seuraamiseen motivoivia syitä. /2/

- Ohjelmistonkehitysprosessi antaa ohjeita siitä, mitä kussakin vaiheessa tulee tehdä.

- Ihmisten on helpompi siirtyä projektista toiseen, kun kaikkien projektien toimintatavat muistuttavat toisiaan ja dokumentaatio on kunnossa.
- Kun tiedetään, mitä prosessissa tulisi tapahtua, on projektin suunnittelu ja seuranta helpompaa.
- Ulkopuolisen arvioijan on helpompi arvioida projektin tilannetta.

Edellä olevan nojalla on siis tärkeää, että toiminnassa ja dokumentaatiossa seurataan prosessia mahdollisimman tarkasti. Projektin tavoitteena olevat tulokset pystytään hallitsemaan ja niiden tila arvioimaan paremmin, mikäli dokumentaatio on tuotettu sovitulla tavalla ja sovitussa laajuudessa. Projektien erilaisuudesta huolimatta niissä on riittävästi yhteisiä piirteitä, että prosessoinnin hyödyt tulevat esille.

Ohjelmistoprosessien tarkoitukseksi voidaan myös todeta laadunvarmistus. Ohjelmistonkehityksen laatua mitataan nimenomaan prosessien toiminnan perusteella. Prosessit mahdollistavat mittatiedon keräämisen ja tämän hyödyntämisen historiatiedon perusteella. Tämä edellyttää sitä, että prosessit on suunniteltu järkeviksi ja ovat siten noudatettavissa.

2.4 OHJELMISTON LAATU

Ohjelmistoprosessien kehittämisen tavoitteena on saavuttaa tietty kyvykkyys. Prosessointi ei ole itsetarkoitus, vaan sillä tavoitellaan toimintojen ja lopputuloksen laadun ja hallittavuuden parantamista. Ohjelmistoprosessin kyvykkyys kuvaa siis menetelmien kyvykkyyttä. Menetelmien kehittyessä myös tuotteen laatu paranee parempien testausmenetelmien ja aikataulujen pitävyyden myötä, eli on siis aikaa myös testata. /10/

2.4.1 Capability Maturity Model (CMM)

Capability Maturity Model on Carnegie Mellon University:n Software Engineering Institute-yksikön Mitre Corporation:in avustuksella kehittämä ohjelmistojen laadun arviointimenetelmä. Kehitystyö aloitettiin 1986. Vuonna 1987 julkaistiin lyhyt kuvaus ohjelmistoprosessien kypsyysasteesta ja kyselylomake. /10/

Vuonna 1991 esiteltiin CMM:n 1. versio. CMM perustuu neljän vuoden aikana hankittuun tietoon ja ohjelmistoteollisuudelta saatuun palautteeseen. Lopullisesti CMM 1.0 hyväksyttiin työryhmässä huhtikuussa 1992. Vuonna 1993 julkaistiin versio 1.1. /10/

CMM perustuu tiettyjen prosessiryhmien olemassaoloon. Kypsymättömissä organisaatioissa prosessit ovat ohjelmiston tekijöiden improvisoimia kulloisessakin tilanteessa. Jos prosessi onkin määritelty, sitä ei noudateta. Tällaisissa organisaatioissa johtajien työ on välittömien ongelmien ratkaisemista. Aikataulut ja budjetit ylitetään säännöllisesti. /10/

Kypsymättömissä organisaatioissa ei ole perustaa tuotteen laadun arviointiin. Tuotteen laatuun liittyvät katselmoinnit ja testaukset jätetään tekemättä, koska aikataulut ylittyvät. /10/

Kypsissä organisaatioissa koko ohjelmistotuotanto ja johtaminen on prosessoitu. Ohjelmistoprosessit on sisäistetty koko organisaatiossa niin vanhojen, kuin uusienkin työntekijöiden toimesta ja aktiviteetit ovat prosessien mukaisia. Prosesseja seurataan ja niitä kehitetään edelleen. Roolit ja vastuut määritellyissä prosesseissa ovat selkeät projekteissa ja koko organisaatiossa. /10/

CMM toi ensimmäisen mittarin, jolla voitiin mitata ohjelmistoprosessien kypsyyttä. Kypsyys esitettiin viidellä tasolla, joista ensimmäinen kuvaa toiminnan olevan enemmän sattumanvaraista, kuin ammattimaista. Tasoa kutsutaan nimellä ”initial”, eli alkuperäinen. Tällä tasolla muutama prosessi on kuvattu, mutta suurin osa avainprosesseista puuttuu. /10/

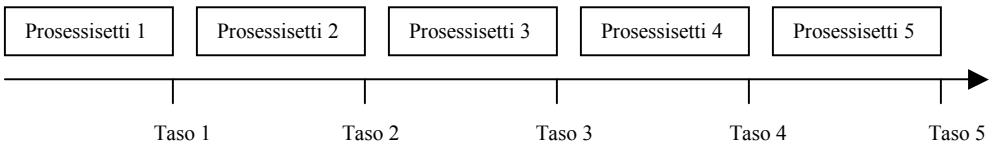
Toinen taso kuvaa ohjelmistonkehitystä toistettavina projekteina. Samantyyppiset hyvin onnistuneet projektit pystytään prosessien avulla kerta toisensa jälkeen kopioimaan. Tämä on mahdollista, koska projektinhallintaprosessit on perustettu ja nämä ohjaavat toimintaa. /10/

Kolmas taso esittää varsinaisen ohjelmistonkehityksen ja sen ohjauksen standardisoituina ja integroituina prosesseina. Tässä vaiheessa koko organisaation on tuettava ohjelmistonkehitystä, eli esimerkiksi laatuorganisaatio on oltava olemassa. /10/

Neljännen tason vaatimukset ovat sellaisia, että sekä ohjelmistoprosessin että tuotteen laatua mitataan ja käytetään edelleen prosessien kehittämiseen. Sekä tuote, että prosessit tunnetaan hyvin. /10/

Viides taso tarkoittaa sitä, että prosesseja parannetaan jatkuvasti niistä saadun tiedon perusteella. Uusia ideoita ja teknologioita pilotoidaan. /10/

CMM kuvaa ohjelmistotuotannon laatua yksiulotteisesti, vain prosessien olemassaolon perusteella. Uudemmat laadunarviointimallit arvioivat erikseen prosessien olemassaoloa ja niiden laatua. Kuva 2.4.1 esittää CMM:n mittausperiaatteen



Kuva 2.4.1 CMM:n toiminta

2.4.2 SPICE

SPICE-hanke aloitettiin ISO:n toimesta vuonna 1992. Projekti aloitettiin, koska tarvittiin yhteinen standardi ohjelmistotuotannon prosessien arviointiin ja parantamiseen. Lopputuloksena saatu arviointimalli on standardi ISO 15504. Tällä hetkellä on käytössä versio 3. Uuden version julkistus Suomessa on 8.4.2003.

SPICE:n lähtökohta on se, että prosessien arviointi on edellytys niiden parantamiselle. Ajatuksena ei ole kilpailla siitä, kenen prosessit toimivat parhaiten, vaan verrataan prosesseja ja ohjelmistojen laatua toivottuun tasoon. Lopputuloksena saadaan paitsi menetelmien arviointi, myös keinot, joilla prosesseja voidaan parantaa.

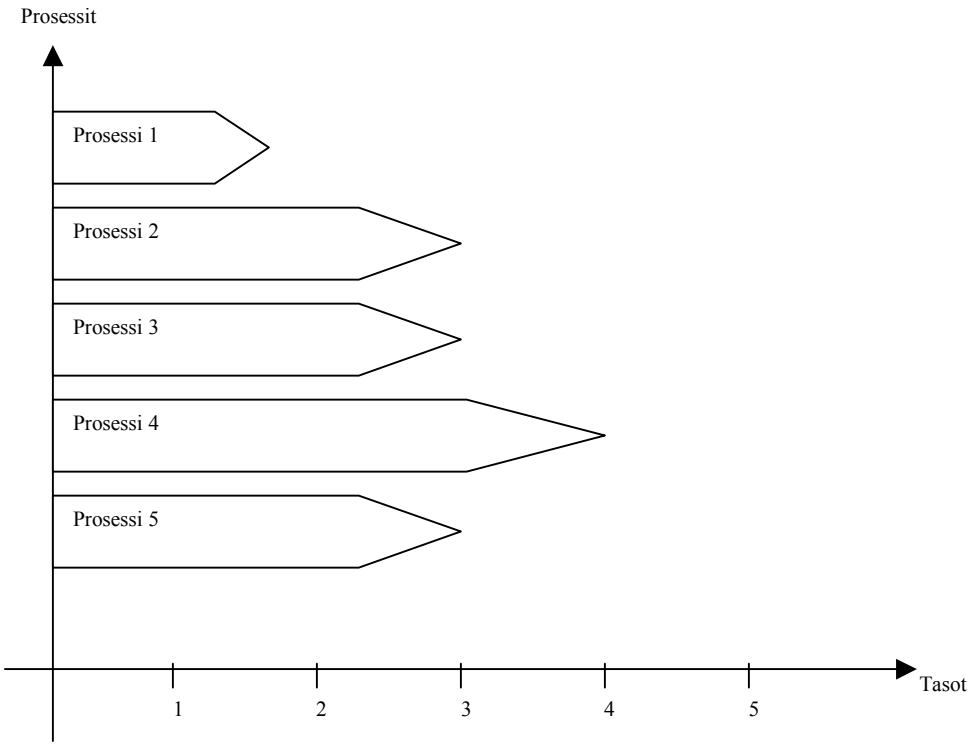
SPICE:n arviointi on prosessikohtaista. Toisin kuin esimerkiksi CMM, SPICE arvioi kunkin prosessin erikseen, jolloin saadaan kyseiselle prosessille laatua kuvaava arvo. SPICE kuvaa laatua kaksiulotteisesti, eli pelkän prosessin olemassaolon lisäksi prosessista arvioidaan myös sen laatu. Prosessin olemassaoloa kuvaa taso yksi. /11/

SPICE:n tasolla kaksi prosessi on hallittu. Prosessi toimii ennustettavasti annettujen aikataulujen ja resurssien rajoissa sekä tuottaa määritellyn laatutason mukaiset työn tulokset. /11/

Tasolla kolme oleva prosessi on vakiintunut. Prosessi suoritetaan organisaation tasolla määritellyn prosessin mukaisesti. Prosessi on ohjattu tilanteeseen sovitettu, eli takaisinkytketty. /11/

SPICE:n tasolla neljä prosessi on ennustettava. Prosessin toiminta pystytään ennakoimaan astettujen rajojen sisällä. Mittarit johdetaan liiketoiminnan tavoitteista. Tällaista prosessia kutsutaan myös eteenpäin kytketyksi. /11/

Tason viisi prosessi on itseohjautuva. Prosessi kykenee parantamaan ja optimoimaan toimintaansa saavuttaakseen nykyiset ja tulevat liiketoiminnan tarpeet. /11/

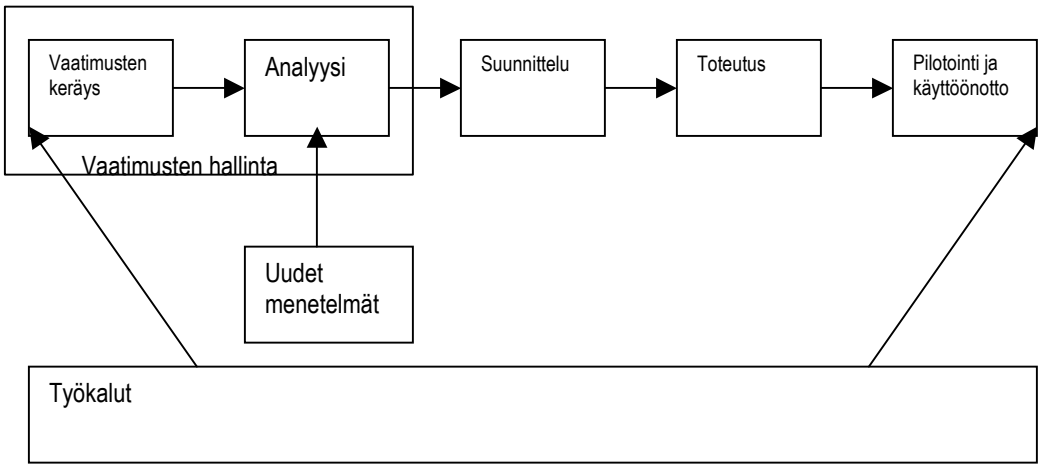


Kuva 2.4.2 SPICE:n kaksiulotteinen mittaustapa

3 OLIOPOHJAINEN SYSTEEMITYÖ

Ohjelmistonkehitysprosessia hyödynnetään itse ohjelmiston kehityksessä systeemityössä. Prosessin aliprosessit kuvaavat systeemityön vaiheet ja sen mitä niissä tehdään ja minkälaisia dokumentteja on tuloksena missäkin vaiheessa. Prosessikuvaukset ovat yksityiskohtaisia työohjeita, joista kussakin projektissa otetaan tarpeelliset osat käyttöön.

Kuvassa 3.1 on esitetty prosessimallin esittämä ohjelmistojen kehitystyön vaiheistus. Kunkin vaiheen työn tulokset siirtyvät seuraavan vaiheen syötteiksi. Itse vaiheiden kuvaukset toteutusta ja pilotointia lukuunottamatta on esitetty seuraavissa luvuissa. Toteutus ja pilotointi on jätetty pois siksi, että ne on rajattu tämän työn ulkopuolelle. Toteutus on koodin kirjoittamista ja pilotoinnilla varmistetaan se, että järjestelma on sopiva lopulliseen tarkoitukseensa.



Kuva 3.1 Ohjelmistokehityksen vaiheistus

3.1 Vaatimusten hallinta

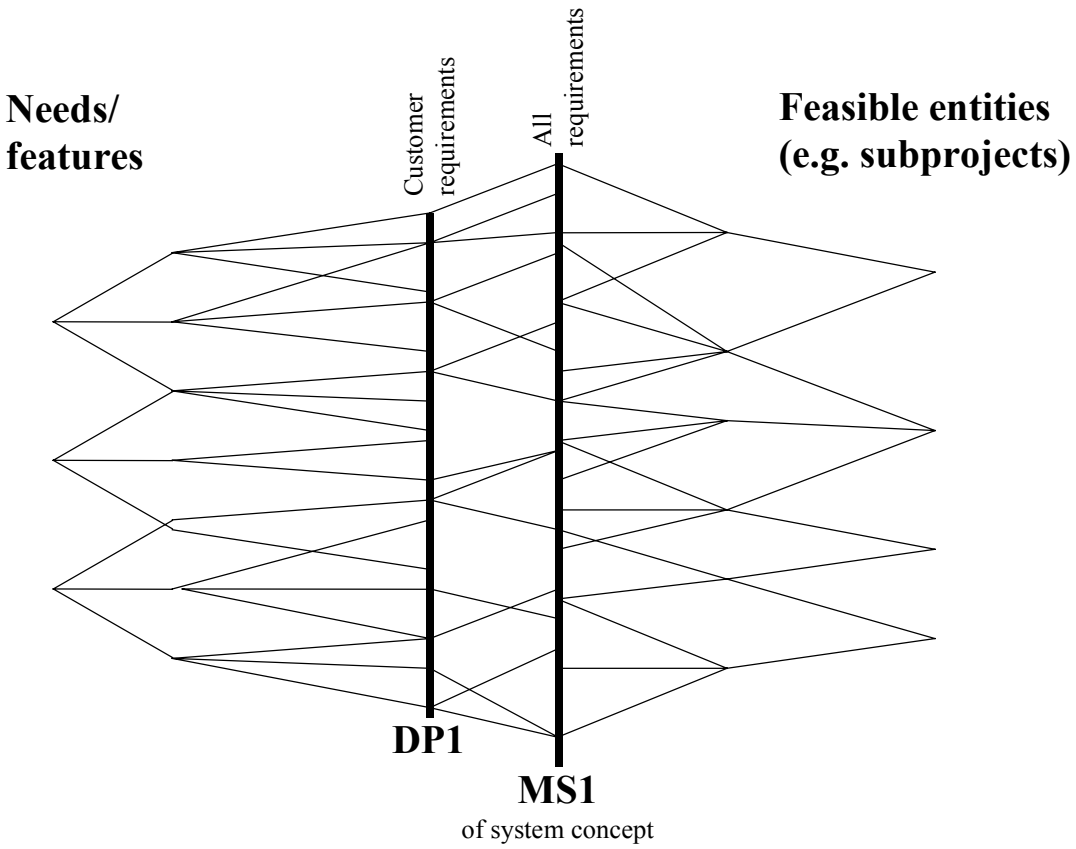
Jaettiinpa ohjelmistotyö vaiheisiin miten tahansa, on päätavoite tehdä asiakasvaatimuksista asiakasvaatimukset täyttävä ohjelmisto. Tämän varmistamiseksi tehtäviä toimenpiteitä kutsutaan nimellä vaatimusten hallinta. Vaatimusten hallinnan keskeisiä tehtäviä on varmistaa, että lopputuotteessa on kaikki asiakkaan haluamat ominaisuudet ja vain ne. /2/

Vaatimusten hallinta on tärkein vaihe ohjelmiston kehityksessä. Jos asiakkaan toiveita ei tunneta, ei niitä ole mahdollista toteuttaa. Oikeiden vaatimusten kerääminen ja niiden tarkastaminen on syytä ohjeistaa tarkasti, että hyvät menetelmät tulevat käyttöön koko organisaatiossa. Lisäksi asiakkaan on hyvä tietää karkealla tasolla, mitä minkäkin ominaisuuden toteuttaminen maksaa, esimerkiksi ominaisuuden X toteuttaminen vie aikaa nelinkertaisesti ominaisuuteen Y verrattuna. Näin saadaan analyysin avulla asiakkaan tietoon ne ominaisuudet, jotka todella on syytä toteuttaa ja ne jotka voidaan siirtää seuraavaan versioon tai jättää kokonaan pois.

Asiakkaan aluksi esittämien vaatimusten ja toiveiden lisäksi on otettava huomioon vaatimuksiin projektin aikana tulevat muutokset. Toivottavaa on, että vaatimukset pysyisivät mahdollisimman stabiileina muutosten aiheuttamien suurien kustannusten takia. Muutoksilta ei ainakaan suurempien projektien kyseessä ollen kuitenkaan tavallisesti vältytä. Tämä johtuu siitä, että asiakaskaan ei yleensä tiedä tarkalleen, mitä haluaa, eikä ohjelmiston suunnittelija tai järjestelmän analysoija osaa etsiä oikeita toiveita. Tästä syystä asiakkaalle on tehtävä selväksi, että vaatimukset jäädytetään siinä vaiheessa, kun sopimus on allekirjoitettu. Jäädytettyyn vaatimuslistaan tehdään muutoksia vain etukäteen sovitulla menetelmällä. Muuttunut työmäärä myös vaikuttaa ohjelmistoversion lopullisiin kustannuksiin. Versiosta poisjääneitä vaatimuksia voidaan sitten käyttää ohjelmiston seuraavassa versiossa. Toisaalta asiakkaalle tulee kertoa

muutosten aiheuttamat kustannukset, mikäli hän haluaa uusia ominaisuuksia jo suunnittelu- tai implementointivaiheessa olevaan projektiin. Järjestelmällinen vaatimusten kartoittaminen on ainoa oikea keino yrittää välttää muutoksia jo suunnitteilla olevaan ohjelmistoon. Kokemus tuo mukanaan tietoa oikeasta tavasta kartoittaa asiakkaan toiveet.

Kuva 3.1.1 esittää vaatimusten jakautumisen alivaatimuksiin, joista kyseinen vaatimus koostuu ja niiden kasaamisen analyysissä täysin uudeksi vaatimusten joukoksi. Vasemmalla näkyy asiakasnäkökulma tehtävään ohjelmistoon, kun taas oikealla on toteutettavuusnäkökulma, joka saadaan ohjelmiston kehittävältä yksiköltä. Kun kaikki vaatimukset on saatu kerättyä ja vaatimusluetteloon on lisätty tekniset vaatimukset, on vaatimusjoukko suurimmillaan. Kun varsinaista analyysiä aletaan tehdä, vaatimukset kasautuvat uudelleen joukoiksi, jotka esimerkiksi tietyt oliot tai komponentit toteuttavat. Kuva on Sonera Service Softwaren Requirement capture prosessista.



Kuva 3.1.1 Vaatimusten jakautuminen asiakas- ja toteutettavuusnäkökulmaan

3.1.1 Asiakasvaatimusten kartoitus

Asiakkaan lisäksi vaatimuksia ohjelmistolle voivat esittää myös muut sidosryhmät. Lisäksi on käytössä useita eri menetelmiä vaatimusten keräämiseksi. Seuraavassa on esitetty menetelmiä ja sidosryhmiä, joilla ominaisuuksia tai niiden poistamista voidaan kartoittaa. Sonera System Softwaren Requirement Capture-prosessin liitteenä on taulukko Requirement eliciting, joka kuvaa menetelmien ja sidosryhmien välistä suhdetta vaatimusten määrittelyvaiheessa. Lisäksi sidosryhmien kuulemisen tärkeyden astetta on

yritetty selvittää ja tämä ominaisuus on kuvattu kirjaimilla kussakin matriisin elementissä.

Sidosryhmiä Soneran käyttämään taulukkoon on merkitty 14. Niistä tärkeimpiä ovat asiakas, loppukäyttäjä, tuki sekä myynti ja markkinointi. Asiakas ja loppukäyttäjä tarkoittavat yleensä samaa vain yksityishenkilöille tarkoitettujen ohjelmistotuotteiden yhteydessä. Asiakas on se, joka tilaa tuotteen ja maksaa sen. Loppukäyttäjä voi olla esim. asiakkaan työntekijä tai asiakkaan asiakas. Tuella tarkoitetaan ohjelmistotukea, jolla on hyvä käsitys ohjelmiston toiminnasta ja jatkokehitystarpeista. Myynti ja markkinointi on ohjelmiston toimittajan asiakkaiden kanssa yhdessä toimiva elin.

Taulukkoon on menetelmistä merkitty:

- Demo, jolla tarkoitetaan esimerkiksi jollain ohjelmistolla tehtyjä kuvia käyttöliittymästä. Demo voi olla esimerkiksi pelkkä kalvoesitys, ettei asiakas erehdy luulemaan tuotteen olevan lähes valmis pelkkien käyttöliittymien perusteella.

- *Skenaario*, joka on esitys ohjelman käytöstä ja sen saamista syötteistä ja tuottamista tuloksista.

- *Aivoriihi*, jolla tarkoitetaan jotain määrämuotoista ideoiden keräämismenetelmää.

- *Haastattelu*, jolla yritetään kerätä alkuperäisiä ominaisuuksia tarkentavia yksityiskohtia.

- Työryhmillä haetaan yhteisesti näkemyksiä vaatimuksista eri näkökulmista.

- Muistilistoilla yritetään selvittää kaikkien näkökulmien mukanaoloa. Requirement eliciting-taulukko on itse esimerkki muistilistasta.

- Neuvottelun tarkoituksena on sopia asioista. Jos esimerkiksi tuotekehityksen mielestä jonkin ominaisuuden toteuttaminen vie paljon aikaa, on asiakkaan kanssa syytä sopia tämän ominaisuuden toteuttamismahdollisuuksista.

- Asiantuntijoiden käytöllä haetaan erityisosaamiseen perustuvien ominaisuuksien löytämistä, tai rajoitteita joidenkin ominaisuuksien käytölle. Esimerkkinä erityisosaamisesta voidaan mainita lakiasiain osasto, joka voi esittää lainsäädäntöön perustuvia rajoituksia ohjelmistolle.

- Tietokannalla tarkoitetaan sekä aikaisempien ohjelmistojen, että kyseessä olevan ohjelmiston aikaisempien versioiden toteutuksen aikana syntyneitä ideoita, tai esimerkiksi kilpailijoiden tuotteiden ominaisuuksien ottamista mukaan.

- Suunnattu ideoiden luonti tarkoittaa tiettyyn ominaisuuteen liittyvää pohdiskelua tietyssä tilanteessa.

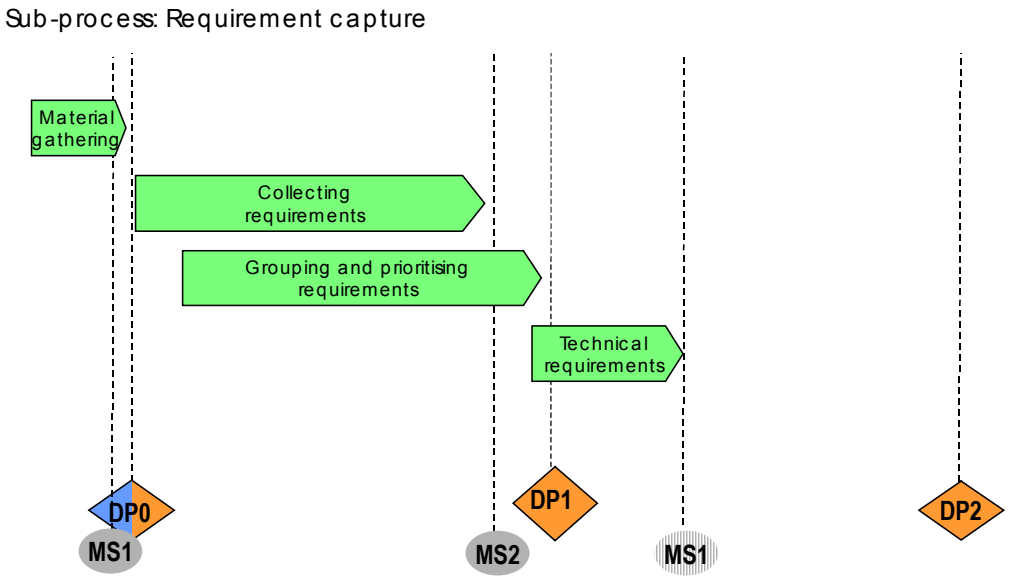
- Vapaa ideoiden luonti on irrotettu ajasta ja paikasta. Esimerkkinä voi olla ideataulu seinällä, johon jokainen ohikulkija voi vapaasti kirjoittaa ajatuksia ominaisuuksista.

Taulukossa on kuvattu sidosryhmien osallistumisen tärkeyttä kuhunkin menetelmään kirjaimilla A, B ja C. A:lla tarkoitetaan, että osallistuminen on välttämätöntä, B:llä osallistumisen olevan suotavaa ja C:llä prosessin suorituksen aikana kultakin sidosryhmältä tulevia ajatuksia.

Taulukkoa voi pitää ohjeena, kun valitaan menetelmiä ja sidosryhmiä ohjelmiston ominaisuuksia kartoitettaessa. Kokemus tuo oman lisänsä menetelmien valintaan, koska eri tyyppiset ohjelmat vaativat myös erilaisia menetelmiä.

Jäljitettävyyden varmistamiseksi on vaatimukset merkittävä yksikäsitteisellä tunnisteella. Tunniste voi sisältää tietoa myös siitä, mistä vaatimus on tullut, milloin ja millä menetelmällä. Jäljitettävyys on tärkeää vertailtaessa suunniteltua ja toteutettua järjestelmää vaatimuksiin. Näin voidaan todeta, että kaikki vaatimukset on toteutettu.

Vaatimusten keräilyn lisäksi vaatimukset tulee myös priorisoida. Tämä on ennen kaikkea asiakkaan tehtävä. Vaatimukset on saatava järjestykseen niin, että asiakkaan toiveisiin voidaan paneutua tärkeysjärjestyksessä. Tämä on tärkeää siitä syystä, että kaikkia asiakkaan vaatimuksia ei aina pystytä toteuttamaan annetussa aikataulussa. Seuraavassa vaiheessa vaatimukset ryhmitellään toteutuksen näkökulmasta. Nämä aktiviteetit on esitetty kaaviona kuvassa 3.1.2.



Kuva 3.1.2 Soneran Requirement capture-prosessi prosessikaaviona.

3.1.2 Vaatimusten analysointi

Vaatimusten analysointivaihe poikkeaa oliopohjaisessa ratkaisussa perinteisestä ohjelmointimallista. Oliomallissa tieto ja sen käyttö on yhdistetty samaan olioon. Vaatimusten analysointi onkin olioiden etsimistä, järjestelyä sekä kommunikoinnin, toiminnan ja olion sisällön määrittelyä. /6/

Analyysivaiheessa ohjelmistosta tehdään kaksi mallia: käyttötapaus- ja analyysi-mallit. Tässä vaiheessa mallit eivät ota vielä mitään kantaa toteutusympäristöön, eli ohjelmointikieleen, tietokantaohjelmistoihin tai käyttöjärjestelmään. Tästä on se etu, että ohjelmiston ominaisuuksista voidaan keskustella käyttäjien kanssa ilman toteutukseen liittyviä teknisiä yksityiskohtia. Lisäksi tällä menetelmällä varmistetaan se, että ohjelmiston arkkitehtuuri perustuu ratkaistavaan ongelmaan, eikä toteutuksen aikaisiin ratkaisuihin. Mallit on kuitenkin tehtävä sellaisiksi, että ne ovat myöhemmin toteutettavissa. /6/

Käyttötapausmalli rakennetaan käyttötapauskuvauksista. Käyttötapauskuvaukset ovat myös UML-kuvauskielen (UML = Unified Modeling Language) määrityksissä. Niillä kuvataan jokainen käyttötapaus jokaisen käyttäjän kannalta. Käyttötapausmalli on ohjelmistosuunnittelijan kuvaus tehtävästä järjestelmästä, jolloin päästään vertailemaan näkemyksen samanlaisuutta käyttäjien kanssa.

Analyysimallilla muodostetaan luokat, niiden instanssit ja liitynnät. Tarkoituksena on tehdä hyvä alusta, ns. arkkitehtuuri, suunnittelun pohjaksi, jolla pystytään toteuttamaan käyttötapausmallin esittämät toiminnot. /6/ UML:n luokkakaavio on hyvä esitystapa. Tätä tapaa voidaan käyttää myös kirjallisuudessa kuvaamaan karkealla tasolla erilaisia menetelmiä. /7/

Analyysimalli on looginen kuvaus järjestelmästä ja sen käyttö mahdollistaa ohjelmiston ylläpidettävyyden. Loogisuus tarkoittaa sitä, että mitään toteutusympäristöön liittyviä asioita ei vielä ole otettu huomioon. /6/ Ylläpidettävyyttä taas helpottaa, kun tiedetään muutosten ja lisäysten vaikutukset ohjelmiston eri moduleihin.

3.2 Ohjelmiston suunnittelu

Ohjelmiston suunnittelu on arkkitehtuurin täydentämistä yksityiskohtaiseksi kuvaukseksi ohjelmiston toiminnasta. Tässä vaiheessa otetaan huomioon myös ympäristö, johon ohjelmisto toteutetaan. Suunnittelu on tarpeen myös siksi, että voidaan todeta arkkitehtuurikuvauksen toimivuus. Pelkkä toteutus arkkitehtuurin suunnittelun jälkeen ei siis riitä. /6/ Suunnitteluprosessiin (-vaiheeseen) tulee koottu tieto analyysi- ja käyttötapausmalleista.

Suunnitteluvaiheen keskeisiä laatuvaatimuksia ovat ohjelmien ymmärrettävyys ja muunneltavuus. Tämä johtuu elinkaarikustannusten kohdistumisesta valtaosin

testaukseen, virheiden jäljitykseen ja ylläpitotyöhön. Tästä syystä suunnittelun yleisperiaatteista voidaan nostaa esille seuraavat. /2/

- *Yksinkertaisuus ja suoraviivaisuus*, mihin päästään pyrkimällä löytämään yksikertaisin ja suoraviivaisin ratkaisu, joka on riittävän hyvä, ei siis välttämättä hienoin.
- *Osittaminen ja lokaalisuus*. Osittamisella tarkoitetaan monimutkaisen ongelman hajoittamista pienempiin helpommin hallittaviin kokonaisuuksiin, moduuleihin. Lokaalisuudella tarkoitetaan tässä yhteydessä osittamisen suunnittelua siten, että suunnittelupäätökset kapseloidaan mahdollisimman hyvin moduulien sisään niin, että muutosten tekeminen lokalisoituu.
- *Abstraktioiden hyödyntäminen*. Abstraktiolla tarkoitetaan mallia, joka kuvaa esittämästään asiasta oleellisen. Tarpeettomat yksityskohdat voidaan näin kapseloida moduulin sisälle.
- *Yhdenmukainen toteutusfilosofia*. Tällä tarkoitetaan uusien moduulien mahdollisimman yhdenmukaista toteutusta ja liittämismekanismia ohjelmistoon. Tähän tarjoaa pohjan jo hyvin tehty arkkitehtuurisuunnittelu.

Moduulin rajapinta on sopimus moduulin käyttäjän ja sen toteuttajan välillä moduulin tuottamista palveluista. Rajapintojen hyvä suunnittelu parantaa ylläpidettävyyttä ja uudelleenkäytettävyyttä, jos rajapintojen väliset kytkennät jäävät vähäisiksi. Varautuminen muutoksiin edellyttää rajapintojen mahdollisimman pientä muuttumista ohjelmiston elinkaaren aikana /2/.

Suunnitteluvaiheessa ohjelmistosta tehdään kaksi mallia, eli *suunnittelu*- ja *toteutusmallit*. Suunnittelumallissa arkkitehtuurikuvaukseen lisätään toteutusympäristöön liittyvät asiat (ohjelmointityökalun antamat mahdollisuudet, kertyneet luokkakirjastot sekä käyttöjärjestelmän asettamat rajoitukset ja toteutustavat). /6/ Tässä vaiheessa lisätään analyysimalliin esimerkiksi se, mistä

kukin arkkitehtuurin esittämä luokka periytyy, ja mahdolliset toteutusympäristön vaatimat luokat.

Toteutusmalli esittää luokkien sisäiset toiminnot, niiden sisältämän tiedon ja rajapinnat yksityiskohtaisesti. Toteustusmallista ohjelmointi on suoraviivaista, mutta sen analysointi on huomattavasti helpompaa, kuin lähdekoodin. Tässä vaiheessa ohjelmiston yksityiskohdat ovat jo täysin selvillä.

3.3 Uusia menetelmiä

Viime aikoina uusia menetelmiä on tutkittu laajasti. Useissa yliopistoissa on havaittu uusien ohjelmistotuotteiden nopean markkinoille saannin merkitys. Ongelmaa onkin tutkittu sekä liiketaloudellisissa, että teknisissä tiedekunnissa.

3.3.1 Rapid Application Development (RAD)

Artikkelissaan, Risks of Rapid Application Development, Agarval, Prasad, Tanniru ja Lynch kertovat Rapid Application Development-menetelmän (RAD) riskeistä. RAD ei ole yksikäsitteisesti määritelty menetelmä. Se on enemmänkin iteroiva vaihejakomalli, johon on liitetty ohjelmistotyökaluja, jotka mahdollistavat nopean oliokehityksen, graafisen käyttöliittymän ja koodin uudelleenkäytön /9/.

Agarval & al ovat sitä mieltä, että RAD:issa on käytetty liian vähän aikaa ohjelmiston koko elinkaaren hallintaan. Heidän mielestään RAD:issa ei kiinnitetä tarpeeksi huomiota projektin päättymisen tai seuraavan tuotteen julkaisun jälkeiseen aikaan eikä ohjelmiston ylläpidettävyyteen /9/.

Ongelmana RAD:issa nähdään kehittäjien haluttomuus noudattaa mitään mallia analyysivaiheen jälkeen. Kehittäjät haluavat aloittaa koodaamisen heti, kun ratkaistava ongelma on selvillä. Suunnitteluun ei siis haluttaisi käyttää aikaa /9/.

RAD näyttäisi sopivan isompiin, mutta nopeatempoisiin kehityshankkeisiin. Pieniin projekteihin se ei tuo ratkaisua, koska siinä odotetaan eri henkilöiltä eri rooleja hankkeen aikana ja se lähinnä keskittyy useamman henkilön työn rinnakkaisuuden hallintaan sekä koodin uudelleenkäyttöön. /9/

3.3.2 Vaatimusten uudelleenkäyttö analyysivaiheessa

Viimeisimmät tutkimukset ovat lähteneet etsimään keinoja analyysivaiheen tulosten uudelleenkäytön edistämiseksi. Koska analyysissä selvitetään kohdealueen ongelmia ja niiden ratkaisuja, on analyysien uudelleenkäytön tehostamisessa löydettävä yhteisiä tekijöitä jo liiketoiminnan alueella. Mitä yleisemmäksi kohdealue saadaan määriteltyä, sitä yleisemmin voidaan analyyseja käyttää hyväksi. Yleisyydellä voidaan tässä tapauksessa tarkoittaa yhtä laajaa aluetta, jolloin uudelleenkäytetään vaatimuksia esim. koko pankkisektorille tehtävissä sovelluksissa. Toinen mahdollinen tapa on se, että analyysissä voidaan käyttää hyväksi pieniä yksittäisiä yleisesti käytössä olevia vaatimuksia.

Sugumaran, Tanniru ja Storey ovat artikkelissaan Supporting Reuse In Systems Analysis etsineet keinoja toimialatietokantojen luomiseen. Tutkimuksessa rakennettiin järjestelmä, johon kerättiin pankkitoimialan vaatimuksia. Tutkimuksessa pystyttiin osoittamaan, että ainakin pankkitoimialalla järjestelmä toimii. Osa kerätyistä vaatimuksista pystyttiin hyödyntämään seuraavassa projektissa toteutusta myöten. /8/

3.4 Työkaluista

Tässä luvussa esitellyt työkalut ovat lähinnä arkkitehtuurin ja toiminnallisuuden suunnitteluun tarkoitettuja. Projektiin liittyvät dokumentit kirjoitettiin Microsoft Word-ohjelmistolla.

3.4.1 Unified Modeling Language (UML)

Unified Modeling Language (jatkossa UML) on Object Management Group:in (OMG) marraskuussa 1997 hyväksymä standardi oliosuunnittelussa käytettäväksi kuvauskieleksi. UML:a on käytetty systeemityössä useilla eri tietotekniikan osa-alueilla. UML:a on käytetty arkkitehtuurin ja toiminnallisuuden suunnitteluun useilla eri kielillä toteutetuissa ohjelmistoprojekteissa. /12/

UML esittää tavan kuvata ohjelmisto käyttötapauskaavioista aina yksityiskohtaisiin toimintakaavioihin. Se on siis käyttökelpoinen käyttötapausmallista aina toteutusmalliin asti. UML:n kehitys alkoi 1980-luvun loppupuolella. Grady Booch alkoi kehittää omaa Boochin menetelmäänsä. Samoihin aikoihin alkoi tulla käyttöön useita eri oliopohjaisen ohjelmistokehityksen metodeja, esimerkiksi jo mainittu Boochin menetelmä sekä Rumbaughin OMT ja Jacobsonin OOSE. Kullakin menetelmällä oli omat vahvuutensa ja heikkoutensa. /12/

UML:n kehitys nopeutui vuonna 1994, kun James Rumbaugh ryhtyi Boochin työtoveriksi Rational Corporation:in menetelmälaboratorioon. Alustava versio 0,8 Unified Method julkaistiin lokakuussa 1995. Näihin aikoihin ryhmään liittyi myös Ivar Jacobson. Kesäkuussa 1996 julkaistiin UML:n versio 0,9. Vuoden 1996 aikana kerättiin kokemuksia käyttäjiltä ja versio 1,0 esiteltiin OMG:lle tammikuussa 1997. /12/

UML:a tukevia mallinnustyökaluja on tarjolla useita. Maailmanlaajuisesti tunnetuin lienee Rational Corporation:in Rose. Suomessa on lisäksi käytössä Prosa, jonka on kehittänyt oululainen Intellisoft OY.

3.4.2 Rational Rose

Rose on Rational:in tuottama ohjelmistojen mallintamiseen tarkoitettu työkalu. Se on tarkoitettu UML-pohjaiseen kehitykseen. Rose tukee vaiheittaista ohjelmistonkehitystä siten, että se muokkaa projekteja eri vaiheessa olevista suunnitelmista ja pitää ne yhtenäisenä pakettina. /13/

Rose tarjoaa liitynnän myös MFC-ympäristöön. Siinä on valmiiksi esitettynä MFC:n luokkakirjastot, joten kehitys nopeutuu myös Windows-ympäristössä Microsoft:in työkaluilla. /13/

3.4.3 Microsoft Foundation Class (MFC)

MFC on Microsoft:in tarjoama työkalu C++-kehitykseen. Se on luokkakirjasto, joka on tehty Microsoft Windows käyttöjärjestelmän ohjelmointirajapinnaksi. /14/

MFC tekee ns. sovellusrungon (kokoelma oliokeskeisiä ohjelmistokomponentteja), jota käytetään hyväksi ohjelmoitaessa. Sovellusrunko poikkeaa pelkästä luokkakirjastosta siinä, että se määrittelee myös ohjelman rakenteen. Tämä sovellusrunko toteuttaa ns. näkymä-dokumentti-arkkitehtuurin. Tämä arkkitehtuuri perustuu erillisiin näkymä- ja dokumenttiluokkiin. Näin kaikki MFC:n avulla tehdyt sovellukset rakentuvat samalla tavalla, jolloin sovellusten ylläpito helpottuu. /14/

MFC tukee useita eri Windows-tekniikoita. Näistä voidaan mainita esimerkkeinä OLE, ODBC, MDI ja DLL. Visual C++:n kääntäjä tuottaa 32-bittistä koodia ja tukee siten sekä Windows 95/98 että NT/2000 ympäristöjä.

3.4.4 Reliable Transaction Engine (RTE)

RTE on Soneran ja sen asiakkaiden käytössä oleva neljännen sukupolven syöteohjattu ohjelmointikieli. Kieli on havaittu käyttökelpoiseksi kehitystyökaluksi myös tietoliikenneympäristössä, vaikka se on alunperin suunniteltu ja toteutettu sähköisessä liiketietojen vaihdossa (EDI) käytettävien muuntimien tekoon. Muuntimet lukevat annetun dokumentin ja sijoittavat siinä olevat tiedot määriteltyyn viestiin. /15/

RTE sisältää kaikki normaalissa ohjelmoinnissa käytettävät lauserakenteet. Erityisominaisuutena siinä on dokumentin lukemisessa vaadittavat ominaisuudet ja vertailu EDI-sanomien syntaksiin. Vertailu otetaan huomioon, kun koodin alkuun kirjoitetaan message-lause, jossa ilmoitetaan viestikuvauksen sijainti kääntäjälle.

EDI-sanomien lukua varten on RTE-kieleen lisätty segment- ja line-lauseet, joilla ilmoitetaan viestin segmentti ja dokumentin rivi. Dokumentin rivi alkaa tunnisteella, joka ilmoitetaan line-lauseen parametrina. Näin muunnin tietää, miltä dokumentin riviltä segmenttiin tuleva tieto luetaan. Kukin tieto luetaan tunnisteeseen verrattuna suhteellisen rivin numeerisesti ilmoitetusta paikasta numeerisesti ilmoitetun mittaisena.

4 CASE SONERA TRADEXPRESS MAPPER

Mapper-ohjelmisto on toteutettu 1.10. 1998 – 17.8. 2001 kolmena erillisenä projektina, joissa kussakin on toteutettu erilaisia ominaisuuksia kulloisenkin tarpeen mukaan. Koska ohjelmisto on kokonaisuutena liian iso käsiteltäväksi yhdessä diplomityössä, tässä työssä käsitellään viimeisimmän projektin yhtä osaa, joka on esitetty kokonaisarkkitehtuuriin sidottuna. Seuraavassa lyhyt kuvaus koko ohjelmiston toiminnasta.

Ohjelmiston tarkoituksena on tehdä muuntimia (translators), joilla muunnetaan tekstimuotoinen data EDI- (Electronic Data Interchange) tai XML-sanomaksi (eXtended Markup Language). Menetelmänä on yhdistää (mapping) rakenteista tietoa kahdesta eri tiedostosta toisiinsa, jonka jälkeen muunnin saadaan generoitua RTE-kielelle. RTE-kielinen koodi käännetään aluksi c-kielelle, minkä jälkeen se käännetään suoritettavaksi tiedostoksi. Muunnin käynnistetään siten, että annetaan parametrinä luettavan tekstitiedoston nimi. Tämän jälkeen muunnin lukee tekstin rivi kerrallaan ja rivin alussa olevan tunnisteen avulla päättää, mitä kyseisen rivin sisältämälle datalle tehdään.

Tiedostot voivat olla joko eri standardien mukaisia EDI-viestikuvauksia (XML:n kyseessä ollen DTD-kuvauksia) tai tavallisia tekstitiedostoja, jotka ovat esimerkkejä siirrettävistä tiedostoista. Viesti- tai DTD-kuvauksia voi kussakin muuntimessa olla vain yksi johtuen RTE-kielen rakenteesta.

Mapper-ohjelmisto tarjoaa graafisen käyttöliittymän muuntimien tekoon Windows-ympäristössä. Ennen viimeistä projektia yksi tunnettu ongelma oli, että ohjelmisto vaati valmiin mallidokumentin, jota käytettiin dokumentin sisältämän tiedon rakenteen kuvaamisessa. Dokumentin sisältämä tieto paloiteltiin puurakenteeksi ja vaatimuksena oli mm. se, että dokumenttia ei enää tarvittaisi, vaan puumainen tietorakenne voitaisiin tehdä graafisesti.

4.1 Vaatimusten keräys

Mapper-ohjelmiston vaatimuksia etsittiin lähinnä haastatteluilla ja palavereilla. Sidosryhminä käytettiin tuotteistusorganisaatiota, joka oli myös ohjelmiston tilannut asiakas, sekä tuotetukea, joka oli rajapinta loppuasiakkaisiin. Lisäksi muutama käyttäjä sai esittää toiveita ja ehdotuksia.

Vaatimukset kirjattiin vaatimusluetteloon (Requirement catalogue). Vaatimukset lajiteltiin osa-alueittain niin, että toiminnallisten vaatimusten tunnus oli 1.x.x, käytettävyyteen liittyvien vaatimusten tunnus oli 2.x.x, suorituskykyyn liittyvien vaatimusten tunnus oli 3.x.x jne. Lisäksi käytettiin tekniikkaa, jossa perusvaatimus jaettiin alivaatimuksiin. Näin saatiin vaatimusjoukkoja, joita pystyttiin käsittelemään yhtenäisesti. Lisäksi hierarkinen rakenne tarkensi vaatimuksia ja teki niistä entistä yksiselitteisempiä.

Vaatimusten keräysten yhteydessä on tärkeä tieto se, mistä vaatimus tuli. Tästä syystä vaatimusluettelossa oli lähde- ja päivämääräkentät, joihin kirjattiin vaatimuksen lähde ja päivämäärä, jolloin vaatimus ensimmäisen kerran esitettiin. Lähde saattoi olla esimerkiksi yksittäinen henkilö, tai palaveri, jossa asiaa on käsitelty. Vaatimusluettelon täydentämistä jatkettiin ohjelmistokehityksen myöhemmissä vaiheissa.

Esimerkkinä käytettävän vaatimusryhmän tarkoituksena oli, että kehitetään ominaisuus, jonka avulla on mahdollisuus tehdä dokumentin rakenteen kuvaus ilman mallidokumenttia. Mallia ei aina ole saatavilla ja lisäksi ominaisuus lisää käyttömahdollisuuksia työkalulle myös muualla, kuin pelkästään EDI-ympäristössä.

Group	**ID**	**Description**	**Date**	**Source**
Functional requirements	1.3	General format learning feature	10.5.2001	JJ/JT

	1.3.1	Tree and branch editing windows for document structuring	15.5.2001	JJ
	1.3.2	Make a new line branch	15.5.2001	JJ
	1.3.3	Edit an existing line branch	15.5.2001	JJ
	1.3.4	Save the line branch	15.5.2001	JJ
	1.3.5	Open an existing line branch	15.5.2001	JJ
	1.3.6	Add an element to the branch	15.5.2001	JJ
	1.3.7	Delete an element of the branch	16.5.2001	165
	1.3.8	Make a new document tree	16.5.2001	165
	1.3.9	Open an existing document tree	16.5.2001	165
	1.3.10	Add a line branch to the document	15.5.2001	JJ
	1.3.11	Edit element data	25.5.2001	JJ,JT,TSA,SK

Taulukko 4-1 Vaatimusluettelo

4.2 Analyysi

Analyysin yhtenä tarkoituksena on selvittää eri vaatimusten toteuttamisen vaikeus ja sitä kautta auttaa päätöksenteossa. Analyysin jälkeen tiedetään lopulliset vaatimukset ja se, että toteutetaanko kyseinen ohjelmisto. Soneralla lopullinen sopimus toteutettavasta tuotteesta kirjoitetaan analyysin perusteella.

Analyysivaiheessa tehtiin kaksi dokumenttia, joissa esitettiin analysoidut päävaatimukset (analyzed main requirements, AMR) ja toteutussuunnitelma (technical implementation proposal, TIP). Sopimuksen tekoa varten yhtenä analyysin lopputuloksena lisättiin vaatimusluetteloon kunkin vaatimuksen toteutuksen vaikeusaste karkealla kolmiportaisella asteikolla.

4.2.1 Analysoidut päävaatimukset

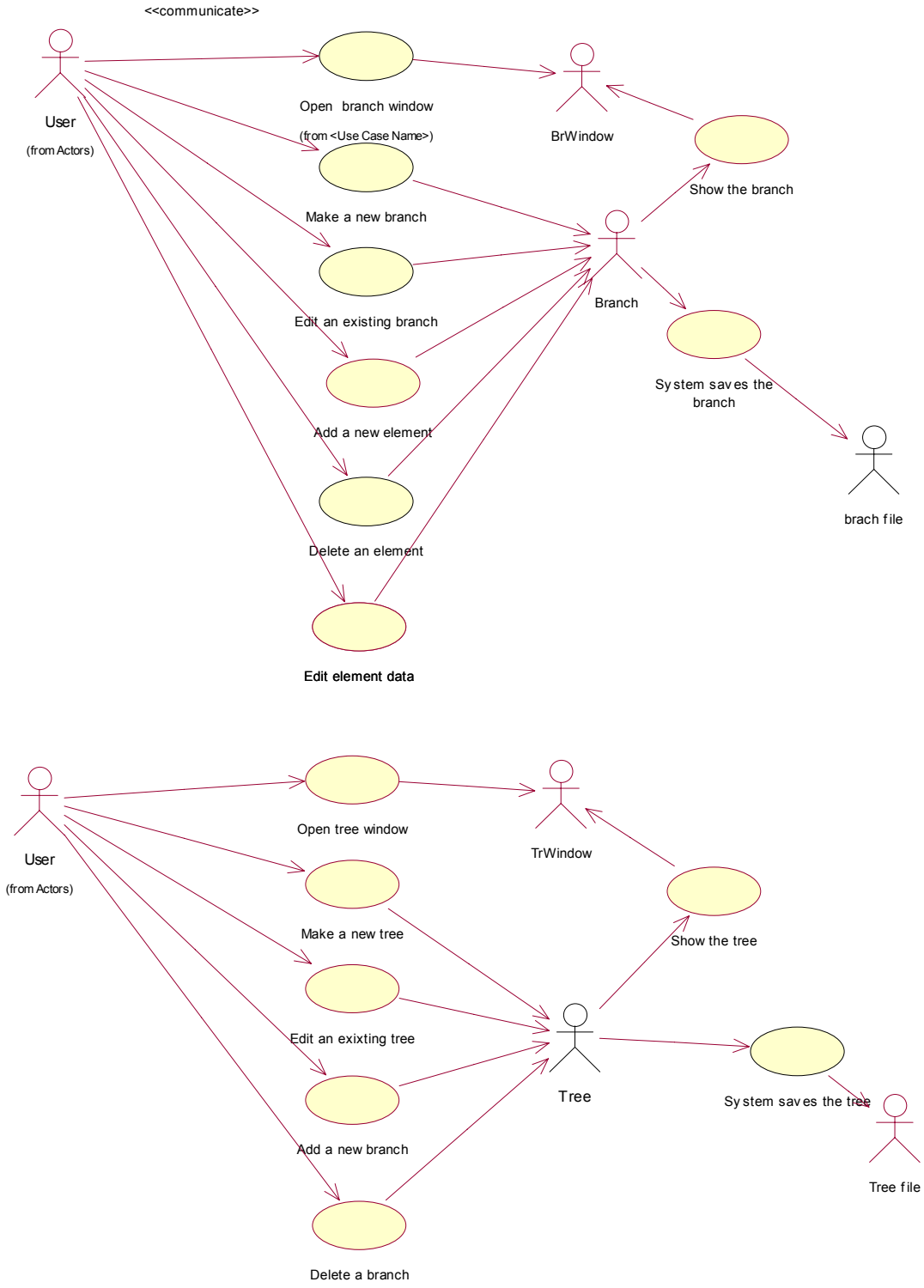
AMR-dokumentin tarkoituksena oli selvittää, miten tehtävä ohjelmisto toimii. Dokumentti kuvasi vaatimukset systeemin toiminnan kannalta. Toiminnalliset vaatimukset analysoitiin käyttäen käyttötapauskaavioita (use case diagram).

Use case on tyypillisesti käyttäjän ja tietokonejärjestelmän välinen tapahtuma. Use case-malli esittää järjestelmän odotettuja toimintoja (use cases), sen ympäristöä (actors) ja näiden välisiä suhteita (use case-kaaviot).

Seuraavia periaatteita noudatettiin dokumentin teossa:

- Vain ne käyttötapaukset kuvattiin, jotka ovat tärkeitä ohjelmiston toiminnan ymmärtämiseksi.
- Käyttötapauskuvaukset ovat lähtökohta suunnitteluryhmän työlle.
- Mahdolliset luokka- ja vuorovaikutuskaaviot lisätään vain järjestelmän selkiyttämiseksi. Ne eivät ole ehdotuksia seuraavaan suunnitteluvaiheeseen.

Käyttötapaukset piirrettiin aluksi yksittäisinä kaavioina. Kukin toiminnallinen vaatimus piirrettiin yksitellen. Toimijoina saattoi olla käyttäjän lisäksi esimerkiksi tiedosto, käyttöliittymän ikkuna tai tietorakenne, johon tieto tallennetaan. Käyttötapausmalli piirrettiin tämän jälkeen yhdistämällä käyttötapaukset toimijoiden mukaan niin, että kukin toimija piirrettiin yhteen kertaan ja siihen liittyvät käyttötapaukset yhdistettiin siihen. Näin saatiin alustava kuva järjestelmän mahdollisesta rakenteesta. Seuraavalla sivulla oleva kuva esittää aikaisemmin esitettyjen vaatimusten mukaisesti tehdyn käyttötapausmallin.



Kuva 4.2.1 Käyttötapausmalli

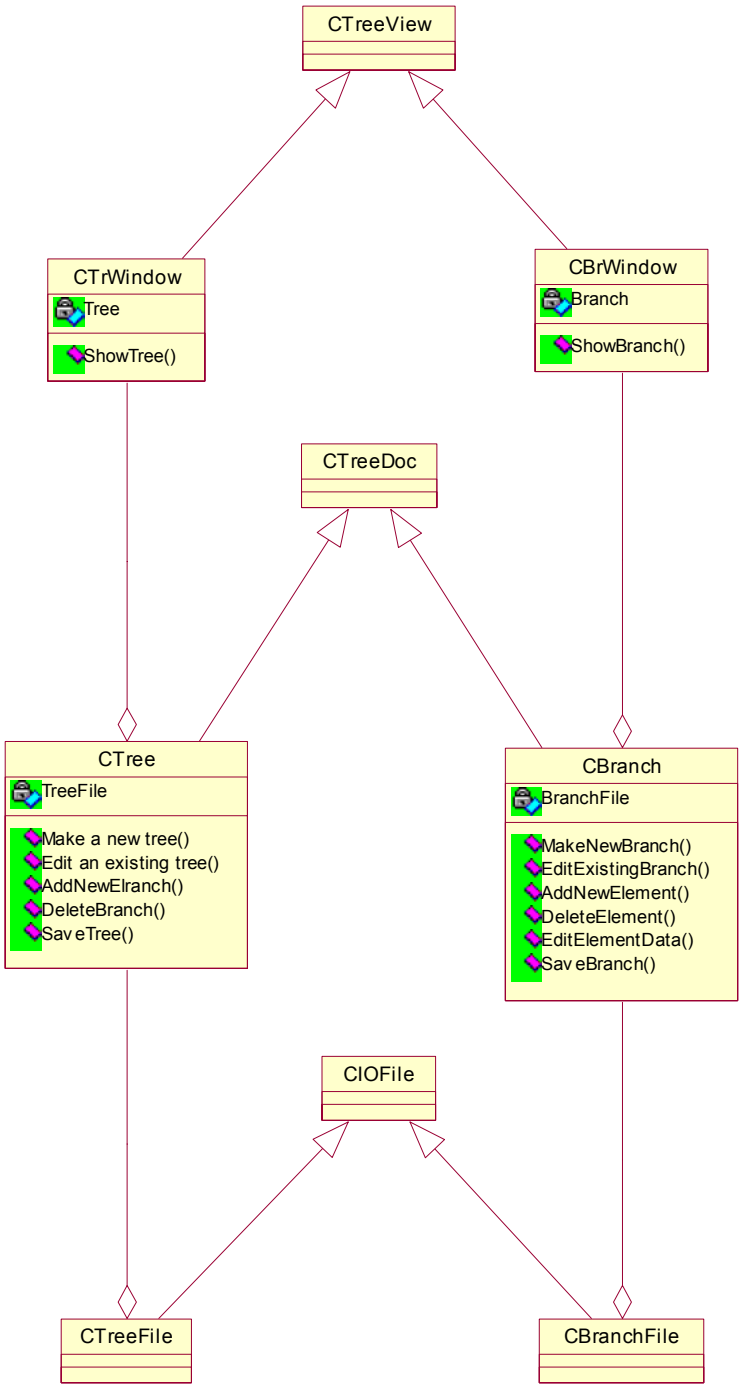
4.2.2 Toteutussuunnitelma

Toteutussuunnitelmassa käytiin läpi toteutettavuus ja järjestelmän toiminta, jotka on selvitetty jo edellisessä vaiheessa. Tärkeimpänä motiivina dokumentin tekemiselle oli arkkitehtuurin suunnittelu.

Periaatteessa TIP-dokumenttiin kuuluvat sekä ohjelmisto-, että järjestelmäarkkitehtuurit, mutta tässä tapauksessa ohjelmisto on tehty uudelleenkäyttäen vanhaa koodia, eikä järjestelmäarkkitehtuuria tarvinnut laatia, koska ohjelmisto liittyy ympäristöön samoin, kuin aikaisemmat versiot. Ohjelmistoarkkitehtuurin pohjana käytettiin käyttötapausmallia.

Käyttötapausmallista saatiin oliomalli, kun toimijoista, käyttäjä poislukien, tehtiin luokkia. Esimerkkinä käytetty malli oli sikäli yksinkertainen, että siitä ohjelmistoarkkitehtuuri, eli looginen malli, syntyi automaattisesti. Luokkien julkiset metodit saatiin suoraan käyttötapauksista, jotka olivat yhteydessä kyseiseen käyttötapaukseen. Kuva 4.4.2 esittää ohjelmistoarkkitehtuurin luokkakaavioesityksenä.

Sekä puu-, että oksakehitystyökalujen ikkuna- ja dokumenttiluokat periytyvät jo olemassaolevasta CTreeView-luokasta, koska tarkoitus on saada aikaiseksi alkuperäisen tyypisiä tietorakenteita, jotka näytetään samalla tavalla, kuin alkuperäisetkin. CIOFile-luokasta periytyvät luokat eivät sisällä tässä vaiheessa vielä metodeita, koska ne jätettiin toteutusvaiheessa laadittaviksi.



Kuva 4.2.2 Ohjelmiston arkkitehtuuri esitettynä UML:n luokkakaaviona.

4.3 Ohjelmiston suunnittelu

Ohjelmiston suunnittelussa lähtökohtana oli analyysivaiheessa aikaansaatu arkkitehtuuri. Tässä vaiheessa metodeihin lisättiin parametrit, paluuarvot ja toiminta kuvattiin UML:n tilakaavioilla.

Suunnitteluvaiheessa dokumentteja kirjoitettiin useampia. Yleisnäkemys suunniteltavasta ohjelmistosta kirjoitettiin Technical specification: Main document – dokumenttiin. Lisäksi jokaista modulia kohti kirjoitettiin oma suunnitteludokumentti.

Yleisnäkemyksen lisäksi päädokumentiin lisättiin myös tarkemmat suunnitelmat pienemmistä vaatimuksista, joita varten ei katsottu olevan tarpeen kirjoitaa kokonaista dokumenttia. Nämä vaatimukset olivat jopa vain yhdellä c-kielisellä rivillä toteutettavia ominaisuuksia. Niinpä dokumentissakin oli c-kielellä kirjoitettuja lauseita. Esimerkkinä pieni lisäys CCodeGen-luokkaan:

```
sCode += "\tnN " +  sTAG + " ++\n\tnLine ++";
```

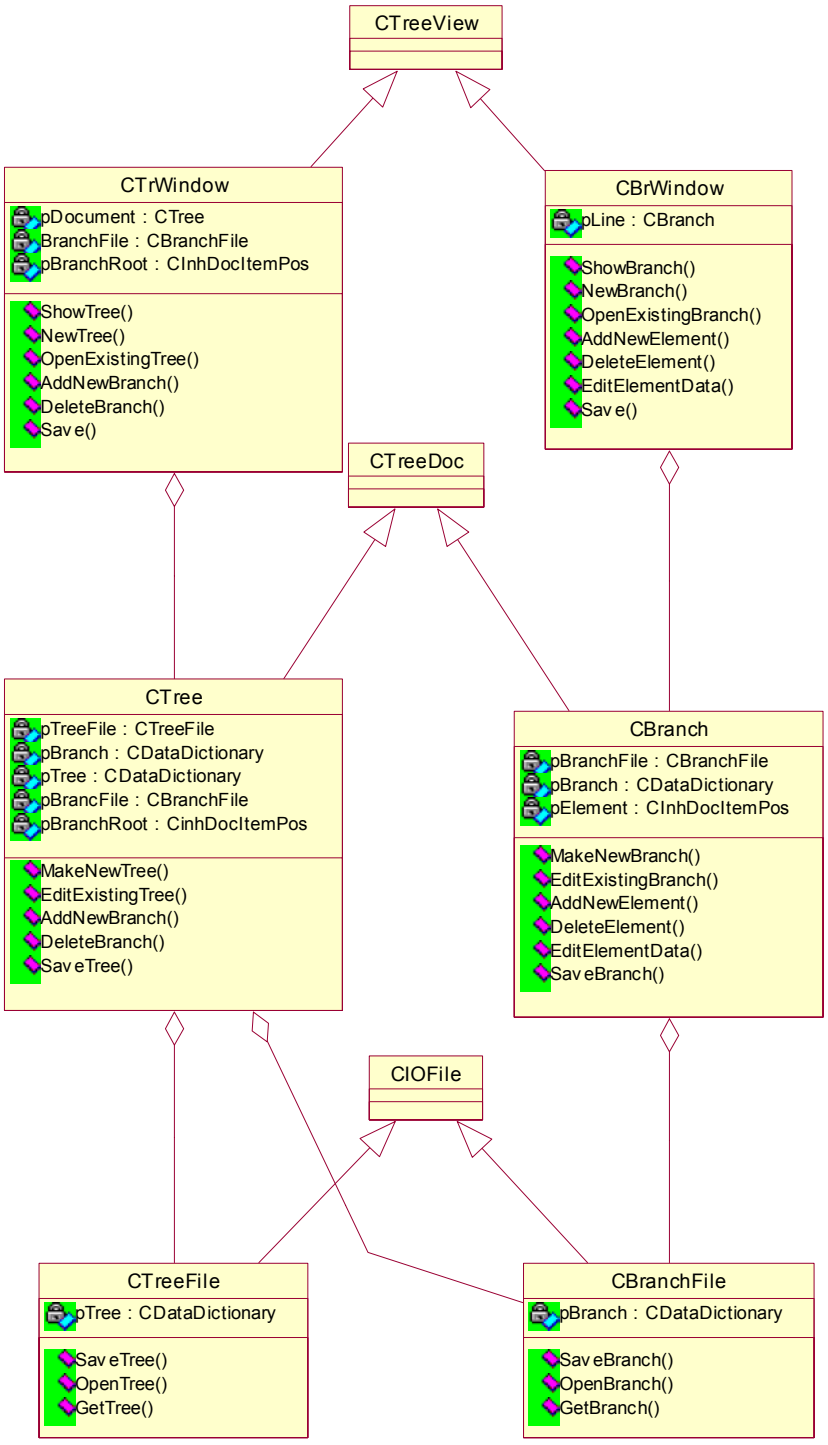
Edellinen lause tuottaa sCode-muuttujaan kaksi riviä RTE-kielistä koodia. Aluksi muuttujan nimeen kirjoitetaan nN-merkkijono, koska sTAG-muuttujan sisältämää merkkijonoa ei tiedetä ja koska RTE-kielessä laskurin nimen tulee alkaa pienellä n-kirjaimella ja seuraavan merkin tulee olla iso kirjain. Lisäksi luetaan rivin alusta tunniste ja kasvatetaan laskurin arvoa yhdellä. Seuraavalla rivillä kasvatetaan nLine-laskuria yhdellä. Näin saadaan toteutettua vaatimus, jossa esitettiin sekä rivityyppien, että kaikkien rivien laskureita.

Koska käyttöliittymä pysyi suurinpiirtein entisellään, ei tehty erillistä käyttöliittymädokumenttia. Päädokumentissa esitettiin myös vaadittavat muutokset käyttöliittymässä.

Aikaisemmin esimerkkeinä käytetyt vaatimukset muodostivat modulin, jolle suunnitteluvaiheen tuloksena syntyi oma tekninen dokumentti, Technical specification: module New format learning.

Ohjelmistomoduulin suunnitteludokumentissa käsiteltiin yksityiskohtaisella tasolla uusien ominaisuuksien toimintaa. Aikaisempaa arkkitehtuurikuvausta täydennettiin tarvittavilla metodeilla ja muuttujilla.

Menetelmänä käytettiin pöytätestausta, eli toimintoja käytiin läpi luokkakaavion perusteella. Näin todettiin, että kaikki vaatimukset kyetään toteuttamaan. Jokainen testattaessa löytynyt uusi metodi ja jäsenmuuttuja lisättiin kaavioon. Kuva 4.3.1 esittää lopullisen luokkakaavion.



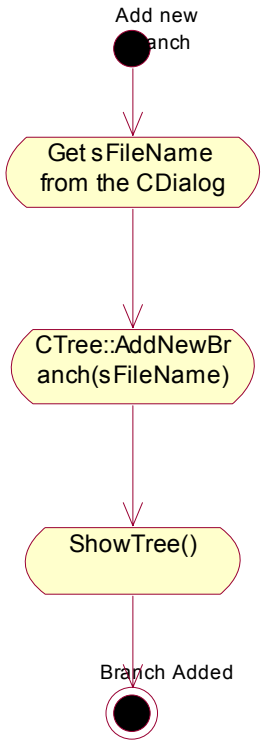
Kuva 4.3.1 Lopullinen luokkakaavio.

Kun luokkakaavioon oli lisätty uudet metodit ja jäsenmuuttujat, käytiin jokaisen metodin toiminta läpi tarkemmin. Työkaluna oli edelleen pöytätestaus. Tulokset esitettiin UML:n tilakaavioina.

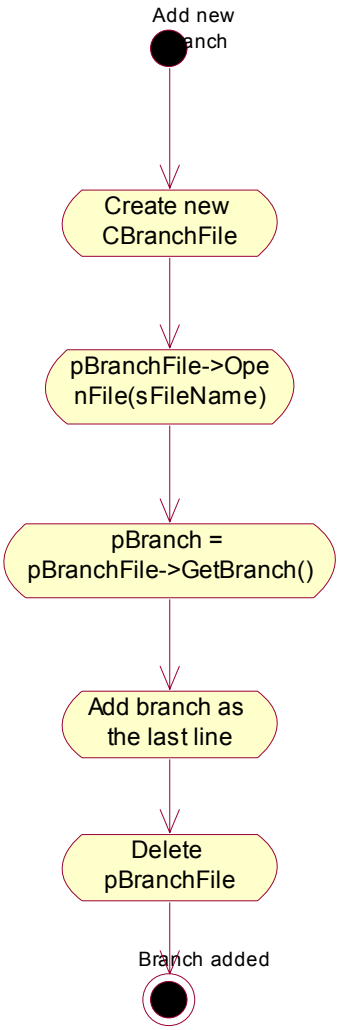
Esimerkkinä käytetään vaatimusta uuden oksan lisääminen puuhun. Aluksi viestinkäsittelijä käynnistää CTrWindow-luokan AddNewBranch()-metodin. Metodi kutsuu CDialog-luokan oliota, jolta saadaan CString-tyyppisen olion sisältämänä avattavan tiedoston nimi.

Kun tiedoston nimi on selvillä, kutsutaan CTree-luokan olion AddNewBranch()-metodia ja välitetään nimi parametrina. Tämä metodi puolestaan luo uuden CBranchFile-luokan olion. Tämän olion OpenFile()-metodi avaa tiedoston. Tiedoston sisältämä data asetetaan tämän jälkeen CBranch-tyyppiseen pBranch-olioon. Olio sijoitetaan sen jälkeen tietorakenteessa viimeiseksi oksaksi ja lopuksi tuhotaan pBranchFile-olio.

Lopuksi palataan CTrWindow::AddNewBranch()-metodiin. Tämän jälkeen tehtävänä onkin enää päivittää näyttö, jonka tekee ShowTree()-metodi. Kuvat 4.3.2 ja 4.3.3 esittävät tapahtumaketjun tilakaavioina.



Kuva 4.3.2 Metodi CtrWindow::AddNewBranch-metodi tilakaaviona.



Kuva 4.3.3 Metodi CTree::AddNewBranch(sFileName) tilakaaviona.

5 LOPPUTULOKSET JA JOHTOPÄÄTÖKSET

Ohjelmiston kehitysprosessit ohjaavat projektiryhmiä ja –päälliköitä. Kun kaikki projektit tehdään saman mallin mukaan, on projektista toiseen siirtyminen helppoa ja nopeaa. Tämä sallii joustavan resurssien käytön ja säästää näin kustannuksia, kuten ohjelmistomoduulien uudelleenkäyttökin.

Kehitysprosessit tuovat projekteihin vaihejakomallin. Ohjelmistosta saadaan näin tarkka kuva jo ennen ohjelmoinnin aloittamista. Toteutusvaihe nopeutuu näin merkittävästi ja lisäksi kustannukset kyetään laskemaan tarkemmin. Ohjelmistokehityksen prosessoiminen auttaa yritystä laskemaan tarkemmin niin kustannukset, resurssit, kuin myös toteuttamisen vaatiman ajan.

Tarkemmat laskelmat siitä, että kuinka paljon aikaa minkäkin asian tekeminen vie, mahdollistaa myös testauksen tarkemman suunnittelun ja ajoituksen. Testaushenkilökunta on mahdollista varata jo hyvissä ajoin ennen testauksen aloittamista. Lisäksi voidaan varmistaa, että kaikki tarpeellinen testataan.Tämä merkitsee paitsi kustannussäästöjä, myös laadun parantumista. Kustannussäästöjä tulee erityisesti siitä, että ominaisuudet tehdään vain kerran, eikä jo toimitettuun ohjelmistoon myöhemmin uudestaan.

Ohjelmiston vaatimukset saadaan esimerkiksi Soneran prosessimallin mukaan varmasti toteutettua. Tämä johtuu tarkasta dokumentoinnista, jossa otetaan huomioon jo aikaisemmin kirjoitetut dokumentit ja dokumenttien jatkuva päivitys. Vaatimusten jäljitettävyyteen käytetään kahta erilaista menetelmää, jolloin suoraan vaatimusdokumentista tai erillisestä taulukosta voidaan tarkastaa, että jokainen asiakkaan kanssa sovittu ohjelmiston ominaisuus on toteutettu.

Ongelmiakin toki löytyy. Suurin ongelma, joka työssä tuli ilmi, oli se, että suuressa organisaatiossa asiakas ja loppukäyttäjä ovat usein eri instansseja. Jos

käy niin, että käyttäjän aivan ehdottomasti tarvitsema tai toivoma ominaisuus maksaa organisaation sisäisen asiakkaan mielestä liikaa, vaatimus jää pois lopullisesta toteutettavasta luettelosta. Lopputulos voi olla käyttäjän mielestä jopa käyttökelvoton ja ohjelmiston kehitys olisi kannattanut jättää analyysivaiheeseen. Analyysivaiheessa mahdolliset tappiot ovat huomattavasti pienemmät, kuin jos verrataan siihen, että tuote on tehty loppuun ja käyttökelvotonta tuotetta markkinoidaan.

Jos ohjelmiston kehitysyksiköllä olisi entistä suurempi mahdollisuus vaikuttaa ohjelmistokokonaisuuteen, pienenisi tämä ongelma, koska kehitysyksiköissä on yleensä aikaisemmin parempi näkemys toteutettavasta kokonaisuudesta. Tämä johtuu siitä, että esimerkiksi käytettävyysasiantuntijat työskentelevät yleensä kehitysyksiköissä.

Yksi mahdollisuus on se, että jos kehitysyksikkö saa osansa lisenssituloista, on motivaatio hyvän tuotteen aikaansaamiseksi entistä suurempi. Tämä voitaisiin toteuttaa esimerkiksi siten, että myös kehitysyksikössä jateaan tulospalkkioita myynnin perusteella. Näin saadaan helpommin asiantuntijat esittämään huonon projektin lopettamista. Nykyään kehitysyksikön ainoa tehtävä on laskuttaa mahdollisimman paljon organisaation sisäisiä asiakkaita, joka ei välttämättä ole kokonaisuuden kannalta hyvä asia. Kuitenkin myös myynnin ja markkinoinnin on nähtävä tuotekehityskustannukset, että osataan tehdä oikeita päätöksiä.

Oikeat ratkaisut löytynevät tutkimuksen avulla. Tavoittena on saada aikaiseksi oikeanlainen tuote oikealla hinnalla ja oikeaan aikaan.

Yhdestä havaitusta vakavasta puutteesta huolimatta prosessointi on edistysaskel ohjelmistojen kehityksen muuttamisessa ohjelmistotuotannoksi. Rakennuksissakin perustukset tehdään ensin, ei katto.

Lopputuloksena voidaan todeta, että ohjelmistoon tehdyt uudet ominaisuudet syntyivät helposti. Kun ajatukset oli selvitetty jo aikaisemmin, ei ohjelmointivaiheessa enää tarvinnut miettiä, mitä tehdään. Lisäksi tarkempi tieto vaatimuksista auttoi tekemään juuri sen, mitä tarvittiin, eikä yhtään enempää. Tämä on tärkeää ohjelmistojen kustannusten kurissa pitämiseksi.

Ohjelmiston laadun kannalta voidaan esittää myös tieto, ettei ohjelmistosta löytynyt yhtään virhettä. Tämä oli erinomainen tulos. Tämä on mahdollista tarkan toiminnallisen suunnittelun lisäksi myös siksi, että toteuttajat tunsivat muokattavan ohjelmiston hyvin. Aivan kaikkea ei siis voida katsoa prosessin tuomaksi hyödyksi.

Uudet ominaisuudet toivat ohjelmistolle aivan uusia käyttömahdollisuuksia. Tämä toi siis lisämahdollisuuksia uudelleenkäyttöä ajatellen. Esitetty prosessimalli ei kuitenkaan sellaisenaan esitä mitään uutta, selkeää dokumentointia lukuunottamatta, uudelleenkäytön tukemiseksi. Uusimpien tutkimusten mukaan tulisikin pohtia arkkitehtuurin erottamista omaksi prosessikseen siten, että uudelleenkäytön tutkiminen liitettäisiin tämän prosessin yhteyteen.

LÄHTEET

1. van Solingen Rini, Berghout Egon: The Goal/Question/Metric Method:a practical guide for quality improvement of software development. The Mc Graw-Hill Publishing Company, 1999, ISBN 007-709553-7.
2. Haikala Ilkka, Märijärvi Jukka: Ohjelmistotuotanto. Suomen ATK-kustannus, 1998, ISBN 0-13-869017-0.
3. Hannus Jouko: Prosessijohtaminen, ydinprosessien uusiminen ja yrityksen suorituskyky. HM&V Research, 1993, ISBN 951-96708-0-7.
4. Laamanen Kai, Tinnilä Markku: Prosessijohtamisen käsitteet. Metalliteollisuuden kustannus, 1998, ISBN 951-817-692-2.
5. Davenport Thomas H.: Process innovation reengineering work through information technology. Boston Harvard Business School Press Cop., 1993, ISBN 0-87584-366-2.
6. Jacobson Ivar, Christerson Magnus, Jonsson Patrik, Övergaard Gunnar: Object-Oriented Software Engineering, A Use Case Driven Approach. Addison-Wesley Publsihing Company, 1995, ISBN 0-201-54435-0.
7. Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John: Design patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1994, ISBN 0-201-63361-2.
8. Sugumaran Vijayan, Tanniru Mohan, StoreyVeda C.: Supporting Reuse In Systems Analysis, Communications of the ACM, marraskuu 2000.
9. Agarwal Ritu, Prasad Jayesh, Tanniru Mohan, Lynch John: Risks of Rapid Application Development, Communications of the ACM, marraskuu 2000.
10. Carnegie Mellon University, tutkimusraportit, raportti CMU/SEI-93-TR-24, saatavissa http://www.cmu.edu [viitattu 30. 10.2000].
11. Nevalainen Risto, SPICE:n arviointilomakkeet. STTF OY, 1999.
12. Booch Grady: UML in Action. Communications of the ACM, lokakuu 1999.

13. Rational:in kotisivut [viitattu 3.9.2001]. Saatavissa: http://www.rational.com
14. Kruglinski David J.: Tehokäyttäjän opas, Visual C++. Suomen ATK-kustannus, 1996, ISBN 951-762-431-X.
15. Sonera Ltd Electronic Commerce Software Solutions: TradeXpress, product overview, 1998.