OHJELMISTON TOIMINNALLISEN KOON LASKENTA

Markku Savolahti

6.3.2002

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

TIIVISTELMÄ

Ohjelmistojen mittaamista tarvitaan, jotta niiden ominaisuuksia voitaisiin arvioida. Toimintopistearvo voidaan laskea ohjelmistosta sen elinkaaren eri vaiheissa ja siitä voidaan johtaa useita erilaisia tunnuslukuja. Toimintopistearvon laskemiseen on kehitetty useita menetelmiä. Tässä tutkielmassa keskitytään kahteen menetelmään: IFPUG 4.1 ja COSMIC-FFP 2.1. Näistä IFPUG-menetelmä on ollut hyvin suosittu 1980-luvulla ja 1990-luvun alussa, mutta menetelmän suosio on kuitenkin laskenut viime vuosina. Ongelmaksi on koettu lähinnä menetelmän monimutkaisuus sekä se, ettei IFPUG sovellu helposti uusimpien sovellus-tyyppien kanssa käytettäväksi. COSMIC-FFP kehitettiin 1990-luvun lopussa tuomaan paran-nusta havaittuihin ongelmiin. Tutkielmassa on tarkoitus selvittää kirjallisuuden perusteella näiden kahden menetelmän mukaisen laskennan suorittaminen ja menetelmiin liittyviä tutki-mustuloksia.

Avainsanat: COSMIC-FFP, FPA, IFPUG, toimintopisteanalyysi, toimintopistelaskenta

SISÄLLYSLUETTELO

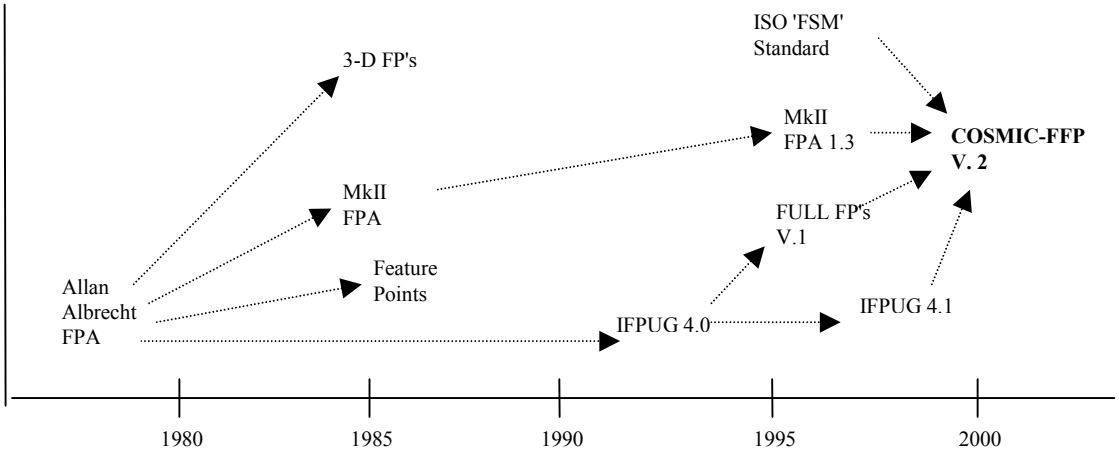
LIITTEET: 1. Esimerkki MIS-sovelluksesta
2. Esimerkki reaaliaikaisesta sovelluksesta

1. JOHDANTO

Ohjelmistojen koon mittaamisen tarpeellisuudesta on puhuttu lähes yhtä kauan, kuin ohjelmistoja on kehitetty. Ohjelmistojen mittaamista tarvitaan, jotta ohjelmistot voitaisiin esimerkiksi hinnoitella, määritellä valmistumisaikataulut ja tarvittava työmäärä.

Ohjelmistojen koon mittaamiseen on ehdotettu erilaisia menetelmiä. Yleinen tapa mitata ohjelmistoja on laskea niiden sisältämä *koodirivimäärä* (lines of code, LOC), mutta se ei välttämättä ole paras tapa mitata ohjelmiston kokoa. Rivimäärää arvioitaessa on vaikea verrata toisiinsa eri ohjelmointikielillä ja -tyyleillä tehtyjä ohjelmistoja. Eräs tapa arvioida ohjelmiston kokoa voisi olla sen vertaaminen aikaisemmin tehtyihin, koska ohjelmistot koostuvat usein hyvin samantapaisista komponenteista. Tämä menetelmä ei sovellu standardina käytettäväksi eikä silloin, jos tehtävä ohjelmisto on ensimmäinen tai se on uuden tyyppinen ohjelmiston kehittäjälle.

Tässä tutkielmassa käsiteltävän toimintopistelaskennan avulla muodostetaan kuva ohjelmiston koosta sen loppukäyttäjälle tarjoaman toiminnallisuuden perusteella. Tämän vuoksi se on riippumaton ohjelmoinnissa käytetystä kielestä ja tyylistä sekä erilaisista ohjelmointivälineistä ja –alustoista. Toimintopistelaskennan periaatteen keksi Alan Albrecht vuonna 1979 työskennellessään IBM:llä. Albrecht kehitti ns. *toimintopisteanalyysin* (Function Point Analysis, FPA), jotta ohjelmiston koko voitaisiin arvioida riippumatta kehityksessä käytettävästä teknologiasta (Symons, 2001). Toimintopisteanalyysistä on kehitetty Albrechtin alkuperäisten sääntöjen pohjalta erilaisia variaatioita (kuva 1.1).



Kuva 1.1 Toiminnallisen koon mittaamisen kehitys (Symons, 2001).

Symonsin (2001) mukaan menetelmä nimeltä *Feature Points* muutti alkuperäistä FPA:ta siten, että voitaisiin mitata ohjelmistossa esiintyvien matemaattisten algoritmien monimutkaisuutta. Samantapainen tarkoitus oli myös menetelmällä *MkII FPA*. Tavoitteena oli ottaa paremmin huomioon MIS-sovellusten (Management Information Systems) sisäisen prosessoinnin monimutkaisuus. Kuvan 1.1 mukaisesti menetelmää on myös pyritty jatkokehittämään. Kolmas variaatio on *3-D Function Points*, joka kehitettiin uudempien reaaliaikaisten ohjelmistojen koon arvioimiseksi. Neljäs haara alkuperäisestä toimintopisteanalyysista on kansainvälisen toimintopistekäyttäjäryhmän (International Function Point Users Group, IFPUG) kehittämät toimintopisteanalyysiversiot IFPUG 4.0 ja IFPUG 4.1. Nimensä mukaisesti tarkoituksena on ollut kehittää kansainvälisesti hyväksytty menetelmä. IFPUG-menetelmä on edelleen ollut lähtökohtana, kun ryhmä kanadalaisia montrealilaisesta yliopistosta kehitti menetelmän nimeltä *Full Function Points* tai *FFP*. Menetelmä perustuu IFPUG:iin arvioitaessa MIS-sovellusten kokoa. Sen sijaan reaaliaikaisten sovellusten laskentaa varten menetelmään on lisätty kuusi uutta komponenttia.

Tässä tutkielmassa keskitytään tarkastelemaan kansainvälisen toimintopistekäyttäjäryhmän kehittämää toimintopisteanalyysiversiota IFPUG 4.1 ja ohjelmistomittauksen kansainvälisen konsortiumin (The Common Software Measurement International Consortium, COSMIC) kehittämää toimintopistemittaria COSMIC-FFP. Kuvan 1.1 mukaisesti COSMIC-FFP perustuu menetelmiin MkII, FFP ja IFPUG 4.1. Lisäksi sen kehittämisessä on hyödynnetty ISO FSM – standardia.

Luvussa 2 käsitellään toimintopistelaskentaan liittyviä asioita hieman yleisemmällä tasolla. Tarkoituksena on esitellä syitä toimintopistelaskennan käyttöönottamiseksi. Luvussa pohditaan myös toimintopistelaskennan suorittamisajankohtaa sekä toimintopistemäärän käyttötapoja.

Luvussa 3 kerrotaan esimerkkisovelluksen avulla, kuinka toimintopistelaskenta suoritetaan käyttäen IFPUG 4.1-menetelmää. Menetelmän esittelyn lisäksi tarkastellaan menetelmään kohdistunutta kritiikkiä ja parannusehdotuksia. Kritiikkiä menetelmä on saanut lähinnä monimutkaisuudestaan, ja parannusehdotukset ovatkin yrityksiä yksinkertaistaa toimintopistelaskennan suorittamista. Luvussa 3 esitetään myös esimerkin avulla mahdollisuus mallinnuskielen ja IFPUG 4.1:n yhteensovittamiseksi.

Luvussa 4 esitellään uusin toimintopistelaskentamenetelmä eli COSMIC-FFP 2.1. Luvun alkuosassa selvitetään uuden menetelmän syntyyn vaikuttaneita syitä. COSMIC-FFP:n suorittamista selvitetään kahden esimerkkisovelluksen avulla. Toinen esimerkkisovellus on sama kuin luvussa 3 IFPUG 4.1:n yhteydessä käytetty, joten voidaan havainnollistaa menetelmien erot käytännössä. Luvun lopuksi esitellään kirjallisuuden perusteella tutkimustulos COSMIC-FFP:n soveltuvuudesta työmäärän arviointiin.

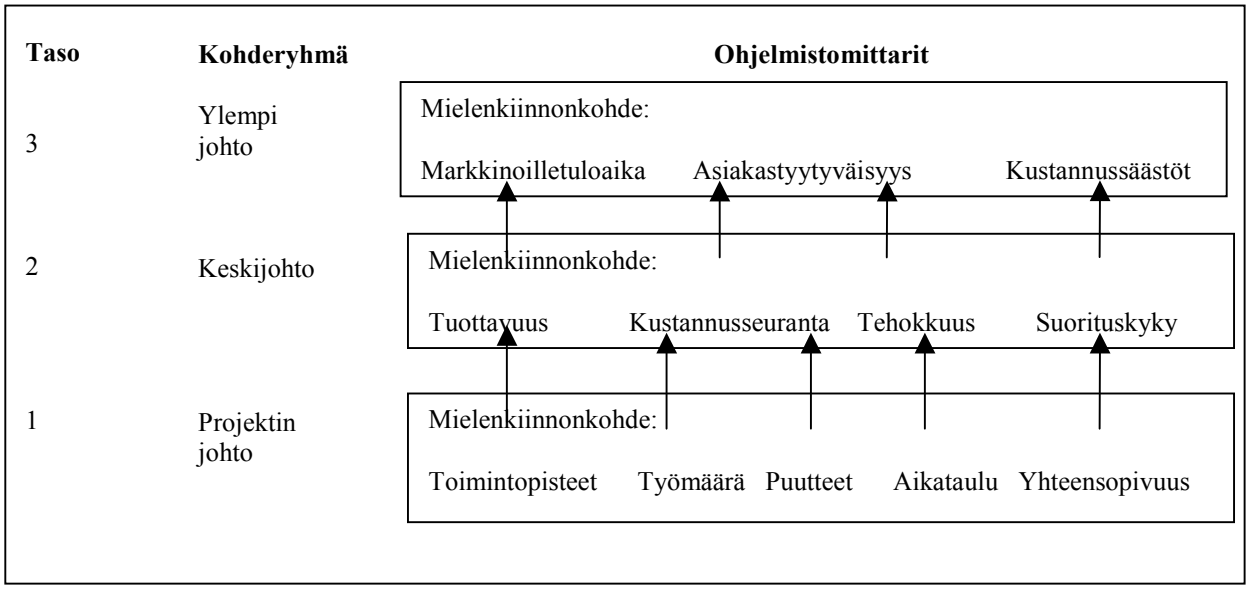
Luvussa 5 esitetään arvioita IFPUG:in ja COSMIC-FFP:n hyvistä ja huonoista puolista sekä menetelmien tulevaisuuden näkymiä ja kehittämistarpeita.

2. OHJELMISTON TOIMINNALLISEN KOON MITTAAMINEN

Jotta jokin yritys alkaisi käyttää toimintopistelaskentaa ohjelmistojensa koon arvioimiseen, täytyy siihen löytyä hyvät perusteet. Ohjelmiston mittaamisen tärkeys pitää pystyä perustelemaan organisaation eri tasoille. Mittaamisen vaikutusten tulee näkyä tuottavuuden ja laadun parantumisena sekä lopulta taloudellisena hyötynä.

2.1 Ohjelmistojen mittaamisen tarve

Kysymällä "ketkä hyötyvät ohjelmistojen mittaamisesta?" saadaan myös vastaus kysymykseen "miksi ohjelmistojen mittaamista tarvitaan?" (Garmus ja Herron, 2000). Kuvassa 2.1 näkyy ohjelmistomittaustoimintojen luokittelu organisaation eri tasoilla.



Kuva 2.1 Ohjelmistomittaustoimintojen luokittelu (Garmus ja Herron, 2000).

Alimmalla tasolla on projektin johto. Projektipäällikkö on vastuussa projektin etenemisestä. Häntä kiinnostaa ensisijaisesti, milloin sovellus on valmis toimitettavaksi. Sen arvioimiseksi pitää olla käytettävissä huomattava määrä tietoa sovelluksen koosta (toimintopisteet), aikataulusta, projektikohtaisista kustannuksista sekä mahdollisten muutospyyntöjen hallinnasta.

Toisella tasolla on keskijohto, jonka vastuulla on useamman projektin loppuunsaattaminen yhtäaikaisesti. Keskijohdolle merkittävät tiedot koskevat tuottavuutta, kustannuksia, tehokkuutta ja suorituskykyä. Keskijohto ei ole niinkään kiinnostunut siitä, toimiiko sovellus vai ei.

Keskijohto huolehtii siitä, että sovellus annetaan ylläpidosta vastuussa oleville tahoille. Tiedot keskijohdolle toimitetaan projektikohtaisesti. Keskijohto kokoaa nämä projektikohtaiset tiedot yhteen ja laskee erilaisia prosenttiosuuksia, kuten kuinka moni projekti on valmistunut ajallaan, budjetin sallimissa rajoissa jne. Tällä tasolla siis kootaan tiedot yhteen ja arvioidaan kokonaisvaltaisesti suorituskykyä.

Hierarkian yläpäässä sijaitsee ylempi johto, joka tarvitsee myös tietoa siitä, kuinka sen johtama liiketoiminta sujuu. Ylemmän johdon pitäisi saada käyttöönsä määrällistä päätöksentekoa tukevaa tietoa. Kohdassa 2.4 tarkastellaan kuinka organisaation eri tasoille saadaan hyödyllistä tietoa.

2.2 Ohjelmiston toiminnallisen koon mittaamisen etuja

Toimintopistelaskenta on riippumaton välineistä ja ohjelmointikielistä, toistettavissa, sen käyttö mahdollistaa vertailun ja sen avulla voidaan estimoida tarvittava työmäärä (Furey, 1997). Jonkin sovelluksen koko saattaa olla esimerkiksi 1500 toimintopistettä. Tällöin koko on 1500 toimintopistettä riippumatta siitä, onko sovellus kirjoitettu COBOL-, C++ vai Smalltalk- ohjelmointikielellä. Sovelluksen vaatima työmäärä ei kuitenkaan ole riippumaton käytetystä ohjelmointikielestä tai välineistä.

Vertailu ohjelmistojen välillä helpottuu toimintopistelaskennan ansiosta. Jos esimerkiksi ohjelmisto A sisältää 640 virhettä sekä 800 toimintopistettä ja ohjelmisto B puolestaan 700 virhettä ja 2200 toimintopistettä, on B:ssä suhteellisesti ottaen vähemmän virheitä. Pelkkä virheiden lukumäärän vertailu johtaa väärään päätelmään (Furey, 1997).

Koska toimintopistelaskenta perustuu ohjelmiston toiminnallisiin vaatimuksiin, on mahdollista tehdä arviointeja ohjelmiston koosta hyvin aikaisessa vaiheessa (Furey, 1997). Ohjelmiston rivimäärän laskeminen on paljolti jo tapahtuneen todentamista ja sopii yhdellä ohjelmointikielellä ohjelmoiville, tosin ns. *takaisinlaskentaa* (Jones, 1999) on esitetty eri ohjelmointikielillä kirjoitettujen ohjelmien vertailemiseksi.

Seuraavassa on kuvaava esimerkki sovelluksen koon laskemisesta sen tarjoaman toiminnallisuuden perusteella (Furey, 1997). Oletetaan, että kaksi ohjelmistoalan yritystä tekee samanlaisen (saman toiminnallisuuden omaavan) ohjelmiston eri ohjelmointivälineillä. Yritys A käyt-

tää COBOL-ohjelmointikieltä ja kirjoittaa 4000 riviä ohjelmakoodia käyttäen aikaa kymmenen henkilötyökuukautta. Yritys B kirjoittaa ohjelmansa Smalltalk-ohjelmointikielellä ja saa aikaan 900 riviä viiden henkilötyökuukauden aikana. Tarkastelemalla näitä lukuja voidaan arvioida, että yritys A on tuottavampi, koska se teki selvästi enemmän ohjelmakoodia. Käyttämällä toimintopisteanalyysiä tilanne muuttuu päinvastaiseksi. Molemmat yrityksethän kirjoittivat ohjelman, jolla oli käyttäjän näkökulmasta katsottuna sama toiminnallisuus. Toimintopisteanalyysi antaa molempien ohjelmien kooksi saman, esimerkiksi 100 toimintopistettä. Yritys B tuotti 100 toimintopistettä viidessä henkilötyökuukaudessa, kun taas yritys A kulutti siihen kaksinkertaisen ajan.

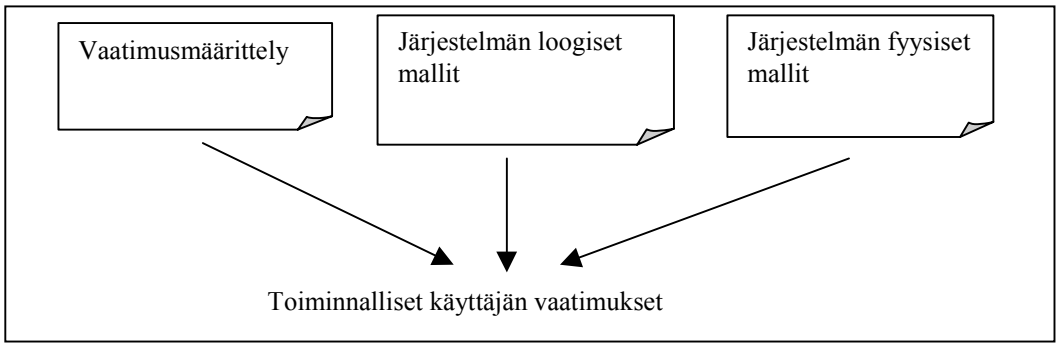
2.3 Ohjelmiston toiminnallisen koon arvioinnin ajankohta

Ohjelmiston toiminnallinen koko voidaan arvioida, kun tulevan sovelluksen toiminnallisuudesta on tietoa saatavilla. Tietoa sovelluksen toiminnallisuudesta kertyy enemmän, kun ohjelmistoprojekti etenee. Arviointi voidaankin tehdä useaan kertaan ohjelmistoprojektin edetessä. Kun sovelluksen toiminnalliset vaatimukset ovat lopulta täysin selvillä, voidaan tehdä tarkka toimintopistelaskenta.

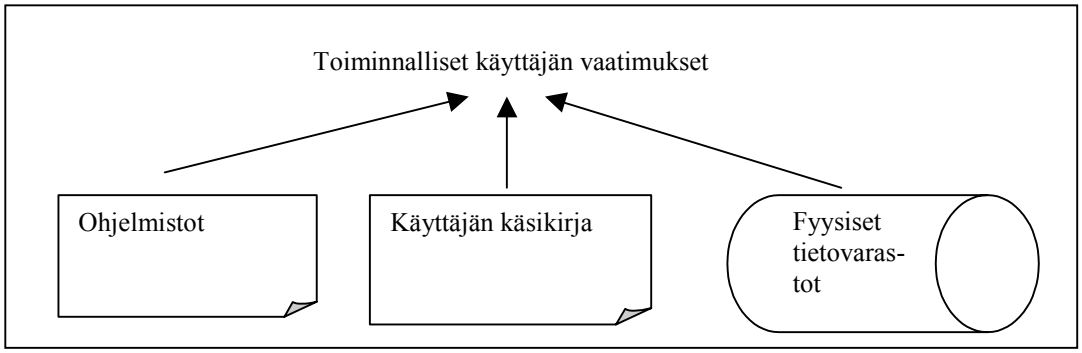
Toiminnallisen koon arviointi vaatii sovelluksen tekijältä systemaattisuutta. Systemaattisuudella tarkoitan tässä yhteydessä erityisesti dokumenttien kirjoittamista sekä mallintamista (kuva 2.2). Niiden avulla kuvataan, mitä sovellus tekee ja minkälaista tietoa se käsittelee. Yksi tunnettu mallinnuskieli on *UML* (Unified Modeling Language) (Booch et al., 1999). UML on alun perin kehitetty tarjoamaan kuvaustapa *oliopohjaisille* (object oriented) sovelluksille, mutta sitä voidaan hyödyntää muissakin sovelluksissa. UML sisältää monia kaaviotyyppejä, joista esimerkiksi käyttötapauskaavio ja luokkakaavio ovat käyttökelpoisia toimintopistelaskennan suorittamiseksi sovelluskehityksen aikaisessa vaiheessa. Hyvä olisi tietenkin, jos sovelluskehityksen alkuvaiheessa määritelty toiminnallisuus ja arvioitu toimintopistemäärä olisi lähellä lopullista toiminnallisuutta ja toimintopistemäärää (kuvat 2.2 ja 2.3).

Yritykselle on tärkeää tietää, kuinka suurta sovellusta se on kehittämässä. Vaatimusmäärittelystä ja malleista saadaan tarvittava tieto toimintopisteanalyysin suorittamiseen (kuva 2.2). Kun lopulta saadaan valmiiksi ohjelmistot ja tarkka kuvaus tarvittavista tiedostoista ja tietokentistä (kuva 2.3), on tarvittava tieto tarkan toiminnallisen koon laskemiseksi käytettävissä. Ohjelmistoprojektin päätyttyä olisi siis syytä tehdä lopullinen toiminnallisen koon arviointi.

Ohjelmistoprojektin aikana tehdyt muut arvioinnit on hyvä ottaa tässä vaiheessa uudelleen tarkasteltaviksi ja tehdä tarvittavat johtopäätökset, jotta jatkossa projektin alkuvaiheessa tehdyt arviot sovelluksen koosta saataisiin entistä tarkemmiksi.



Kuva 2.2 Toiminnallisten vaatimusten määrittely ennen ohjelmiston toteuttamista (Abran et al., 2001b).



Kuva 2.3 Toiminnallisten vaatimusten johtaminen ohjelmiston valmistumisen jälkeen (Abran et al., 2001b).

2.4 Toimintopistearvon käyttötapoja

Garmus ja Herron (2000) esittävät joukon toimintopistearvosta johdettavia tunnuslukuja, joita voi hyödyntää ohjelmistoprojekteissa. Esimerkiksi projektitasolla (vrt. kuva 2.1) tuottavuutta voidaan mitata *tunteina toimintopistettä kohden.* Tällä suhteellisen yksinkertaisella mittarilla saa organisaatio käyttöönsä myös monia erilaisia tuottavuuden vertailulukuja koskien esim. laiteympäristöä tai käytettyä ohjelmointivälinettä. *Toimitusaika* on mittari, jolla mitataan organisaatiolta kuluvaa aikaa loppukäyttäjän vaatimusten mukaisen sovelluksen tuottamiseen. Toimitusaika saadaan toimintopisteiden lukumäärän suhteena kuluneeseen kalenteriaikaan.

Toiminnallisten vaatimusten koko mittaa sovelluksen loppukäyttäjän vaatimien toimintojen kokonaismäärää mitattuna toimintopisteinä. *Täydellisyys* on mitta, jolla verrataan toimitettua

toiminnallisuutta vaadittuun toiminnallisuuteen. Toimintopisteanalyysin kyseessä ollen toiminnallisuus ilmaistaan toimintopistearvona. Loppukäyttäjälle toimitettavan sovelluksen vaadittu toimintopistemäärä voi olla esimerkiksi 100. Vaikka loppukäyttäjä saisikin käyttöönsä sovelluksen, jonka toiminnallisuuteen perustuva koko on 100, ei voida olla varmoja näiden 100 pisteen koostuvan vaaditusta toiminnallisuudesta. Jotta voitaisiin todeta toimitettu sovellus täydelliseksi, täytyy myös verrata vaadittuja toimintoja toimitettuihin.

Kustannus toimintopistettä kohden on tunnusluku, jota tarvitaan yleisimmin projekti- tai organisaatiotasolla (vrt. kuva 2.1). Kaikkien projektien keskimääräistä tunnuslukua voidaan verrata yhden projektin vastaavaan. Kustannusta toimintopistettä kohden voidaan vertailla, kun halutaan tietää kannattaako sovellus ostaa ulkopuolelta vai tehdä itse. Tunnusluku saadaan kokonaiskustannuksien suhteena toimintopisteiden kokonaismäärään.

Toimintopisteanalyysin avulla voidaan tuottaa organisaation eri tasoille hyödyllistä tietoa. Avainsanat toimintopisteanalyysin kohdalla ovat ennustettavuus ja vertailukelpoisuus. Organisaatioille on varmasti hyödyllistä saada projektia koskevat vertailukelpoiset tiedot ohjelmistokehityksen alkuvaiheessa riippumatta käytetystä teknologiasta/ohjelmointikielestä.

3. TOIMINTOPISTEANALYYSI

Symons (2001) määrittelee toimintopisteanalyysin (FPA) suorituksen lyhyesti seuraavalla tavalla: FPA:n suoritukseen vaaditaan viiden komponentin tunnistaminen: syötteet, tulosteet, kyselyt sekä sisäiset ja ulkoiset liitäntätiedostot. Kun nämä komponentit on tunnistettu, lasketaan niille painotettu arvo, joka perustuu komponenttien monimutkaisuuteen. Laskemalla yhteen nämä painotetut arvot saadaan sovittamaton toimintopistearvo, joka lopuksi kerrotaan sovitustekijällä, jolloin saadaan sovitettu toimintopistearvo.

Kohdissa 3.1 - 3.6 tarkastellaan tarkemmin toimintopisteanalyysin suorittamista kuvassa 1.1 mainitun IFPUG 4.1:n mukaisesti. Toimintopisteiden laskemisen voidaan tällöin kuvata tapahtuvan seitsemässä eri vaiheessa (Garmus ja Herron, 2000):

1. Määrittele toimintopistelaskennan tyyppi
2. Määrittele laskennan vaikutusalue ja sovellusalueen rajat
3. Määrittele tietotoiminnot ja niiden monimutkaisuus
4. Määrittele tapahtumatoiminnot ja niiden monimutkaisuus
5. Määrittele sovittamaton toimintopistemäärä
6. Määrittele sovelluksen sovitusarvo
7. Laske sovitettu toimintopistearvo

3.1 Toimintopistelaskennan tyypin määrittäminen

Toimintopistelaskennan tyyppejä ovat: *kehitysprojekti* (development project), *ylläpitoprojekti* (enhancement project) sekä *sovelluksen toimintopistearvo* (application function point counts) (Garmus ja Herron, 2000).

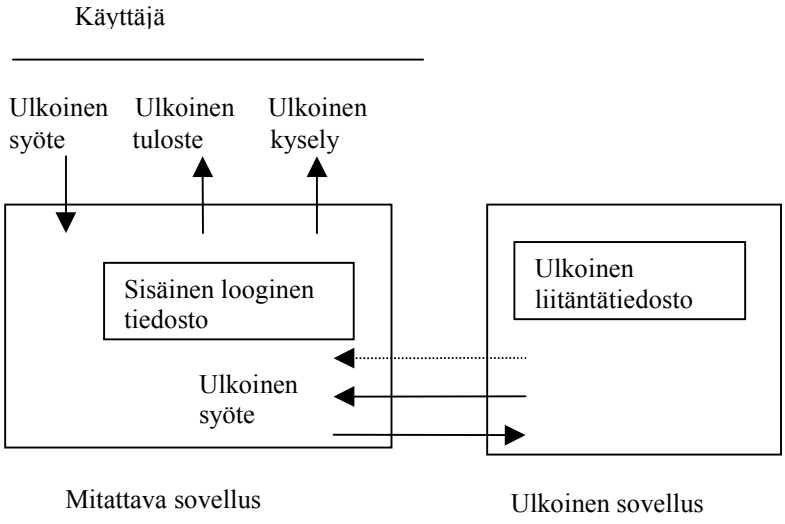
Kehitysprojektissa mitataan ensiasennukseen tulevan ohjelmiston toimintopisteiden lukumäärää. Ylläpitoprojekti tuottaa sovelluksesta uuden version, jolloin mitataan olemassa olevaan sovellukseen tehtyjä muutoksia. Mitattaviin asioihin kuuluvat uusien toimintojen lisääminen sekä olemassa olevien toimintojen poistaminen tai muuttaminen. Sovelluksen toimintopistearvo lasketaan kehitysprojektin tai ylläpitoprojektin tuottamalle sovellukselle.

3.2 Laskennan vaikutusalueen laajuuden ja sovellusaluerajan määrittäminen

Laskennan vaikutusalue määräytyy sen mukaan, mikä on toimintopistelaskennan tarkoitus (Garmus ja Herron, 2000). Mitattavana voi olla kaikki järjestelmään kuuluvat sovellukset, yksittäinen sovellus tai vaikkapa sovelluksen alijärjestelmä (esimerkiksi raportit). *Sovellusalueen raja* on raja mitattavan alueen ja ulkoisten toimijoiden (sovellusten ja/tai käyttäjän) välissä. Rajan määrittämiseksi voidaan soveltaa seuraavaa kolmea sääntöä:

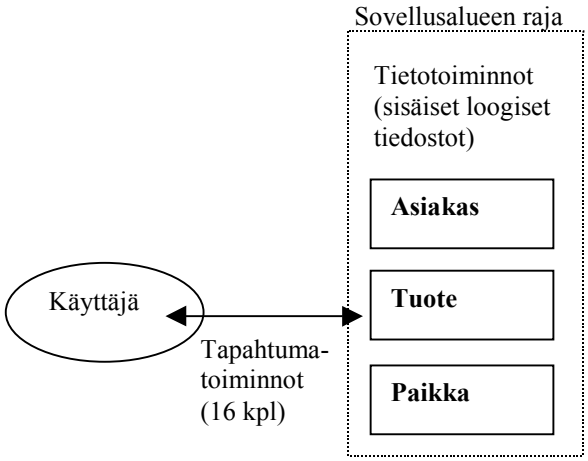
1. Sovellusalueen raja perustuu käyttäjän näkökulmaan. Käyttäjän pitäisi kyetä määrittelemään sovelluksen vaikutusalue ja toiminnallisuus "omalla kielellään".
2. Toisiinsa liittyvien sovellusten välisen rajan tulisi perustua enemmän erilliseen toiminnallisuuteen kuin teknisiin määrittelyihin.
3. Laskennan vaikutusalue ei vaikuta sovellusalueen rajaan. Jos laskettavaan sovellukseen lisätään toimintoja tai siitä poistetaan toimintoja, tehdyt muutokset eivät muodosta omaa sovellusaluettaan, mutta ne vaikuttavat laskennan vaikutusalueeseen.

Toimintopistelaskentaa ja mitattavan sovelluksen rajausta voidaan havainnollistaa kuvan 3.1 mukaisesti.



Kuva 3.1 Toimintopistelaskennan periaate (Garmus ja Herron, 2000).

Liitteen 1 sovellusesimerkissä (Fetcke, 1999) sovellusalueen raja voidaan määritellä kuvan 3.2 osoittamalla tavalla.



Kuva 3.2 Sovellusesimerkin (liite 1) sovellusalueen raja.

3.3 Tietotoimintojen määrittäminen

Tietotoiminnolla tarkoitetaan varastoitua tietoa, joka on sovelluksen käytettävissä päivitystä, viittausta ja hakuja varten (Garmus ja Herron, 2000). Tietotoimintoja ovat *sisäiset loogiset tiedostot* (internal logical files, ILF) sekä *ulkoiset liitäntätiedostot* (external interface files, EIF). Tietotoiminto on loogisesti yhteenkuuluva joukko tietoa eikä tiedon fyysisellä varastointitavalla ole merkitystä toimintopistelaskennan kannalta. Sovelluksella tulisi olla yhtä paljon sisäisiä loogisia tiedostoja ja ulkoisia liitäntätiedostoja riippumatta siitä, onko kyseessä esimerkiksi joukko peräkkäistiedostoja tai tietokanta. Sisäisiä loogisia tiedostoja ylläpidetään sovellusalueen rajan sisäpuolella; ulkoisia liitäntätiedostoja voidaan ainoastaan lukea ja ne sijaitsevat sovellusalueen rajan ulkopuolella kuvan 3.1 mukaisesti.

3.3.1 Sisäiset loogiset tiedostot

Sisäinen looginen tiedosto on *käyttäjän tunnistettavissa oleva loogisesti yhteenkuuluva tietojoukko* tai *ohjaustieto*, jota *ylläpidetään* sovellusalueen rajan sisäpuolella (Garmus ja Herron, 2000). Sisäisen loogisen tiedoston pääasiallinen tarkoitus on säilyttää tietoa, jota ylläpidetään laskennan vaikutusalueen sisällä olevan yhden tai useamman *alkeisprosessin* (elementary process) avulla.

Termillä "käyttäjän tunnistettavissa oleva" viitataan niihin määriteltyihin prosessia ja tietojoukkoa koskeviin vaatimuksiin, jotka sekä ohjelmiston kehittäjä että käyttäjä ovat hyväksyneet ja ymmärtäneet.

Loogisesti yhteenkuuluvalla tarkoitetaan sitä, että sisäisen loogisen tiedoston olemassaolo ei voi olla riippuvainen toisen sisäisen loogisen tiedoston toiminnasta. Esimerkkinä voi mainita osoitetiedoston, joka harvemmin on yksin sisäinen looginen tiedosto, koska se mitä ilmeisemmin on osa jotain muuta, esimerkiksi työntekijätietoja sisältävää, tiedostoa ollen siten riippuvainen toisesta tiedostosta. Jos työntekijän tiedot poistetaan rekisteristä, myös työntekijän osoitetiedot poistuvat. Edellä mainitulla tarkoitetaan siis, että työntekijän tiedot voivat olla fyysisesti eri tiedosto (voivat jopa sijaita eri koneella) kuin työntekijän osoitetiedot, mutta ne kuuluvat loogisesti ajateltuna yhteen ja lasketaan näin ollen toimintopisteanalyysissä yhdeksi ILF:iksi.

Ohjaustietoa sovellus käyttää vaikuttaessaan alkeisprosessin toimintaan. Ohjaustieto määrittelee sen, kuinka tai milloin tietoa käsitellään. Tässä tapauksessa ohjaustieto, erilaiset säännöt tai parametritiedot säilytetään ja ylläpidetään sovellusalueen sisällä. Käyttäjä voi esimerkiksi asettaa tietyn päivämäärän ja kellonajan, jonka jälkeen ohjelma aloittaa suorituksensa.

Tietojoukon ylläpidolla tarkoitetaan sitä, että tietoja muokataan sovelluksen alkeisprosessin kautta. Tiettyä sisäistä loogista tiedostoa voidaan ylläpitää useamman sovelluksen alueella, mutta se lasketaan mukaan vain yhden kerran per sovellus. Alkeisprosessi on pienin toiminto, jolla on merkitystä käyttäjälle. Yhden alkeisprosessin avulla voidaan päivittää yhtä tai useampaa sisäistä loogista tiedostoa. Esimerkiksi laskutussovelluksessa yksi alkeisprosessi saattaisi luoda laskun ja lukea jotain toista tiedostoa selvittääkseen, ovatko asiakkaan luottotiedot kunnossa.

3.3.2 Ulkoiset liitäntätiedostot

Ulkoinen liitäntätiedosto on käyttäjän tunnistettavissa oleva loogisesti yhteenkuuluva tietojoukko tai ohjaustietoa, johon mitattava sovellus viittaa, mutta jota ylläpidetään jonkin toisen sovelluksen alueella (Garmus ja Herron, 2000). Ulkoisen liitäntätiedoston päätarkoitus on säilyttää tietoa, johon viitataan yhdessä tai useammassa mitattavan sovelluksen alueella olevassa alkeisprosessissa. Ulkoiseksi liitäntätiedostoksi määritellyn tietojoukon on oltava jonkin toisen sovelluksen sisäinen looginen tiedosto.

Alkeisprosessi on tässäkin yhteydessä pienin käyttäjälle merkityksellinen toiminto. Alkeispro-

sessi voi lukea yhtä tai useampaa ulkoista liitäntätiedostoa. Esimerkiksi tilaustenkäsittelysovelluksessa voisi olla näyttö, johon luetaan varaston saldo ja varastoartikkelien kuvaukset varastosovelluksen ylläpitämistä tiedostoista. Käyttäjälle pienin merkityksellinen toiminto olisi varastotietojen hakeminen.

3.3.3 Sisäisten loogisten tiedostojen ja ulkoisten liitäntätiedostojen monimutkaisuus

ILF:ien ja EIF:ien lukumäärä ja niiden *toiminnallinen monimutkaisuus* määrittelevät tietotoimintojen osuuden sovelluksen sovittamattomasta toimintopistearvosta (Garmus ja Herron, 2000). Jokaiselle määritellylle ILF:lle ja EIF:lle pitää laskea sen toiminnallinen monimutkaisuus, joka perustuu siinä olevien *tietoalkiotyyppien* (data element type, DET) ja *tietuealkiotyyppien* (record element type, RET) lukumäärään.

Jokaiselle tietotoiminnolle lasketaan taulukon 3.1 mukaisesti monimutkaisuusaste, joka voi olla joko *yksinkertainen*, *keskinkertainen* tai *vaikea*. Monimutkaisuusaste määräytyy tietuealkiotyyppien ja tietoalkiotyyppien lukumäärän mukaan. Tietotoiminnon sovittamaton toimintopistemäärä määräytyy taulukon 3.2 mukaisesti monimutkaisuusasteen perusteella.

Tietoalkiotyypit ovat yksilöllisiä käyttäjän tunnistettavissa olevia ei-toistuvia tietokenttiä tai attribuutteja. Tietuealkiotyypit ovat käyttäjän tunnistettavissa olevia ILF:in tai EIF:in tietoalkioiden aliryhmiä (valinnaisia tai pakollisia)[1]. Käyttäjä voi siis alkeisprosessin avulla käyttää vähintään yhtä tai ei välttämättä yhtään alityyppiä. Esimerkiksi työntekijä voi olla kuukausipalkkainen tai tuntipalkkainen ja käyttäjä voi valita näistä toisen. Työntekijätiedostoon kuuluisi tässä tapauksessa kaksi tietuealkiotyyppiä.

Seuraavat kolme sääntöä pätevät laskettaessa ILF:eille tai EIF:eille tietoalkiotyyppien lukumäärää (Garmus ja Herron, 2000):

1. Laske tietoalkiotyypiksi jokainen yksilöllinen käyttäjän tunnistettavissa oleva, ei-toistuva tietokenttä, jota jokin ILF tai EIF ylläpitää tai lukee. Esimerkiksi opiskelijarekisterissä etunimi, sukunimi ja puhelinnumero ovat kaikki tietoalkiotyyppejä. Ne lasketaan vain yhden kerran riippumatta siitä, kuinka moni tapahtuma niitä päivittää tai lu-

[1] Aliryhmiä esitetään oliosuhdekaaviossa (Elmashri ja Navathe, 2000) alityyppeinä ja UML:n mukaisessa luokkakaaviossa (Booch et al., 1999) aliluokkina.

kee tai kuinka tieto on fyysisesti varastoitu.

2. Kun kaksi tai useampi sovellus ylläpitää ja/tai lukee samaa ILF:iä tai EIF:iä, mutta eri DET:ejä, laske vain kunkin sovelluksen käyttämät DET:it.
3. Laske DET myös sille tiedolle, jonka käyttäjä tarvitsee luodakseen uuden suhteen johonkin toiseen ILF:iin tai EIF:iin. Tästä käytetään yleisesti nimitystä viiteavain.

Seuraavat kaksi sääntöä pätevät laskettaessa ILF:eille tai EIF:eille tietuealkiotyyppien lukumäärää (Garmus ja Herron, 2000):

1. Laske RET jokaiselle valinnaiselle tai pakolliselle ILF:in tai EIF:in alityypille.
2. Jos alityyppejä ei ole, lasketaan ILF tai EIF yhdeksi RET:ksi.

Taulukko 3.1 Tietotoimintojen monimutkaisuusmatriisi (Garmus ja Herron, 2000).

	Tietoalkiotyypit		
Tietuealkiotyypit	1 – 19	20 – 50	>= 51
1	Yksinkertainen	Yksinkertainen	Keskinkertainen
2 – 5	Yksinkertainen	Keskinkertainen	Vaikea
> 5	Keskinkertainen	Vaikea	Vaikea

Taulukko 3.2 Tietotoimintojen mahdolliset toimintopistearvot (Garmus ja Herron, 2000).

Toimintotyyppi	Yksinkertainen	Keskinkertainen	Vaikea
Sisäinen looginen tiedosto (ILF)	7	10	15
Ulkoinen liitäntätiedosto (EIF)	5	7	10

Taulukkoja 3.1 ja 3.2 apuna käyttäen saadaan määriteltyä liitteen 1 sovellusesimerkin tietotoimintojen monimutkaisuus, joka näkyy taulukossa 3.3.

Taulukko 3.3 Sovellusesimerkin (liite 1) tietotoimintojen monimutkaisuus ja sovittamaton toimintopistearvo (Fetcke, 1999).

Numero	Nimi	Tyyppi	RET	DET	Sovittamaton toimintopistearvo
1	Asiakas	ILF	1	3	7
2	Tuote	ILF	1	6	7
3	Paikka	ILF	1	2	7
Yhteensä					21

3.4 Tapahtumatoimintojen määrittely

Ohjelmiston käyttäjän suorittamista toiminnoista käytetään yhteisnimitystä *tapahtumatoimin-*

not (transactional functions) (Garmus ja Herron, 2000). Tapahtumatoimintoja ovat *ulkoiset syötteet* (external inputs, EI), *ulkoiset tulosteet* (external outputs, EO) ja *ulkoiset kyselyt* (external inquiries, EQ). Kaikki nämä kolme tapahtumatoimintoa ovat IFPUG:n terminologian mukaan alkeisprosesseja[2].

3.4.1 Ulkoiset syötteet

Ulkoinen syöte (EI) on sovelluksen alkeisprosessi, joka käsittelee sovellusrajan ulkopuolelta tulevaa tietoa tai ohjaustietoa (Garmus ja Herron, 2000). Käsitelty tieto päivittää yhtä tai useampaa sisäistä loogista tiedostoa (ILF); käsitelty ohjaustieto voi päivittää, mutta sen ei tarvitse päivittää, sisäistä loogista tiedostoa. Ulkoisen syötteen päätarkoitus on ylläpitää yhtä tai useampaa sisäistä loogista tiedostoa ja/tai muuttaa sovelluksen käyttäytymistä *käsittelylogiikkansa* kautta.

Ohjaustiedolla EI:n yhteydessä tarkoitetaan tietoa, jonka avulla sovellus vaikuttaa alkeisprosessin toimintaan. Esimerkiksi järjestelmän yleisiin asetuksiin saatetaan vaikuttaa ohjaustiedolla tai järjestelmän toimintaa saatetaan muuttaa tietyn käyttäjän profiilin mukaiseksi. Ohjaustietoa ei välttämättä tallenneta mihinkään tiedostoon.

Ulkoisista syötteistä puhuttaessa on avainsana päivitys, joka tarkoittaa tiedon muokkaamista alkeisprosessin kautta. Erillisiksi ulkoisiksi syötteiksi voidaan laskea tiedon lisääminen, muuttaminen tai poistaminen. Käsittelylogiikka sisältää käyttäjän asettamat vaatimukset, joilla alkeisprosessi saadaan suoritetuksi loppuun. Käsittelylogiikka voi näin ollen sisältää erilaisia tarkistuksia, tiedon suodattamista tai valitsemista, tiedon muokkaamista oikeaan muotoon, johdetun tiedon luomista ja yhden tai useamman ILF:n päivittämistä tai sovelluksen käyttäytymisen muuttamista.

Ulkoisia syötteitä ovat myös toisilta sovelluksilta tulevat viestit, jotka vaativat prosessointia. Ulkoisiksi syötteiksi ei sen sijaan voida laskea käyttäjän syöttämiä tietoja, joita vaaditaan ulkoisen kyselyn tai tulosteen suorittamiseksi. Toisin sanoen, kun käyttäjä esimerkiksi haluaa tulostaa raportin paperille, saattaa sovellus pyytää käyttäjää antamaan esimerkiksi tietyn päivämäärävälin. Sovellusta toteutettaessa tietty toiminto voidaan saada aikaan useallakin eri ta-

[2] Alkeisprosessi voidaan UML:n (Booch et al., 1999) terminologian mukaisesti ymmärtää käyttötapauksena.

valla, mutta se lasketaan vain yhdeksi EI:ksi.

3.4.2 Ulkoisten syötteiden monimutkaisuuden määrittely

Ulkoisten syötteiden (EI) lukumäärä yhdessä jokaisen EI:n toiminnallisen monimutkaisuuden kanssa määrittelee ulkoisten syötteiden osuuden sovelluksen sovittamattomasta toimintopistearvosta. Jokaiselle yksilöidylle EI:lle pitää laskea sen toiminnallinen monimutkaisuus, joka perustuu EI:n viittaamien tietoalkiotyyppien (DET) ja *tiedostotyyppien* (file types referenced, FTR) lukumääriin taulukon 3.4 mukaisesti (Garmus ja Herron, 2000). Ulkoisten syötteen sovittamaton toimintopistemäärä saadaan taulukon 3.5 mukaisesti toiminnallisuuden monimutkaisuuden perusteella.

Hieman yksinkertaistettuna sanoen ulkoisten syötteiden monimutkaisuuden määrittelyssä tarkastellaan, kuinka moneen tiedostoon (FTR) ja niissä esiintyvään tietokenttään (DET) ulkoinen syöte (alkeisprosessi) viittaa.

Taulukko 3.4 Ulkoisen syötteen monimutkaisuusmatriisi (Garmus ja Herron, 2000).

	Tietoalkiotyypit		
Viitatut tiedostotyypit	1 – 4	5 – 15	>= 16
< 2	Yksinkertainen	Yksinkertainen	Keskinkertainen
2	Yksinkertainen	Keskinkertainen	Vaikea
> 2	Keskinkertainen	Vaikea	Vaikea

Taulukko 3.5 Tapahtumatoimintojen mahdolliset toimintopistearvot (Garmus ja Herron, 2000).

Toimintotyyppi	Yksinkertainen	Keskinkertainen	Vaikea
Ulkoinen syöte (EI)	3	4	6
Ulkoinen tuloste (EO)	4	5	7
Ulkoinen kysely (EQ)	3	4	6

Seuraavien IFPUG:in määrittelemien sääntöjen tulee olla voimassa laskettaessa tietoalkiotyyppien lukumäärää EI:lle (Garmus ja Herron, 2000):

1. Laske yksi DET jokaisesta käyttäjän tunnistettavissa olevasta, ei-toistuvasta kentästä tai attribuutista (myös toissijaisesta avaimesta), joka ylittää sovellusalueen rajan (sovellusalueen rajan ulkopuolelta sisäpuolelle tai päinvastoin) ja joka tarvitaan alkeisprosessin loppuun suorittamiseksi.

2. Älä laske DET:iksi kenttää, jota käyttäjä ei ole syöttänyt (ei siis ylitä sovellusalueen rajaa), mutta jonka sovellus EI:n avulla hakee tai johtaa ja ylläpitää sillä ILF:iä (esim. systeemistä haettu päiväys).
3. Laske yksi DET loogisesti yhteenkuuluvasta kentästä, joka fyysisesti tallennetaan useampana kenttänä ja joka saattaa vaatia käyttäjän syöttämään useamman rivin (esimerkiksi osoitetiedot).
4. Laske kaikki sovelluksen mahdollisesti antamat ilmoitukset tietyn alkeisprosessin osalta (virheilmoitukset, tieto käsittelyn päättymisestä jne.) yhdeksi DET:ksi.
5. Laske yksi DET EI:n vaatiman toiminnon suorittamiseksi, vaikka käyttäjällä olisi useampi mahdollisuus suorittaa toiminto.

Seuraavien IFPUG:in määrittelemien sääntöjen tulee olla voimassa laskettaessa EI:n viittaamien tiedostotyyppien lukumäärää (Garmus ja Herron, 2000):

1. Laske yksi FTR jokaista ILF:iä kohden, jota EI:n alkeisprosessi päivittää.
2. Laske yksi FTR jokaista EI:n suorittamisen aikana luettua ILF:iä ja EIF:iä kohden.
3. Laske vain yksi FTR jokaista ILF:iä kohden, jota sekä luetaan että päivitetään.

Taulukossa 3.6 näkyy kuvaukset esimerkkisovelluksen (liite 1) ulkoisista syötteistä (Fetcke, 1999). Taulukossa 3.6 esitettyjen tietoalkiotyyppien lisäksi kuhunkin ulkoiseen syötteeseen pitää laskea kuuluvaksi yksi DET ilmoituksista ja yksi DET käyttäjän suorittamista toimenpiteistä.

Taulukko 3.6 Esimerkkisovelluksen (liite 1) ulkoiset syötteet.

Ulkoinen syöte	DET	FTR
Lisää Asiakas	Nimi, osoite	Asiakas
Muuta asiakkaan tietoja	Nimi, osoite, maksumäärä	Asiakas, tuote
Poista asiakas	Nimi	Asiakas, tuote
Ota maksu	Nimi, maksumäärä	Asiakas
Tallenna tuote	Kuvaus, kuorma, arvo, omistaja, varastopaikka	Asiakas, tuote, paikka
Nouda tuote	Kuvaus, omistaja	Asiakas, tuote
Lisää uusi paikka	Sijainti, tila	Paikka
Muuta paikan tietoja	Sijainti, tila	Paikka, tuote
Poista paikka	Sijainti	Paikka, tuote

Taulukossa 3.7 on lueteltu liitteen 1 esimerkkisovelluksen sovittamattomat toimintopistemäärät ulkoisten syötteiden osalta.

Taulukko 3.7 Liitteen 1 esimerkin ulkoisten syötteiden monimutkaisuus ja toimintopisteet (Fetcke, 1999).

Numero	Nimi	Tyyppi	DET	FTR	Toimintopisteet
1	Lisää asiakas	EI	4	1	3
2	Muuta asiakkaan tietoja	EI	5	2	4
3	Poista asiakas	EI	3	2	3
4	Ota maksu	EI	4	1	3
5	Tallenna tuote	EI	7	3	6
6	Nouda tuote	EI	4	2	3
7	Lisää paikka	EI	4	1	3
8	Muuta paikan tietoja	EI	4	2	3
9	Poista paikka	EI	3	2	3
Yhteensä:					31

3.4.3 Ulkoiset tulosteet

Ulkoinen tuloste (EO) on sovelluksen alkeisprosessi, joka tuottaa tietoa tai ohjaustietoa sovellusrajan ulkopuolelle (Garmus ja Herron, 2000). Ulkoisen tulosteen päätarkoitus on esittää tietoa käyttäjälle käsittelylogiikkaansa avulla, joka sisältää muutakin kuin tiedon tai ohjaustiedon haun. Käsittelylogiikan pitää sisältää vähintään yksi matemaattinen kaava tai laskentaa, luoda johdettua tietoa, ylläpitää yhtä tai usampaa ILF:iä ja/tai muuttaa järjestelmän käyttäytymistä.

Tieto on johdettua, jos siihen kuuluu jotain muutakin kuin vain tiedon hakua. Esimerkiksi tiedon muodostaminen useasta EIF:stä tai ILF:stä saadusta tiedosta on johdettua tietoa. Esimerkkeinä ohjaustiedosta voisivat olla erilaiset viestit, joita järjestelmä omasta toiminnastaan tai sen hetkisestä tilastaan antaa käyttäjälle.

Kun sovellus lähettää toiselle sovellukselle johdettua tai laskentaa vaatinutta tietoa/tiedostoja, on kyseessä EO lähettävälle ja EI vastaanottavalle sovellukselle. Ulkoisia tulosteita eivät ole esimerkiksi tiedon uudelleenjärjestely ilman muunlaista käsittelylogiikkaa, viestit onnistumisesta tai virheilmoitukset epäonnistuneesta tiedon tallentamisesta.

3.4.4 Ulkoisten tulosteiden monimutkaisuuden määrittely

Ulkoisten tulosteiden lukumäärä yhdessä jokaisen EO:n toiminnallisen monimutkaisuuden kanssa määrittelee ulkoisten tulosteiden osuuden sovelluksen sovittamattomasta toimintopistearvosta. Jokaiselle yksilöidylle EO:lle pitää laskea sen toiminnallinen monimutkaisuus, joka

perustuu EO:n viittaamien tietoalkiotyyppien (DET) ja tiedostotyyppien (FTR) lukumääriin taulukoiden 3.5 ja 3.8 mukaisesti (Garmus ja Herron, 2000).

Taulukko 3.8 Ulkoisen tulosteen monimutkaisuusmatriisi (Garmus ja Herron, 2000).

	Tietoalkiotyypit		
Viitatut tiedostotyypit	1 – 5	6 – 19	>= 20
< 2	Yksinkertainen	Yksinkertainen	Keskinkertainen
2 – 3	Yksinkertainen	Keskinkertainen	Vaikea
> 3	Keskinkertainen	Vaikea	Vaikea

Seuraavien IFPUG:in määrittelemien sääntöjen tulee olla voimassa laskettaessa tietoalkiotyyppien lukumäärää EO:lle (Garmus ja Herron, 2000):

1. Laske yksi DET jokaisesta käyttäjän tunnistamasta, ei-toistuvasta kentästä, joka siirtyy sovellukseen sovellusalueen rajan yli ja jota tarvitaan määrittelemään mitä, milloin ja/tai kuinka alkeisprosessi hakee ja/tai johtaa tarvittavan tiedon. Tällaisia kenttiä ovat usein erilainen ohjaustieto ja parametrit tai tarvittavaa tietoa suodattava tieto.
2. Laske yksi DET jokaisesta käyttäjän tunnistamasta, ei-toistuvasta kentästä tai attribuutista, joka siirtyy sovelluksesta sovellusalueen rajan yli.
3. Laske vain yksi DET, vaikka DET ylittää sovellusalueen rajan molempiin suuntiin.
4. Laske vain yksi DET alkeisprosessin antamista erilaisista viesteistä (esim. jonkin toiminnon onnistunut suorittaminen, virheilmoitukset)
5. Laske vain yksi DET, vaikka käyttäjä voisikin suorittaa EO:n (saman loogisen prosessin) eri tavoilla, (esim. OK-näppäin, hiiren näppäin tai funktionäppäin).
6. Älä laske sivutukseen liittyviä tai järjestelmän itsensä luomia merkkejä, kuten sivunumerot tai järjestelmän luomat päiväys- tai kellonaikakentät.
7. Älä laske vakiomerkintöjä, kuten raportin otsikkotietoja, näytön tunnistetietoja tai kenttien otsikkotietoja.
8. Älä laske niitä ILF:n tietokenttiä, joita päivitetään EO:n alkeisprosessin aikana, jos ne eivät ylitä sovellusalueen rajaa.
9. Laske vain yksi DET useasta loogisesti yhteenkuuluvasta kentästä, kuten päivämäärä tai henkilön nimi.
10. Laske yksi DET tekstitiedosta, joka koostuu yhdestä sanasta, lauseesta, kappaleesta tai monesta kappaleesta (esim. erilaiset huomautustiedot).
11. Laske yksi DET jokaisesta otsikkotyypistä ja numeerisesta vastineesta graafisessa kuviossa. Esimerkiksi piirakkakaaviossa voi olla kaksi DET:iä, toinen on luokka (osiot,

joihin kaavio on jaettu) ja toinen luokkaa vastaava prosenttimäärä.

Seuraavien IFPUG:in määrittelemien sääntöjen tulee olla voimassa laskettaessa EO:n viittaamien tiedostotyyppien lukumäärää (Garmus ja Herron, 2000):

1. Laske yksi viitattu tiedostotyyppi (FTR) jokaisesta sisäisestä loogisesta tiedostosta (ILF) tai ulkoisesta liitäntätiedostosta (EIF), jota luetaan suoritettaessa EO:ta
2. Laske yksi FTR jokaisesta EO:n alkeisprosessin päivittämästä ILF:stä.
3. Laske vain yksi FTR jokaisesta ILF:stä, jota EO:n alkeisprosessi sekä päivittää että lukee.

Taulukossa 3.9 näkyy kuvaukset esimerkkisovelluksen (liite 1) ulkoisista tulosteista (Fetcke, 1999). Taulukossa 3.9 esitettyjen tietoalkiotyyppien lisäksi kuhunkin ulkoiseen tulosteeseen pitää laskea kuuluvaksi yksi DET ilmoituksista ja yksi DET käyttäjän suorittamista toimenpiteistä. Taulukossa 3.10 näkyy esimerkkisovelluksen ulkoisten tulosteiden monimutkaisuus.

Taulukko 3.9 Esimerkkisovelluksen (liite 1) ulkoiset tulosteet.

Ulkoinen tuloste	DET	FTR
Tulosta asiakkaan tuotteet	Nimi, kuvaus, arvo, varastointipäivä, tuotteiden kokonaismäärä ja kokonaisarvo	Asiakas, tuote
Tulosta lasku	Nimi, osoite, maksumäärä, tuotteiden kokonaismäärä	Asiakas, tuote
Tulosta lista varastoiduista tuotteista	Kuvaus, kuorma, varastopaikka, tuotteiden kokonaismäärä	Tuote

Taulukko 3.10 Liitteen 1 esimerkin ulkoisten tulosteiden monimutkaisuus (Fetcke, 1999).

Numero	Nimi	Tyyppi	DET	FTR	Toimintopisteet
1	Tulosta asiakkaan tuotteet	EO	8	2	5
2	Tulosta lasku	EO	6	2	5
3	Tulosta lista varastoiduista tuotteista	EO	6	1	4
Yhteensä:					14

3.4.5 Ulkoiset kyselyt

Ulkoinen kysely (EQ) on sovelluksen alkeisprosessi, joka tuottaa tietoa tai ohjaustietoa sovellusrajan ulkopuolelle (Garmus ja Herron, 2000). Ulkoisen kyselyn päätarkoitus on esittää tietoa käyttäjälle hakemalla tietoa tai ohjaustietoa sisäisestä loogisesta tiedostosta tai ulkoisesta

liitäntätiedostosta. Käsittelylogiikka ei saa sisältää yhtään matemaattista kaavaa tai laskentaa eikä sillä voida luoda minkäänlaista johdettua tietoa. Ulkoinen kysely ei saa päivittää sisäisiä loogisia tiedostoja (ILF) eikä se saa muuttaa sovelluksen käyttäytymistä.

Tyypillinen ulkoinen kysely on tiedon hakeminen yhdestä tai useammasta ILF:stä ja/tai EIF:stä ja sen esittäminen näytöllä tai tulostaminen paperille. Ulkoisia kyselyjä ovat myös tiedostot, joita sovellus lähettää toisille sovelluksille. Näiden tiedostojen lähettäminen ei saa sisältää laskentaa tai johdettua tietoa eikä näillä tiedoilla voi päivittää lähettävän sovelluksen ILF:iä. Sen sijaan tiedostoilla voidaan päivittää vastaanottavan sovelluksen ILF:iä, jolloin tapahtuma on lähettävälle sovellukselle EQ ja vastaanottavalle sovellukselle EI.

3.4.6 Ulkoisten kyselyiden monimutkaisuuden määrittely

Ulkoisten kyselyiden lukumäärä yhdessä jokaisen EQ:n toiminnallisen monimutkaisuuden kanssa määrittelee ulkoisten kyselyiden osuuden sovelluksen sovittamattomasta toimintopistearvosta. Jokaiselle yksilöidylle EQ:lle pitää laskea sen toiminnallinen monimutkaisuus, joka perustuu EQ:n viittaamien tietoalkiotyyppien (DET) ja tiedostotyyppien (FTR) lukumääriin taulukoiden 3.5 ja 3.11 mukaisesti (Garmus ja Herron, 2000).

Taulukko 3.11 Ulkoisen kyselyn monimutkaisuusmatriisi (Garmus ja Herron, 2000).

	Tietoalkiotyypit		
Viitatut tiedostotyypit	1 – 5	6 – 19	>= 20
1	Yksinkertainen	Yksinkertainen	Keskinkertainen
2 – 3	Yksinkertainen	Keskinkertainen	Vaikea
> 3	Keskinkertainen	Vaikea	Vaikea

Tietoalkiotyyppien lukumäärää laskettaessa pätevät muuten samat säännöt kuin ulkoisille tulosteille, mutta sääntö 8 ei sovellu ulkoisille kyselyille.

Määriteltäessä ulkoisen kyselyn viittaamien tiedostotyyppien lukumäärää lasketaan yksi viitattu tiedostotyyppi (FTR) jokaisesta sisäisestä loogisesta tiedostosta (ILF) tai ulkoisesta liitäntätiedostosta (EIF), jota luetaan suoritettaessa EQ:ta (Garmus ja Herron, 2000).

Taulukossa 3.12 näkyy kuvaukset esimerkkisovelluksen (liite 1) ulkoisista kyselyistä (Fetcke, 1999). Taulukossa 3.12 esitettyjen tietoalkiotyyppien lisäksi kuhunkin ulkoiseen kyselyyn pitää laskea kuuluvaksi yksi DET ilmoituksista ja yksi DET käyttäjän suorittamista toimenpi-

teistä. Taulukossa 3.13 näkyy esimerkkisovelluksen ulkoisten kyselyiden monimutkaisuus.

Taulukko 3.12 Esimerkkisovelluksen (liite 1) ulkoiset kyselyt.

Ulkoinen kysely	DET	FTR
Asiakaskysely	Nimi, osoite, maksumäärä	Asiakas
Kysely asiakkaan tuotteista	Nimi, kuvaus, kuorma, arvo, varastointipäivä	Asiakas, tuote
Kysely paikoista	Sijainti, tila	Paikka
Kysely varastoiduista tuotteista	Sijainti, tila, kuvaus, kuorma	Paikka, tuote

Taulukko 3.13 Liitteen 1 esimerkin ulkoisten kyselyiden monimutkaisuus (Fetcke, 1999).

Numero	Nimi	Tyyppi	DET	FTR	Toimintopisteet
1	Asiakaskysely	EQ	5	1	3
2	Kysely asiakkaan tuotteista	EQ	7	2	4
3	Kysely paikoista	EQ	4	1	3
4	Kysely varastoiduista tuotteista	EQ	6	2	4
Yhteensä:					14

3.5 Sovittamattoman toimintopistemäärän ja sovitustekijän laskeminen

Projektin *sovittamaton toimintopistearvo* (Unadjusted Function Points, UFP) saadaan laskemalla tietotoimintojen ja tapahtumatoimintojen sovittamaton toimintopistemäärät yhteen. Näin ollen sovellusesimerkin (liite 1) sovittamaton toimintopistearvo lasketaan taulukoiden 3.3, 3.7, 3.10 ja 3.13 perusteella seuraavasti: UFP = 21 (ILF/EIF) + 31 (EI) + 14 (EO) + 14 (EQ) = 80.

Sovelluksen sovitustekijän (Value Adjustment Factor, VAF) avulla voidaan laskea projektin *sovitettu toimintopistemäärä* (Adjusted Function Points, AFP), kun tunnetaan projektin sovittamaton toimintopistemäärä. Sovitustekijän laskemiseksi pitää tuntea *kokonaisvaikutusaste* (Total Degree of Influence, TDI), joka saadaan laskemalla yhteen 14 *yleiselle systeemipiirteelle*[3] (general system characteristics, GSC) annetut arvot väliltä 0 - 5 (Garmus ja Herron, 2000).

Sovitustekijän VAF arviointi tapahtuu kolmessa vaiheessa (Garmus ja Herron, 2000):

1. Määritä jokaisen 14 systeemipiirteen arvo välille 0 - 5. Näin määritellään jokaisen systeemipiirteen *vaikutusaste* (Degree of Influence, DI).
2. Laske nämä 14 vaikutusastetta yhteen. Näin muodostuu kokonaisvaikutusaste (TDI).

[3] Yleiset systeemipiirteet on selitetty tarkasti esimerkein lähteissä Dreger (1989) sekä Garmus ja Herron (2000).

3. Laske sovelluksen sovitustekijä (VAF) TDI:tä apuna käyttäen seuraavasti:

$$VAF = (TDI \times 0.01) + 0.65 \quad (3.1)$$

Systeemipiirteiden yhteenlaskettu kokonaisvaikutusaste (TDI) voi olla välillä 0 - 70. Sovelluksen sovitustekijän laskentakaavassa (3.1) on kaksi vakiota 0.01 ja 0.65. Kokonaisvaikutusaste kerrotaan luvulla 0.01. Näin saatuun arvoon lisätään toinen vakioista eli 0.65, jonka jälkeen sovelluksen VAF:ksi saadaan luku väliltä 0.65 - 1.35. Sovelluksen sovitettu toimintopistemäärä saadaan kertomalla sovittamaton toimintopistemäärä VAF:lla, joten sovitustekijän vaikutusalue on välillä -35% - +35%.

3.6 Sovitetun toimintopistearvon laskennassa käytetyt kaavat

Kun sovelluksesta tehdään ensimmäisen versio, on kyseessä kehitysprojekti. Kehitysprojektille lasketaan toimintopistemäärä käyttäen seuraavaa kaavaa (Garmus ja Herron, 2000):

$$DFP = (UFP + CFP) * VAF, \quad (3.2)$$

jossa

- DFP on kehitysprojektin sovitettu toimintopistemäärä.
- UFP on kehitysprojektin sovittamaton toimintopistemäärä.
- CFP on tiedon konvertoinnista aiheutunut toimintopistemäärä. Konvertointia tapahtuu yleensä, kun olemassa olevista tiedostoista (esim. jonkin toisen sovelluksen käyttämistä tiedostoista) siirretään tietoa uuden kehitysprojektin sisäisiin loogisiin tiedostoihin. Nämä lasketaan yleensä ulkoisina syötteinä (EI). Tästä konvertoinnista saattaa myös aiheutua raportti (EO) siitä, kuinka tiedonsiirto on onnistunut.
- VAF on sovelluksen sovitustekijä.

Kun olemassa olevaa sovellusta muutetaan eli siihen lisätään toimintoja, sen toimintoja muutetaan tai siitä poistetaan toimintoja, on kysymyksessä ylläpitoprojekti (Garmus ja Herron, 2000). Ylläpitoprojektin toimintopistearvo lasketaan kaavan (3.3) mukaisesti:

$$EFP = [(ADD + CHGA + CFP]) * VAFA] + (DEL * VAFB), \quad (3.3)$$

jossa

- EFP on ylläpitoprojektin sovitettu toimintopistearvo.
- ADD on ylläpitoprojektin lisäämien toimintojen sovittamaton toimintopistearvo.
- CHGA on ylläpitoprojektin muuttamien toimintojen sovittamaton toimintopistearvo.
- CFP on tiedon konvertoinnista aiheutunut toimintopistemäärä.
- VAFA on sovelluksen sovitustekijä ylläpitoprojektin jälkeen.
- DEL on ylläpitoprojektin aikana poistettujen toimintojen sovittamaton toimintopistearvo.
- VAFB on sovelluksen sovitustekijä ennen ylläpitoprojektia.

Sovelluksen toimintopistearvo tulisi laskea seuraavissa tapauksissa (Garmus ja Herron, 2000):

1. Kun sovellus ensimmäisen kerran toimitetaan.
2. Kun ylläpitoprojekti on muuttanut sovelluksen toiminnallista arvoa. Ylläpitoprojekti on voinut muuttaa sovellusta:

    a) Lisäämällä siihen uutta toiminnallisuutta kasvattaen sovelluksen toimintopistearvoa.
    b) Muuttamalla sen toiminnallisuutta (tämä ei välttämättä muuta sovelluksen toimintopistearvoa).
    c) Poistamalla siitä toiminnallisuutta pienentäen sovelluksen toimintopistearvoa.
    d) Muuttamalla sovitustekijöiden arvoa (tällä ei välttämättä ole merkitystä sovelluksen toimintopistearvoon).

Sovelluksen sovitettu toimintopistearvo lasketaan kehitysprojektin tuottamalle sovellukselle kaavalla (3.4) (Garmus ja Herron, 2000):

$$AFP = ADD * VAF, \quad (3.4)$$

jossa

- AFP on sovelluksen sovitettu toimintopistearvo.

- ADD on sovelluksen asennettujen osien sovittamaton toimintopistearvo.
- VAF on sovelluksen sovitustekijä.

Sovelluksen toimintopistemäärä ylläpitoprojektin tuottamalle sovellukselle lasketaan kaavan (3.5) mukaisesti (Garmus ja Herron, 2000):

$$AFP = [(UFPB + ADD + CHGA) - (CHGB + DEL)] * VAFA, \quad (3.5)$$

jossa

- AFP on sovelluksen sovitettu toimintopistearvo.
- UFPB on sovelluksen sovittamaton toimintopistearvo ennen ylläpitoprojektia.
- ADD on ylläpitoprojektin lisäämien toimintojen sovittamaton toimintopistearvo.
- CHGA on ylläpitoprojektin muuttamien toimintojen sovittamaton toimintopistearvo muutoksen jälkeen.
- CHGB on ylläpitoprojektin muuttamien toimintojen sovittamaton toimintopistearvo ennen muutosta.
- VAFA on sovelluksen sovitustekijä ylläpitoprojektin jälkeen.
- DEL on ylläpitoprojektin poistamien toimintojen sovittamaton toimintopistearvo.

Kaavassa (3.5) kannattaa kiinnittää huomio muuttujiin CHGA sekä CHGB, joiden käytöstä ilmenee kaavan perusidea. Kun ylläpitoprojekti muuttaa tieto- tai tapahtumatoimintoja, lisätään niiden sovittamaton toimintopistearvo sovelluksen toimintopistearvoon. Jotta näitä muutettuja toimintoja ei laskettaisi kahteen kertaan, joudutaan muutettujen toimintojen vanha toimintopistearvo vähentämään sovelluksen toimintopistearvosta. Tämän lisäksi vähennetään tietenkin sovelluksesta poistettu toiminnallisuus (DEL).

3.7 Tutkimustuloksia toimintopisteanalyysistä

Toimintopisteanalyysin on todettu korreloivan ohjelmiston kehityksessä vaadittavan työmäärän kanssa (Abran ja Robillard, 1996; Low ja Jeffery, 1990). Monet ovat kuitenkin nähneet toimintopisteanalyysin ongelmana sen tulkinnanvaraisuuden. Toimintopistelaskentaa on myös arvosteltu mm. siitä, että sen antamia tuloksia käytetään väärin ja osa tuloksista myös laske-

taan väärin. Toimintopisteanalyysistä tehdyt tutkimukset sisältävät paljon kritiikkiä, mutta myös joitakin parannusehdotuksia.

3.7.1 Toimintopisteanalyysin ongelmakohdat

Kitchenhamin (1997) mukaan toimintopisteanalyysissä on perustavaa laatua olevia rakennevikoja. Sen vuoksi on olemassa vaara, että mittaukset antavat vääriä tuloksia. Esittämänsä kritiikin Kitchenham perustelee varsin järkeenkäyvästi. Toimintopisteanalyysissä absoluuttisen mitta-asteikon pisteet muunnetaan järjestysasteikon pistemääriksi "yksinkertainen", "keskinkertainen" ja "vaikea". Järjestysasteikon arvoja ei toisaalta voida laskea yhteen keskenään sillä esim. laskutoimitus "yksinkertainen" + "vaikea" ei ole mielekäs. Järjestysasteikon muuttujia ei voida myöskään jakaa tai kertoa, joten erilaiset tuottavuusmittarit ovat kyseenalaisia.

Kitchenham (1997) puuttuu myös voimakkaasti siihen seikkaan, että eri henkilöt saavat erilaisia toimintopistearvoja samoista (saman toiminnallisuuden omaavista) sovelluksista. Vaikka samassa yrityksessä työskentelevät henkilöt saisivatkin hyvin lähellä toisiaan olevia toimintopistearvoja, ei Kitchenhamin mukaan voida kuvitella, että yritysten saamia tuloksia voitaisiin vertailla keskenään. Samoista sovelluksista lasketut toimintopistemäärät ovat eronneet toisistaan 10 - 30 %. Kitchenham ei jätä myöskään yleisiä systeemipiirteitä kritiikittä. Nekin ovat järjestysasteikon arvoja, joten niitäkään ei voida lisätä toisiinsa. Ei ole myöskään selvästi osoitettu, että sovitustekijät antaisivat lisäarvoa toimintopisteanalyysiin.

Kitchenhamin mukaan on kuitenkin tarvetta mittareille, joiden avulla mittauksia voidaan suorittaa sovelluskehityksen aikaisessa vaiheessa. Keinona voisi olla toimintopistelaskennan yksinkertaistaminen niin, että toimintopistearvo saataisiin laskettua sovelluksesta automaattisesti. Tämä automatisoitu laskenta pitäisi pystyä tekemään hyvin aikaisessa vaiheessa tuotetuista kaavioista. Kitchenham pitää myös tärkeänä, ettei sovelluksista laskettuja toimintopisteitä laskettaisi yhteen keskenään. Hän ehdottaakin, että eri mittaustuloksia käytettäisiin erillisinä joukkoina arvoja, jotka kuvaisivat sovelluksen luonnetta. Kitchenham ei myöskään usko, että samat mittausmallit sopisivat sekä kehitysprojektien että ylläpitoprojektien kartoittamiseen.

3.7.2 Toimintopisteanalyysin yksinkertaistaminen

Lokan (1999) on tutkinut toimintoanalyysissä ilmenevää *korrelaatiota* eli riippuvuutta eri toi-

mintojen (EIF, ILF, EI, EO sekä EQ) välillä. Tunnistamalla erilaiset toimintopisteanalyysiin kuuluvat riippuvuudet voidaan kenties lyhentää ja yksinkertaistaa toimintopisteanalyysiprosessia.

Lokan jakoi aineistonsa alijoukkoihin ohjelmointikielen, projektin tyypin, organisaation tyypin ja protoilun perusteella. Näistä alijoukoista Lokan löysi seuraavat tilastollisesti merkitsevät riippuvuudet:

- Ne ohjelmistot, jotka on kehitetty käyttäen *kolmannen sukupolven ohjelmointikieliä* (3GL, esim. C-kieli ja Cobol) sisältävät enemmän tiedostoja ja vähemmän tapahtumia, kuin *neljännen sukupolven ohjelmointikielillä* (4GL, esim. SQL) tai sovelluskehittimillä tehdyt.
- Ylläpitoprojekteihin kuuluu enemmän tapahtumia ja vähemmän tiedostoja, kuin kehitysprojekteihin.
- Pankkien ja liikemaailman sovellukset käyttävät vähemmän syötteitä ja tuottavat enemmän tulosteita kuin julkishallinnon sovellukset.
- Jos kehitysprojektin aikana tehdään paljon prototyyppejä, viittaa se siihen, että sovellus tulee sisältämään myös paljon syötteitä ja kyselyjä, mutta vähemmän tulosteita.

Aineistossa, johon kuului 269 projektia, oli havaittavissa kauttaaltaan korrelaatio syötteiden ja sisäisten tiedostojen välillä. Usein syötteiden ja sisäisten tiedostojen välinen korrelaatio oli heikko, mutta myös vahvaa riippuvuutta oli havaittavissa etenkin 4GL:iä käytettäessä. Ilmeisesti tässä riippuvuudessa kyse ei kuitenkaan ole 4GL:stä sinänsä, vaan siitä, että tietyn tyyppisiin sovellusalueisiin valitaan käytettäväksi nimenomaan 4GL. Näillä sovellusalueilla saattaa olla mahdollista hyödyntää myös syötteiden ja sisäisten tiedostojen välistä riippuvuutta haluttaessa yksinkertaistaa toimintopisteanalyysiprosessia. Lokan huomauttaa keskittyneensä tutkimuksessaan ainoastaan sovellusten koon mittaamiseen eikä työmäärään liittyvien riippuvuussuhteiden tutkimiseen. Lokanin mielestä se voisi olla hyvä tutkimuksen kohde jatkossa.

Myös Horganin et al. (1998) mukaan on olemassa todisteita siitä, että yksinkertaiset laskentatavat toimivat aikaisen vaiheen laskelmissa yhtä hyvin kuin itse toimintopisteanalyysikin. Tutkimuksenaan he ehdottavat käytettäväksi ns. *raakaa toimintopistemäärää* (Raw Function Point count, RFP). RFP saadaan laskemalla yhteen ulkoisten syötteiden (EI), ulkoisten tulos-

teiden (EO), ulkoisten kyselyiden (EQ), sisäisten loogisten tiedostojen (ILF) sekä ulkoisten liitäntätiedostojen (EIF) lukumäärä.

Taulukossa 3.14 on esimerkki RFP- ja AFP-arvoja sisältävästä tietojoukosta. Tiedot oletetaan saaduksi siten, että ohjelmoija on tehnyt aikaisemmin viisi sovellusta ja laskenut niistä toimintopisteanalyysin tuottaman AFP-arvon, vastaavan RFP-arvon sekä näiden välisen suhteen.

Taulukko 3.14 Esimerkki RFP- ja toimintopistearvojen tietojoukosta (Horgan et al., 1998).

RFP	AFP	AFP/RFP
8	30	3.75
20	140	7.00
25	150	6.00
32	232	7.25
19	114	6.00

Horganin et al. (1998) esittelemä malli (Early E model) koostuu kahdesta vaiheesta:

1) Lasketaan yksittäinen painotettu vakio W kaavasta (3.6).

$$W = \frac{\sum_{i=1}^{i=n} \frac{AFPi}{RFPi}}{n} \qquad (3.6)$$

2) Lasketaan W:n ja RFP:n avulla arvio E sovelluksen toimintopistearvosta kaavan (3.7) avulla:

$$E = RFP * W \qquad (3.7)$$

Kaavaa (3.6) soveltamalla voidaan taulukosta 3.10 laskea vakio W:

$$W = \frac{3.75 + 7.00 + 6.00 + 7.25 + 6.00}{5} = 6,00$$

Horganin et al. (1998) esittämät tutkimustulokset ovat rohkaisevia. Tutkittavana oli kaksi havaintojoukkoa, joihin kuului 24 ja 21 projektin tiedot. Kumpikin havaintojoukko analysoitiin seuraavasti:

1) Valittiin taulukon 3.15 mukaisesti havaintojoukon projekteista viisi osajoukkoa (20-

25%). Jokainen osajoukko muodosti testiryhmän.

2) Näistä osajoukoista jokaisesta johdettiin painotettu vakio (W).
3) Vakion W avulla arvioitiin toimintopistearvo muista havaintojoukkojen projekteista.

Taulukko 3.15 Ensimmäisestä havaintojoukosta muodostetut viisi testiryhmää ja W-arvot (Horgan et al., 1998).

Testiryhmä	Projektin numero	RFP	AFP	AFP/RFP	W
1	1	310	1750	5.65	5.03
	2	397	1902	4.79	
	3	109	428	3.93	
	4	132	759	5.75	
2	1	310	1750	5.65	5.29
	4	132	759	5.75	
	7	53	205	3.87	
	10	135	794	5.88	
3	1	310	1750	5.65	5.25
	6	55	283	5.15	
	11	102	512	5.02	
	16	99	512	5.17	
4	1	310	1750	5.65	5.59
	8	54	289	5.35	
	16	99	512	5.17	
	24	42	260	6.19	
5	4	132	759	5.75	5.19
	14	138	682	4.94	
	22	140	694	4.96	
	23	39	199	5.10	

Taulukko 3.16 Arvioidun toimintopistearvon tarkkuus ensimmäisen havaintojoukon osalta (Horgan et al., 1998).

Testiryhmä	W	Arvion tarkkuus		
		10%	25%	50%
1	5.03	60%	85%	100%
2	5.29	60%	90%	100%
3	5.25	55%	85%	100%
4	5.59	35%	80%	95%
5	5.19	55%	80%	100%

Taulukossa 3.16 näkyy arvioiden tarkkuudet. Arviot laskettiin soveltamalla jokaisen testiryhmän (taulukko 3.15) W-arvoa kaavan (3.7) osoittamalla tavalla havaintojoukon muiden projektien RFP-arvoihin. Jokaista laskutoimitusta verrattiin kyseisen projektin AFP-arvoon ja laskettiin E:n prosentuaalinen virhemarginaali. Taulukosta 3.16 näkyy, että 80-90% sovelluksista pääsi 25%:n tarkkuuteen. Näin ollen 25%:n tarkkuusvaatimuksella voidaan sanoa, että havaintojoukolla yksi päästiin hyvään tulokseen (Horgan et al., 1998).

Toiselle havaintojoukolle suoritettiin samat toimenpiteet kuin ensimmäisellekin. Taulukossa 3.17 näkyy toisen havaintojoukon perusteella arvioidun AFP:n tarkkuus. Toisessa havaintojoukossa oli mukana myös kunkin projektin sovittamaton toimintopistearvo (UFP). Taulukossa 3.18 näkyy arvioidun toimintopistearvon tarkkuus, kun käytettiin RFP- ja UFP-arvoa.

Taulukko 3.17 Arvioidun toimintopistearvon tarkkuus toisen havaintojoukon osalta (Horgan et al., 1998).

Testiryhmä	W	Arvion tarkkuus		
		10%	25%	50%
1	6.76	35%	71%	94%
2	6.22	47%	82%	94%
3	6.25	53%	82%	94%
4	6.50	59%	82%	94%
5	6.12	47%	88%	94%

Taulukko 3.18 Arvioidun UFP-arvon tarkkuus toisen tietojoukon osalta (Horgan et al., 1998).

Testiryhmä	W	Arvion tarkkuus		
		10%	25%	50%
1	8.30	53%	94%	94%
2	7.61	71%	94%	94%
3	7.52	94%	100%	100%
4	7.82	71%	94%	94%
5	7.79	71%	94%	94%

Taulukon 3.18 tulokset 25% tarkkuudella ovat hyvät. Käytännössä näyttää siltä, että jokaisen sovelluksen kohdalla päästiin 25% tarkkuuteen. Horgan et al. pitävätkin tutkimustuloksiaan hyvinä, mutta pitävät tärkeänä saada lisäselvitystä eri toimintotyyppien välisistä riippuvuussuhteista. Tutkimukset osoittavat kuitenkin selvästi, että Early E -malli sopii mainiosti IFPUG:n toimintopisteanalyysin kanssa käytettäväksi (Horgan et al., 1998).

3.7.3 Yleisten systeemipiirteiden arviointia

Toimintopisteanalyysissa muodostetaan sovitettu toimintopistearvo neljäntoista yleisen systeemipiirteen avulla. Tämä vaihe toimintopisteanalyysissa on osoittautunut ristiriitaiseksi (Lokan, 2000). Lokan on tehnyt mittavan tutkimuksen neljäntoista yleisen systeemipiirteen hyödyllisyydestä toimintopistelaskennassa. Aineisto koostui 235:n projektin tiedoista. Jokaisesta projektista oli käytettävissä kehitykseen käytetty työmäärä sekä kaikkien neljäntoista systeemipiirteen arvot. Käytettävissä oli myös tiedot sovelluksen tyypistä, kehitysalustasta, ohjelmointikielestä ja toteutuspäivämäärästä.

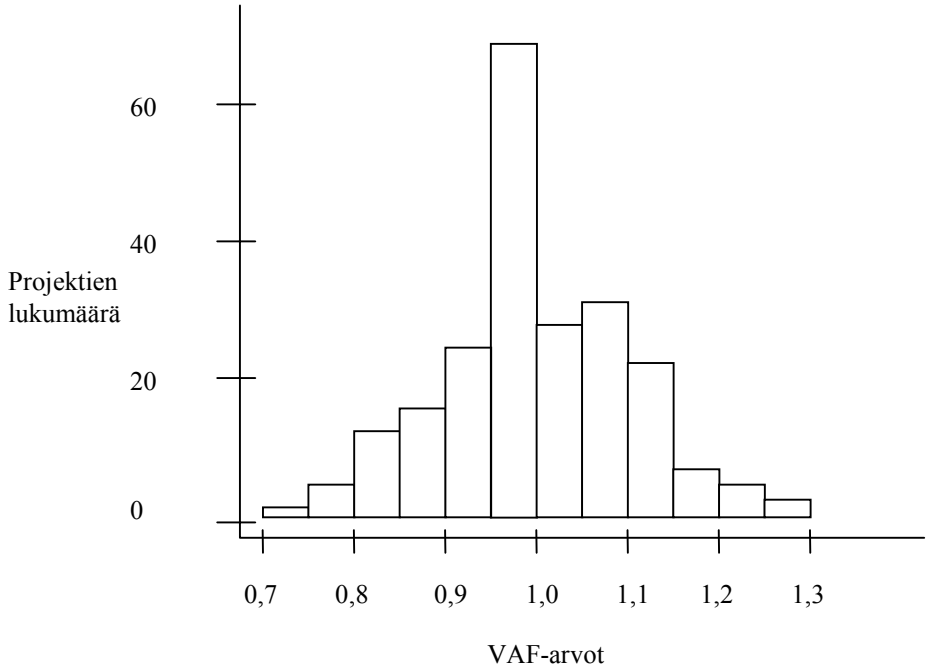
Taulukko 3.19 Systeemipiirteiden prosenttijakaumat Lokanin (2000) tutkimuksessa.

Systeemipiirteet	0	1	2	3	4	5
01 Tietoliikenne	3	0	7	34	27	29
02 Hajautettu käsittely	33	9	24	12	19	3
03 Suorituskyky	6	15	11	30	26	12
04 Raskaasti käytetty kokoonpano	9	23	34	20	5	9
05 Tapahtumatiheys	20	11	13	29	23	5
06 Suora tiedon syöttö	3	2	8	26	1	60
07 Loppukäyttäjän työteho	6	7	21	23	27	15
08 Suora päivitys	3	5	10	32	37	13
09 Monimutkainen käsittely	9	15	20	30	23	2
10 Uudelleen käytettävyys	15	17	27	26	10	4
11 Asentamisen helppous	20	11	23	24	7	15
12 Käytön helppous	17	11	34	23	8	6
13 Monipaikkaisuus	33	11	24	16	11	5
14 Muutosten teon helpottaminen	29	7	17	27	10	10

Taulukossa 3.19 näkyvistä systeemipiirteistä tietoliikenne (01) ja suora tiedon syöttö (06) pitäisi Lokanin mielestä arvioida uudelleen, koska näiden systeemipiirteiden arvot ovat poikkeuksellisen korkeita. Mitä ennen pidettiin poikkeuksellisena, onkin tänä päivänä normaalia. Pienimmät arvot ovat hajautetulla käsittelyllä (02), monipaikkaisuudella (13), asentamisen helppoudella (11), käytön helppoudella (12) sekä uudelleen käytettävyydellä (10). Hajautetun käsittelyn ja monipaikkaisuuden arvot tulevat Internetin voimakkaan kasvun myötä ilmeisesti kasvamaan.

Lokan on myös havainnut systeemipiirteiden arvojen vaihtelevan ymmärrettävästi riippuen sovellustyypistä. Esimerkiksi suorituskyvyllä on suuri merkitys sovelluksissa, jotka joutuvat käsittelemään paljon tapahtumia. Tietoliikennevaatimukset ovat vähäiset henkilökohtaisiin tietokoneisiin kehitetyissä järjestelmissä. Vähiten riippuvaisia sovellustyypistä olivat seuraavat viisi systeemipiirrettä: raskaasti käytetty kokoonpano, loppukäyttäjän työteho, asentamisen ja käytön helppous sekä monipaikkaisuus. Riippuvuussuhteiden puute Lokanin mielestä kertoo siitä, että näiden viiden systeemipiirteen vaihtelun tulkitseminen voisi olla hyödyllistä.

Lokan (2000) tutki myös VAF:n vaikutusta sovittamattomaan toimintopistearvoon. Kuvassa 3.3 näkyy VAF-arvojen jakauma. VAF-arvoista yli puolet muuttivat sovittamatonta pistemäärää vain noin viidellä prosentilla. Maksimiarvo oli 1.29 ja minimi puolestaan 0.73. Taulukossa 3.20 näkyy yhteenveto VAF-arvoista.



Kuva 3.3 VAF-arvojen jakauma (Lokan, 2000).

Taulukko 3.20 VAF-arvojen yhteenveto (Lokan, 2000).

Minimiarvo	Ensimmäinen kvartiili	Mediaani	Kolmas kvartiili	Maksimiarvo	Keskiarvo	Keskihajonta
0.73	0.96	1.00	1.07	1.29	1.01	0.10

Saadakseen selville, tuottaako VAF oikeanlaista hienosäätöä sovittamattomaan toimintopistearvoon, Lokan vertasi sovittamatonta sovitettuun toimintopistearvoon projektin työmäärän kannalta. Projekteja oli kaiken kaikkiaan 235, joista 108:ssa VAF antoi viitteitä oikeaan suuntaan. Jos esimerkiksi sovittamaton toimintopistemäärä aliarvioi ohjelmiston vaatiman työmäärän ja VAF-arvo oli yli 1.00, niin VAF teki korjauksen oikeaan suuntaan. Kaikkiaan 73 projektissa VAF antoi kuitenkin viitteitä väärään suuntaan. Lopuissa 54 projekteista VAF oli 1.00 eikä näin ollen aiheuttanut muutoksia sovittamattomaan toimintopistearvoon. Tutkimustulokset osoittavat Lokanin mielestä, ettei sovitetun toimintopistearvon laskemisesta ole hyötyä projektin työmäärän arvioinnissa.

Lokan (2000) ehdottaakin, että VAF:n ja sovittamaton toimintopistearvon välinen suhde tulisi arvioida uudelleen. Kaikkia neljäätoista systeemipiirrettä voitaisiin esimerkiksi käyttää erikseen arvioimaan sovelluksen luonnetta. Systeemipiirteiden avulla voitaisiin esimerkiksi tunnistaa tietyn tyyppiset ohjelmistot ja tehdä vertailua näiden välillä. Myöskin Symons (2001) on sitä mieltä, että FPA:n tapa laskea sovellukseen liittyviä teknisiä tai laadullisia vaatimuksia

systeemipiirteiden avulla ei ole paras mahdollinen tapa. Symons sanookin, että systeemipiirteiden avulla voidaan ottaa huomioon kustannuksia ja työmäärää, mutta ne eivät vaikuta sovelluksen kokoon.

3.8 Mallinnuskielen tuki olioperustaisten sovellusten toimintopistelaskennassa

Koska toimintopisteanalyysi on riippumaton käytetystä teknologiasta, antaa se mahdollisuuden mitata esimerkiksi olioperustaisia järjestelmiä (Garmus ja Herron, 2000). Toimintopisteanalyysi sisältää monin paikoin kuitenkin tulkinnanvaraisuutta. Esitän sen vuoksi Uemuran et al. (1999) kehittämän toimintopistelaskentamallin, joka perustuu UML-mallinnuskieleen. Mallin idea ja systemaattisuus selviävät mielestäni hyvin kohdassa 3.8.2 esitettävien sekvenssikaavioiden avulla.

Säännöt UML-notaation perusteella tapahtuvaan kehitysprojektin sovittamattoman toimintopistemäärän laskentaan ovat seuraavat (Uemura et al., 1999):

1. Määrittele laskenta-alueen raja. Laskenta-alueen raja määritellään *sekvenssikaaviossa* esiintyvien olioiden tyypin perusteella. *Toimijaoliot* (actor objects) ovat sovellusrajan ulkopuolella ja muut oliot sisäpuolella.
2. Määrittele tietotoiminnot. Tietotoiminnot määritellään automaattisesti luokka- ja sekvenssikaavioiden pohjalta.
3. Määrittele tapahtumatoiminnot. Tapahtumatoiminnot määritellään automaattisesti perustuen luokka- ja sekvenssikaavioihin.
4. Määrittele sovittamaton toimintopistemäärä. Sovittamaton toimintopistemäärä voidaan laskea automaattisesti askelten 2 ja 3 tulosten perusteella.

3.8.1 Tietotoimintojen määrittäminen

Seuraavien sääntöjen avulla voidaan sekvenssi- ja luokkakaavioita apuna käyttäen määritellä sovellukseen kuuluvat tietotoiminnot. Sääntöjen yhteydessä oliot luokitellaan toimijaolioiksi sekä *ei-toimijaolioiksi*. Toimijaoliot sijaitsevat sovellusalueen ulkopuolella eikä niitä lasketa tietotoiminnoiksi (Uemura et al., 1999).

1. Valitse tietotoimintoehdokkaat: Ehdokkaiksi valitaan ne oliot, joilla on omia attribuut-

teja sekä tiedonvaihtoa ei-toimijaolion kanssa.

2. Määritä tietotoimintotyyppi: Ensimmäisessä vaiheessa valituista olioista ne, joilla on operaatioita, jotka muuttavat jonkin toisen olion attribuuttien arvoja, ovat sisäisiä loogisia tiedostoja (ILF). Loput oliot ovat ulkoisia liitäntätiedostoja (EIF).
3. Määritä tietotoimintojen monimutkaisuus: Luokan attribuuttien määrä muodostaa tietoalkiotyyppien lukumäärän (DET). Jos luokka periytyy jostain toisesta luokasta, myös perittävän luokan attribuutit lasketaan. Aikaisemman kokemuksen perusteella voidaan todeta, että tietuealkiotyyppien lukumäärä (RET) on lähes aina yksi. Näin ollen määrittelemme tietuealkiotyyppien lukumäärän aina yhdeksi. Monimutkaisuus lasketaan, kuten muutenkin toimintopisteanalyysissä, käyttäen apuna monimutkaisuustaulukkoa.

3.8.2 Tapahtumatoimintojen määrittäminen

Jokaisen edellä hyväksytyn tietotoiminnon lähettämä viesti sekvenssikaaviossa on mahdollinen tapahtumatoiminto (EI, EO tai EQ). Jos viestiin ei sisälly yhtäkään argumenttia, ei viesti voi olla myöskään tapahtumatoiminto. Jotta tapahtumatoiminnot voidaan määritellä luokka- ja sekvenssikaavion avulla, ovat seuraavat rajoitukset voimassa (Ueamura et al., 1999):

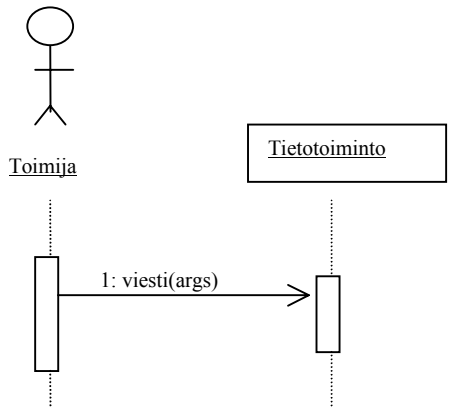
- Kun olio lähettää toiselle oliolle viestin ja vastaanottajaolio palauttaa merkityksellisen viestin, täytyy tämä vastaus kuvata täsmällisen tarkasti sekvenssikaaviossa[4].
- Olioiden välinen tiedonvaihto on kuvattava sekvenssikaaviossa viesteihin kuuluvina argumentteina.
- Jos viesti sisältää argumentin, joka on saman niminen kuin lähettäjäolion jokin attribuutti, oletetaan sen olevan nimenomaan tämä lähettäjäolion attribuutti.
- Jos sekvenssikaaviossa oleva viesti esiintyy toistuvasti, on viestiin kuuluvat argumentit ja viestin nimi kuvattava samalla tavalla joka kerta.

Jokaista toimijaoliota kohden sekvenssikaaviossa tarvitaan kaksi seuraavaa vaihetta tapahtumatoimintojen tunnistamiseksi (Uemura et al., 2000):

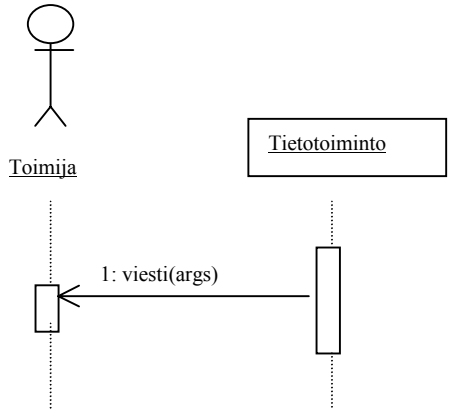
[4] UML ei vaadi tämän vastaanottajaolion lähettämän vastauksen tarkkaa kuvausta.

1. Valitse tapahtumatoimintoehdokkaat: Listaa ne viestisarjat, joissa ensimmäisen viestin lähettäjä on toimijaolio (usein esimerkiksi käyttäjä) ja sekvenssikaaviossa viimeisen viestin vastaanottaja on joko toimija- tai ei-toimijaolio.
2. Määrittele tapahtumatoiminnon tyyppi: jokaisen ensimmäisessä vaiheessa määritellyn tapahtumatoiminnon tyyppi ja monimutkaisuus (DET ja FTR) määritellään kuvien 3.4 - 3.8 avulla.

Kuvassa 3.4 toimija lähettää viestin tietotoiminnolle, jolloin kyseessä on ulkoinen syöte (EI). Jos viestiin ei kuulu yhtään argumenttia, kyseessä ei ole EI. DET on viestissä esiintyvien argumenttien lukumäärä. FTR on 1, koska kaavioon liittyy vain yksi tietotoiminto (Uemura et al., 1999).



Kuva 3.4 Toimija lähettää viestin tietotoiminnolle (Uemura et al., 1999).

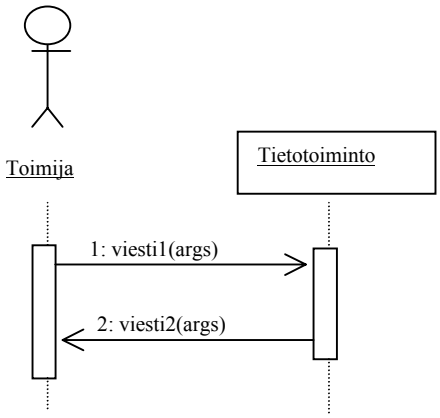


Kuva 3.5 Tietotoiminto lähettää viestin toimijalle (Uemura et al., 1999).

Kuvassa 3.5 tietotoiminto lähettää viestin toimijalle. Jos jokainen viestissä oleva argumentti on jokin tietotoiminnon attribuuteista, on kyseessä ulkoinen kysely (EQ). Jos jotain argumen-

teista ei löydy tietotoiminnon (luokan/olion) attribuuteista, on kyseessä oltava lasket-tua/johdettua tietoa ja näin ollen kyseessä on ulkoinen tuloste (EO). DET on sama kuin argumenttien lukumäärä ja FTR on yksi, koska tapahtumaan kuuluu vain yksi tietotoiminto.

Kolmas malli kuvassa 3.6 edustaa tilannetta, jossa toimija lähettää viestin tietotoiminnolle ja tietotoiminto palauttaa viestin toimijalle. Jos kaikki viestin 2 argumentit ovat samoja kuin tietotoiminnossa olevat attribuutit, on kyseessä EQ. Jos taas jotakin viestin 2 argumenteista ei löydy tietotoiminnon attribuuteista, on kyseessä EO. Viesti 1 on tapahtuman syötte. DET on viestissä 2 lueteltujen argumenttien lukumäärä.

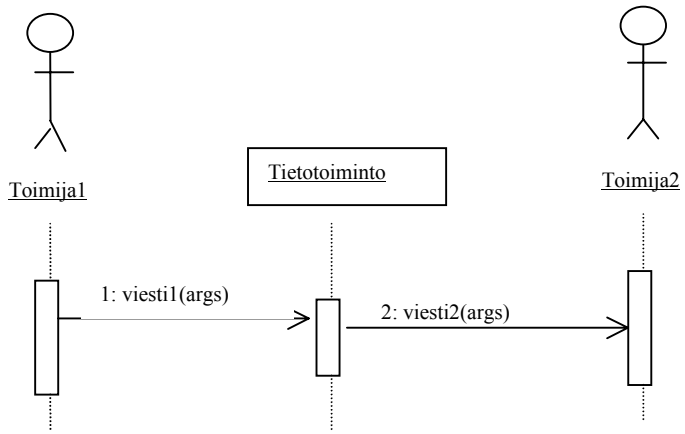


Kuva 3.6 Toimija lähettää viestin tietotoiminnolle, joka lähettää sen takaisin toimijalle (Uemura et al., 1999).

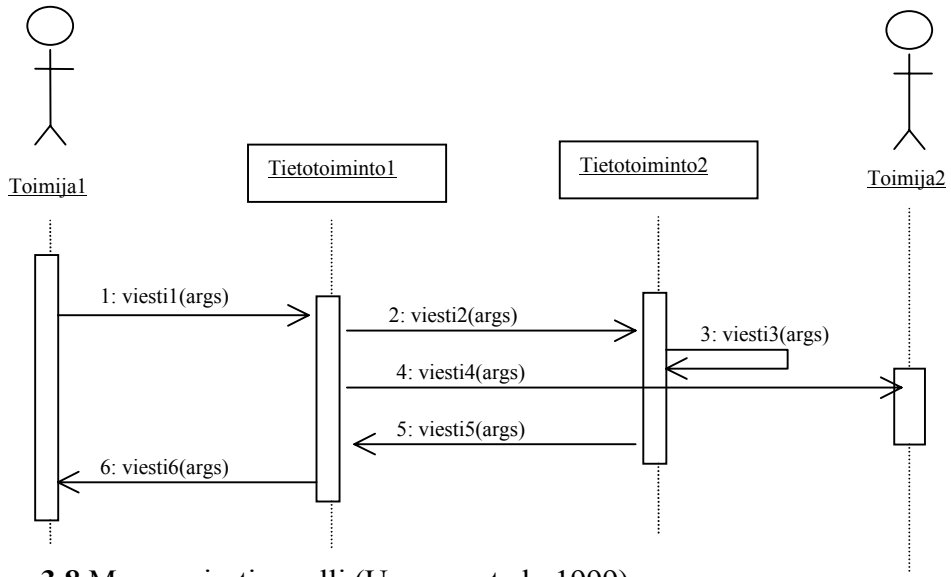
Kuvan 3.7 mallissa toimija lähettää viestin tietotoiminnolle, joka lähettää viestin toiselle toimijalle. Tämä malli jaetaan kahteen osaan, joka koostuu kuvissa 3.4 ja 3.5 olevista malleista.

Viidennessä mallissa (kuva 3.8) toimija lähettää viestin tietotoiminnolle ja lopulta toimija vastaanottaa vastausviestin monen eri tietotoiminnon kautta. Kuvan viestisekvenssi (1, 2, 3, 5, 6) muodostaa yhtenäisen tapahtumatoiminnon. Viestit 1 ja 6 vastaavat kuvan 3.6 mallia, jolloin DET on viestissä 6 lueteltujen argumenttien lukumäärä. Viitattujen tiedostotyyppien lukumäärä (FTR) on viestisarjaan kuuluvien tietotoimintojen lukumäärä eli tässä tapauksessa 2. Viesti 4 on oma tapahtumatoimintonsa, jossa FTR on yksi ja DET on viestin 4 argumenttien lukumäärä. Kuvan 3.8 sekvenssikaavio saattaa näyttää ensi alkuun hieman sekavalta, mutta se ei ole sitä. Sekvenssikaavio etenee ajallisesti ylhäältä alaspäin. Kaavion seuraaminen aloitetaan viestistä 1, jonka ansiosta kontrolli siirtyy tietotoiminto1:lle. Tämä lähettää viestin 2 tietotoiminto2:lle. Tietotoiminto2:lla kontrolli säilyy aina viestin 5 lähettämiseen asti. Lopulta toimi-

ja1 vastaanottaa viestin 6 tietotoiminto1:n lähettämänä. Viesti 4 jää edellä mainitussa seurannassa väliin ja se vastaa kuvan 3.5 mallia.



Kuva 3.7 Toimija lähettää viestin tietotoiminnolle, joka lähettää viestin toiselle toimijalle (Uemura et al., 1999).



Kuva 3.8 Monen viestin malli (Uemura et al., 1999).

Sekvenssikaavion piirtäminen (varsinkaan Uemuran et al. (1999) vaatimalla tarkkuudella) ei ole aivan ensimmäinen toimenpide ohjelmistoprojektissa. Uemura et al. ovat kuitenkin sitä mieltä, että etenkin kokeneet ohjelmoijat pystyvät käyttötapauskaavion avulla laatimaan luokka- ja sekvenssikaavion helposti.

Uemura et al. ryhmineen on tehnyt tietokoneohjelman, joka laskee luokka- ja sekvenssikaavioiden avulla sovelluksen sovittamattoman toimintopistearvon. Mallin toimivuutta on myös testattu, tosin vain yhden ainoan sovelluksen yhteydessä, jossa tämän mallin ja toimintopiste-

laskentaan perehtyneen spesialistin antamat tulokset eivät juuri eroa toisistaan. Uemura et al. (1999) käsittelevät tätä testausta ja aineistoa tutkimuksessaan kovin vähäisesti, vaikka pitävätkin testauksen antamia tuloksia rohkaisevina.

4. COSMIC-FFP

Toimintopisteanalyysi (FPA) on kehitetty alun perin jo 1970-luvulla. FPA:ta on kehitetty vuosien kuluessa vastaamaan uusimpia vaatimuksia. Vanhan mallin jatkuva kehittäminen kuitenkin usein johtaa siihen, että siitä tulee turhan monimutkainen (Symons, 2001). Symonsin mielestä on myös selvää, että tämän päivän ohjelmistoja nykyaikaisine termeineen on vaikea sovittaa yhteen 20 vuotta vanhan IFPUG-käsitteistön kanssa. Nykyisten sovellusten koon arvioinnissa joudutaankin usein turvautumaan toisenlaisiin menetelmiin. Monet asianharrastajat kokevat, että 1990-luvun lopulla alkunsa saanut COSMIC-FFP on selkeä parannus toimintopistelaskentaan.

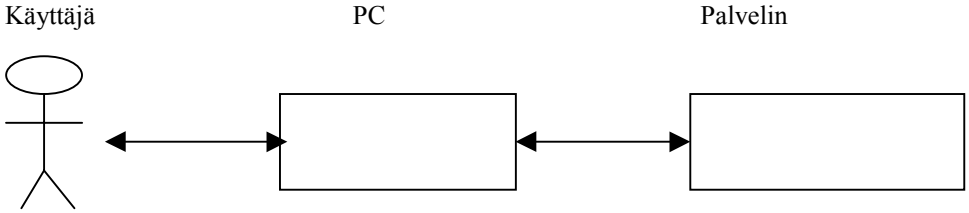
4.1 Toimintopistelaskennan kehittämistarve

Mikään menetelmä sovelluksen koon mittaamisessa ei tule suosituksi, ellei sen suorittaminen ole yksiselitteistä (Symons, 2001). Yksiselitteisyyden puute aiheuttaa sen, että mitattaessa saman toiminnallisuuden omaavan sovelluksen kokoa saadaan selvästi eri tuloksia mittaajien välillä. Esimerkkinä voisi mainita FPA:ssa käytetyn sisäisen loogisen tiedoston (ILF) käsitteen, joka on melko vaikea määritellä käytännössä.

Symonsin (2001) mielestä parannusta mittaamismenetelmiin tarvitaan lisäksi toimialariippumattomuuden saavuttamiseksi. Toiminnallisuuden mittaamisen tulee onnistua yhtä helposti *MIS-sovelluksista* (Management Information Systems) kuin *reaaliaikaisista* (realtime) sovelluksista. Tyypillisimmillään MIS-sovellukset ovat tietokantapohjaisia ohjelmistoja, kuten liitteessä 1 oleva esimerkkisovellus.

MIS-sovellukset ovat enemmän "tietopitoisia", kun taas uudemmat reaaliaikaiset sovellukset ovat "kontrollipitoisia" (Abran et al., 2001c). Tosin tänä päivänä näiden kahden sovellustyypin välillä on hieman vaikeampi tehdä eroa. MIS-sovellukset saavat usein syötteensä reaaliaikaisilta sovelluksilta, kuten tietoliikennesovelluksilta. Toisaalta tietoliikenne- ja prosessinohjaussovellukset käyttävät MIS-sovelluksille tyypillisiä tietokantoja. Sekä MIS- että reaaliaikaiset sovellukset rakennetaan nykyään niin, että ne koostuvat useasta eri *kerroksesta* (layer). Puhutaankin ns. monikerrosarkkitehtuurista, joka on tähän asti tuottanut ongelmia sovelluksen koon arvioinnissa. Garmus ja Herron (2000) tosin tulkitsevat IFPUG:n toimintopisteanalyysia myös monikerroarkkitehtuurissa.

Kun Albrecht aikanaan kehitti toimintopisteanalyysin, hän ymmärsi, että sovelluksen koko pitäisi mitata käyttäjän näkökulmasta. Käyttäjän näkökulmasta katsottuna monikerrosarkkitehtuurit ovat kuitenkin ongelma. Nykyään sovelluksen koon mittaamisessa haluttaisiinkin erottaa toisistaan käyttäjälle näkyvä ja ohjelmiston kehittäjälle näkyvä toiminnallisuus (kuva 4.1). Käyttäjälle on sama asia onko sovellus kokonaisuudessaan sijoitettu PC:lle vai onko siitä osia sijoitettu palvelimelle (esim. tietokanta tai sovelluksen toiminnallisuus), kunhan ohjelma tekee sen minkä käyttäjä haluaa.



Kuva 4.1 Kaksikerrosarkitehtuuri (PC/Palvelin).

Se mitataanko sovellusta käyttäjän vai kehittäjän näkökulmasta, vaikuttaa tietenkin mittaustuloksiin. Jos halutaan tarkkoja mittaustuloksia, etenkin jos sovelluksen eri osat on kehitetty eri teknologiaa käyttämällä, on syytä mitata esimerkiksi kuvan 4.1 tapauksessa toiminnallisuus erikseen sekä PC:ltä että palvelimelta.

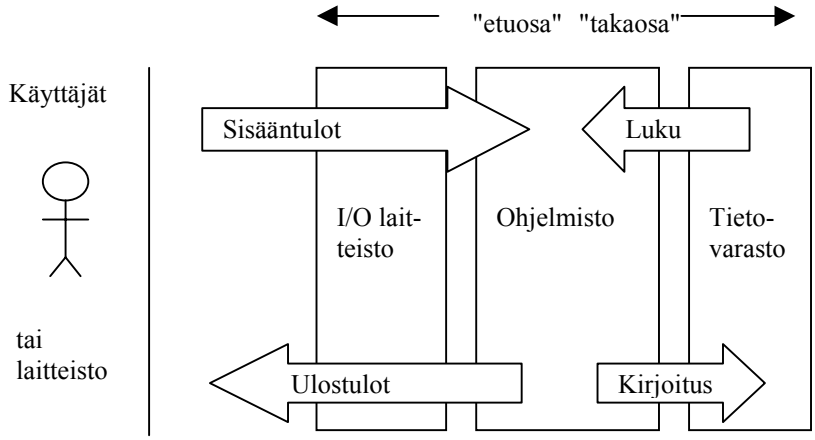
Mitattaessa kerrokset erikseen saattaa käydä niin, että sovelluksen toimittaneen tiimin teho näyttää laskeneen, kun ajatellaan vain käyttäjän näkökulmaa (Symons, 2001). Toisin sanoen kun jokainen kerros mitataan erikseen, käy usein niin, että sovelluksen koko näyttää kasvaneen. Toisaalta käyttäjälle näkyvä toiminnallisuus pysyy samana, joten aikaa toimintopistettä kohden kuluu enemmän.

COSMIC (The Common Software Measurement International Consortium) perustettiin vuonna 1998 tuottamaan uuden sukupolven väline sovellusten koon mittaamiseen (Abran, 2001b). COSMIC-FFP -menetelmä (ks. myös kuva 1.1) soveltuu sekä ns. perinteisempien (MIS) että uusien reaaliaikaisten sovellusten koon arviointiin. Menetelmä sopii myös molempien ohjelmistotyyppien yhdisteiden arviointiin[5].

[5] COSMIC-FFP versio 2.1 ei kuitenkaan huomioi matemaattisten kaavojen monimutkaisuutta eikä jatkuvia muuttujia, kuten musiikin tai filmin esittämistä. COSMIC-FFP suositteleekin matemaattisten kaavojen ja jatkuvien muuttujien yhteydessä käyttämään *paikallisia laajennuksia* (local extensions).

4.2 Mittauksen yleinen malli

COSMIC-FFP jakautuu kahteen vaiheeseen: vaatimusten *kuvausvaihe* (mapping phase) sekä *mittaamisvaihe* (measurement phase). Kuvausvaiheessa pyritään sovelluksesta muodostamaan erityinen käyttäjän toiminnallisia vaatimuksia (Functional User Requirements, FUR) vastaava ohjelmistomalli (COSMIC-FFP software FUR model), josta selviää sovelluksen toiminnallinen rakenne mittausta varten. Mittaamisvaiheessa ohjelmistomallia käytetään mittauksen suorittamiseksi. Vaiheiden tuloksena syntyy sovelluksen koon ilmoittava numeerinen arvo (Abran et al., 2001b).

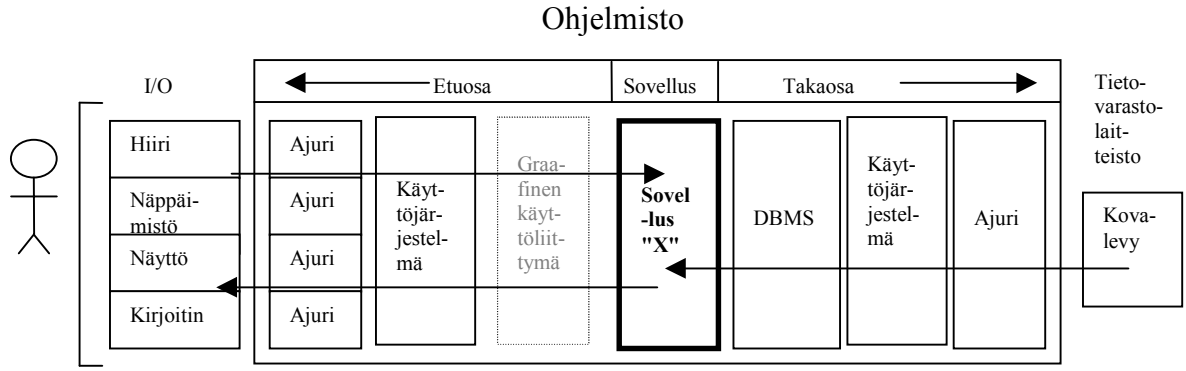


Kuva 4.2 Tiedon siirtäminen ohjelmistossa toiminnallisesta näkökulmasta (Abran et al., 2001b)

Kuvasta 4.2 selviää, mistä COSMIC-FFP:ssä pohjimmiltaan on kysymys. Keskeisenä on *tiedon siirtäminen* (data movement), joka jaetaan neljään aliprosessiin: *sisääntulo* (entry), *ulostulo* (exit), *kirjoitus* (write) ja *luku* (read). Lyhyesti sanottuna "etuosassa" tapahtuvien sisääntulojen ja ulostulojen avulla käyttäjä voi vaihtaa tietoa ohjelmiston kanssa. "Takaosassa" tapahtuvien kirjoittamisen ja lukemisen avulla tapahtuu tiedon siirtäminen ohjelmiston ja tietovarastojen välillä. Kukin tiedon siirtäminen sisältää täsmälleen yhden *tietoryhmän* (data group).

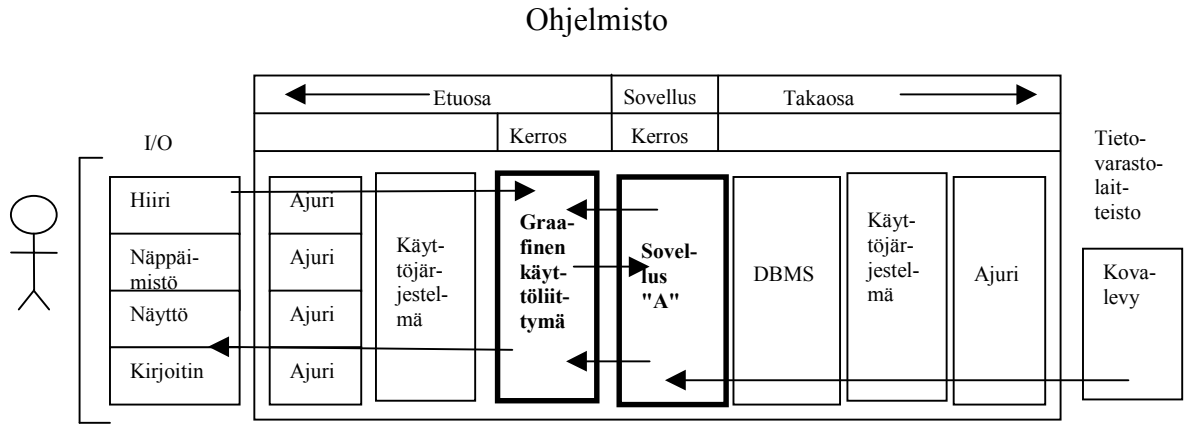
Kuvissa 4.3 ja 4.4 näkyy kuvan 4.2 tilanne yksityiskohtaisemmin. Niissä on kuvattu tyypillinen MIS-sovellus sekä "*moni-osa*"-sovellus (multi-piece). Kuvat selvittävät myös COSMIC-FFP:ssä tärkeää kerroksen (layer) käsitettä. Käsite "moni-osa" ei ole sama asia kuin monikerrosarkkitehtuuri, joskin käytännössä ne ovat hyvin lähellä toisiaan (Abran et al., 2001b).

Kuvan 4.3 mallissa käyttäjän toiminnalliset vaatimukset on kohdennettu yhdelle sovellustasolle. Tällöin on helppoa mitata sovelluksen "X" rajojen sisäpuolella sijaitseva toiminnallisuus. Kuvan 4.4 esimerkin toiminnallisuudesta osa sijaitsee sovelluksen "A" sisäpuolella ja osa erikseen toteutetussa graafisessa käyttöliittymässä. Sovelluksen ja graafisen käyttöliittymän välillä on riippuvuus, joka tosin ei näy käyttäjälle. Käyttäjälle näkyy vain sovelluksen A tarjoama toiminnallisuus.



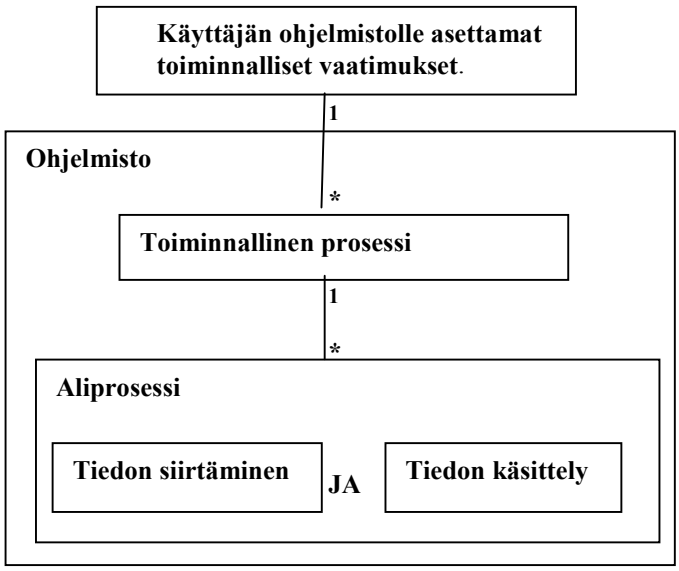
Kuva 4.3 Esimerkki toiminnallisten vaatimusten sijainnista MIS-sovelluksen kohdalla (Abran et al., 2001b).

Kuvassa 4.4 molemmilla kerroksilla eli graafisella käyttöliittymällä sekä sovelluksella "A" on oma raja (boundary) ja näin ollen molempien kerroksien toiminnallisuus voidaan laskea erikseen.



Kuva 4.4 Esimerkki toiminnallisten vaatimusten sijoittumisesta "moni-osa" -sovelluksessa (Abran et al., 2001b).

Sovelluksen koko on siis mahdollista mitata joko kuvan 4.3 tai 4.4 esimerkin tavalla. Kysymys on siitä, ollaanko kiinnostuneita sovelluksen toiminnallisuudesta kokonaisuudessaan vai jokaisesta sen kerroksesta erikseen (Abran et al., 2001b).



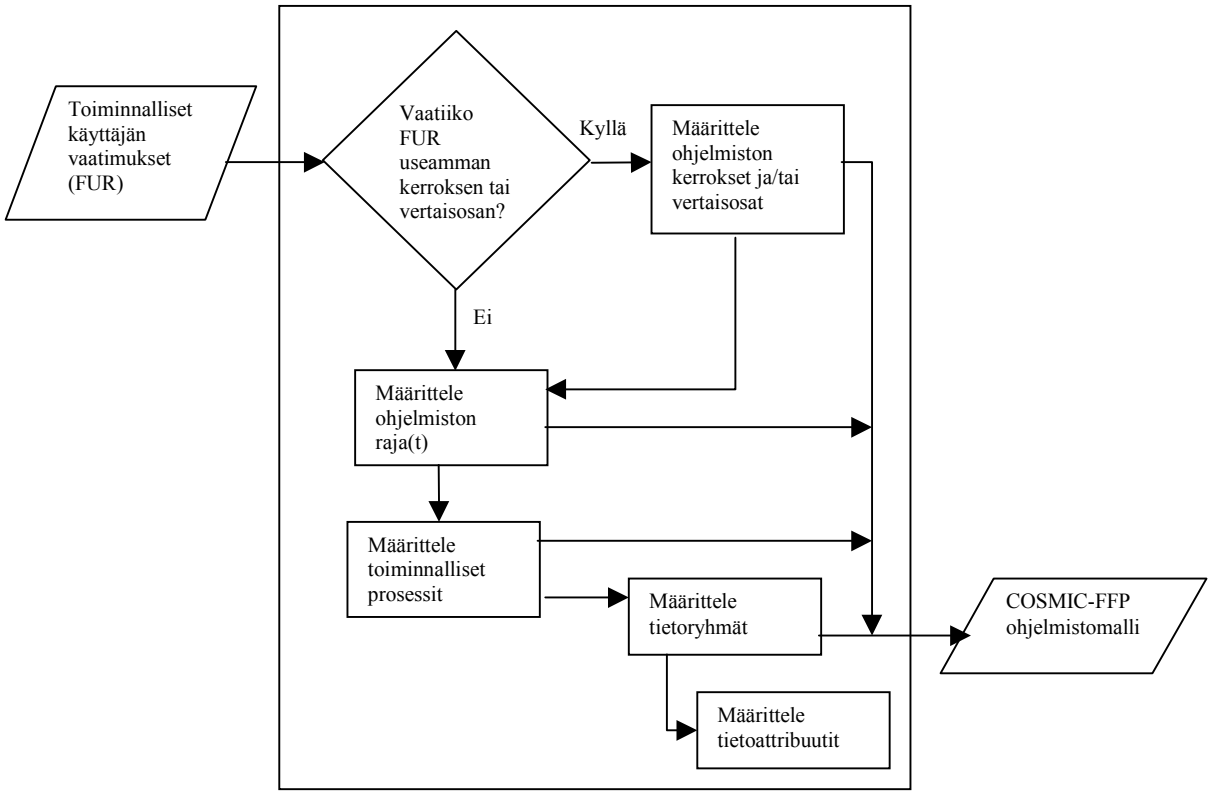
Kuva 4.5 Ohjelmiston malli toiminnallisen koon mittaamiseksi (Abran et al., 2001b).

Kuvassa 4.5 kiteytyy COSMIC-FFP:n idea. Käyttäjä asettaa sovellukselle toiminnallisia vaatimuksia ja joukko toiminnallisia prosesseja toteuttaa nämä vaatimukset. Toiminnalliset prosessit puolestaan koostuvat yksilöllisestä joukosta aliprosesseja, joihin sisältyy tiedon siirtämistä ja tiedon käsittelyä (data manipulation)[6]. Toiminnallinen prosessi sisältää vähintään kaksi tiedon siirtämistä: sisääntulon sekä poistumisen tai kirjoituksen.

4.3 Kuvausvaihe

Kuvassa 4.6 on esitetty kuvausvaihe yleisellä tasolla. Tämän vaiheen tuloksena syntyy *COSMIC-FFP ohjelmistomalli*, josta voidaan mittausvaiheessa tuottaa sovelluksen koon ilmaiseva numeerinen arvo.

[6] Tiedon käsittelyä ei COSMIC-FFP:n versiossa 2.1 oteta huomioon toiminnallista kokoa laskettaessa, joten aliprosessi rajautuu tiedon siirtämiseen eli tiedon siirtämistä voidaan pitää aliprosessina.



Kuva 4.6 COSMIC-FFP:n kuvausvaihe yleisellä tasolla (Abran et al., 2001b).

4.3.1 Kerrosten määrittäminen

Kerros on ohjelmistoympäristön toiminnallisen osittamisen tulos niin, että jokainen toiminnallinen prosessi toimii samalla abstraktiotasolla. Monikerrosarkkitehtuureissa kerrokset vaihtavat tietoa keskenään. Tämä tiedonvaihto on hierarkkista luonteeltaan siten, että kerros on toisen kerroksen "asiakas". Asiakas käyttää alemmalla tasolla olevan kerroksen palveluja. Määriteltäessä kerroksia puhutaan myös ns. *vertaisosista* (peer-to-peer), jolloin tietoa keskenään vaihtavat osat sijaitsevat samassa kerroksessa (Abran et al., 2001b).

Kerrosten määrittäminen on iteratiivinen prosessi. Kun kaikki kerrokset sovelluksessa ovat määritellyt, tulee niiden täyttää seuraavat periaatteet (Abran et al., 2001b):

1. Ohjelmiston jokainen kerros tuottaa toiminnallisuutta käyttäjilleen (käyttäjä voi olla ihminen, fyysinen laite tai jokin toinen sovellus).
2. Alemman tason kerros tuottaa palveluja asiakaskerrokselleen.
3. Alemman tason kerros voi toimia ilman asiakaskerroksen apua.
4. Asiakaskerros ei voi toimia oikein, jos alemman tason kerros ei toimi oikein.

5. Asiakaskerroksen ei välttämättä tarvitse käyttää kaikkia alemman kerroksen palveluja.
6. Alemman tason kerros voi olla jonkin alemman kerroksen asiakaskerros.
7. Asiakas- ja alemman tason kerrokset voivat jakaa ja vaihtaa samaa tietoa, mutta kukin kerros tulkitsee tietoa eri tavalla.
8. Ohjelmistoja, jotka käyttävät samaa tietoa ja tulkitsevat sen samalla tavalla, ei voida pitää erillisinä kerroksina.

Kerroksia määriteltäessä ymmärretään erillisiksi kerroksiksi yleensä toiminnalliset "palvelupaketit", kuten tietokannanhallintajärjestelmät, graafiset käyttöliittymät, käyttöjärjestelmät tai ohjelmistoajurit kuvan 4.4 mukaisesti. Jos ohjelmistorakenteena käytetään jotain tunnettua arkkitehtuuria, niin tätä pitää käyttää hyväksi määriteltäessä kerroksia. Ohjelmiston sovellustaso on yleensä määritelty korkeimmalla tasolla olevaksi kerrokseksi. Vuorovaikutteisia kerroksia määriteltäessä kannattaa käyttää *kytkennän*[7] (coupling) käsitettä hyväksi.

4.3.2 Ohjelmiston rajojen ja toiminnallisten prosessien määrittäminen

Kaikki käyttäjän vaatiman toiminnallisuuden kuvaamat prosessit ohjelmistorajan sisällä ovat osa toiminnallisen koon mittaamista. Raja voidaan määritellä erikseen jokaisen kerroksen välille ja se voi sijaita myös kahden samassa kerroksessa sijaitsevan vertaisosan välillä. Seuraavat säännöt voivat olla hyödyksi ohjelmistossa esiintyviä rajoja tunnistettaessa (Abran et al., 2001b):

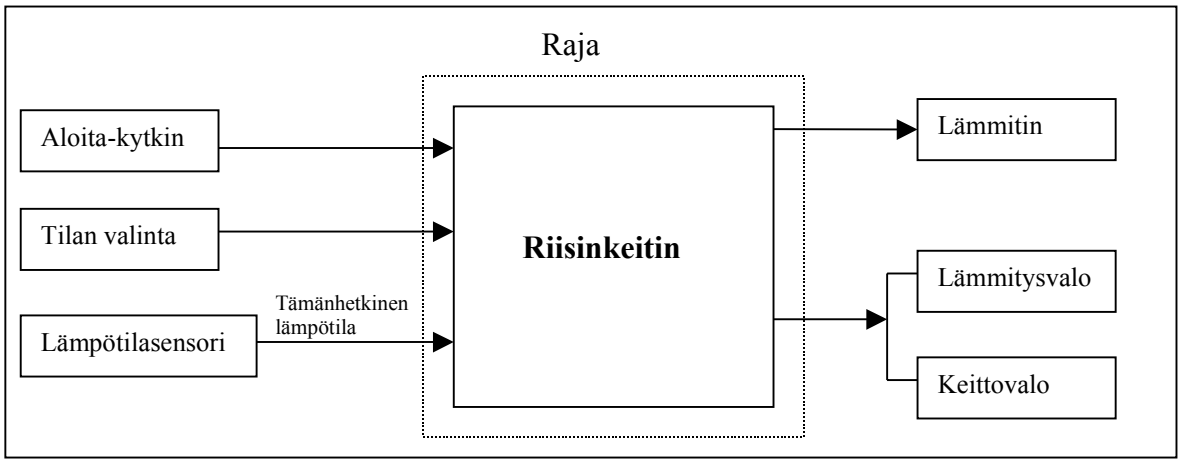
1. Rajojen määrittely kannattaa aloittaa tunnistamalla ns. *laukaisevat tapahtumat* (triggering events) ja näiden aktivoimat toiminnalliset prosessit. Raja sijaitsee laukaisevien tapahtumien ja toimintojen välissä.
2. Tunnista I/O-laitteet, joita mitattava sovellus käyttää. Yhdistä I/O-laitteet sovelluksen tarjoamiin toimintoihin. Raja sijaitsee näiden I/O-laitteiden ja toimintojen välissä.
3. Jos kyseessä on reaaliaikainen tai tekninen sovellus, käytä kerrosten käsitettä rajojen tunnistamisessa.

Laukaiseva tapahtuma tapahtuu mitattavan ohjelmiston ulkopuolella ja se käynnistää yhden tai useamman toiminnallisen prosessin (Abran et al., 2001b). Kello- ja ajoitustapahtumat

[7] Kytkennällä mitataan komponenttien välistä riippuvuutta (Pfleeger, 1998).

voivat olla laukaisevia tapahtumia. Koska jokaisen määritellyn kerroksen välillä kulkee raja, voi laukaiseva tapahtuma sijaita yhdessä kerroksessa ja käynnistää toiminnallisen prosessin jossakin toisessa kerroksessa. Kukin toiminnallinen prosessi johdetaan käyttäjän toiminnallisista vaatimuksista kuvan 4.5 mukaisesti.

Liitteessä 2 kuvatun reaaliaikaisen sovelluksen raja voidaan määritellä esiteltyjen sääntöjen avulla kuvan 4.7 mukaisesti. Laukaisevia tapahtumia on neljä kappaletta eli käyttäjän painamat Aloita-kytkin ja tilan valinta sekä ulkoisen sensorin tekemät tarkistukset riisinkeittimen lämpötilasta (5 ja 30 sekunnin välein). Sovellus puolestaan antaa tietoa sekä käyttäjälle (riisinkeittimen tila: keitto/lämmitys) että lämmittimelle (ON/OFF). Taulukossa 4.3 on lueteltu esimerkkisovelluksen toiminnalliset prosessit.



Kuva 4.7 Liitteen 2 reaaliaikaisen esimerkkisovelluksen raja (Abran et al., 2001a)

4.3.3 Tietoryhmien ja tietoattribuuttien määrittäminen

Tietoryhmä on erillinen, ei-tyhjä, järjestämätön tietoattribuuttien[8] joukko. Tietoryhmä luokitellaan sen pysyvyyden perusteella toiminnallisten vaatimusten suhteen (Abran et al., 2001b):

1. *Hetkellinen* (transient): Tietoryhmä ei säily sitä käyttävän tapahtuman päättymisen jälkeen.

[8] COSMIC-FFP:n versio 2.1 ei edellytä tietoattribuuttien määrittelyä. Tietoryhmään kuuluvien attribuuttien määrittely on kuitenkin mahdollista tehdä, jos haluaa tarkemman arvion sovelluksen koosta. Tietoryhmään kuuluvien attribuuttien määrää voidaan käyttää haluttaessa ”skaalata” COSMIC-FFP:n tuottamaa numeerista arvoa.

2. *Lyhyt* (short): Tietoryhmä on olemassa tapahtuman päättymisen jälkeen, mutta häviää, kun ohjelmiston toiminta loppuu.
3. *Määrittämätön* (indefinite): Tietoryhmä on olemassa vielä ohjelmiston toiminnan lopettamisen jälkeenkin.

Tiedon pysyvyyttä käytetään apuna eroteltaessa neljää aliprosessia (sisääntulo, ulostulo, kirjoitus, luku) toisistaan. Tunnistettuun tietoryhmään liittyy seuraavat periaatteet (Abran et al., 2001b):

1. Tietoryhmän on materialisoiduttava ohjelmistoa ajettavassa tietokonejärjestelmässä.
2. Jokaisen määritellyn tietoryhmän on oltava yksilöllinen ja siksi yksilöitävissä sen attribuuttikokoelman perusteella.
3. Tietoryhmät perustuvat käyttäjän ohjelmistolle asettamiin toiminnallisiin vaatimuksiin.

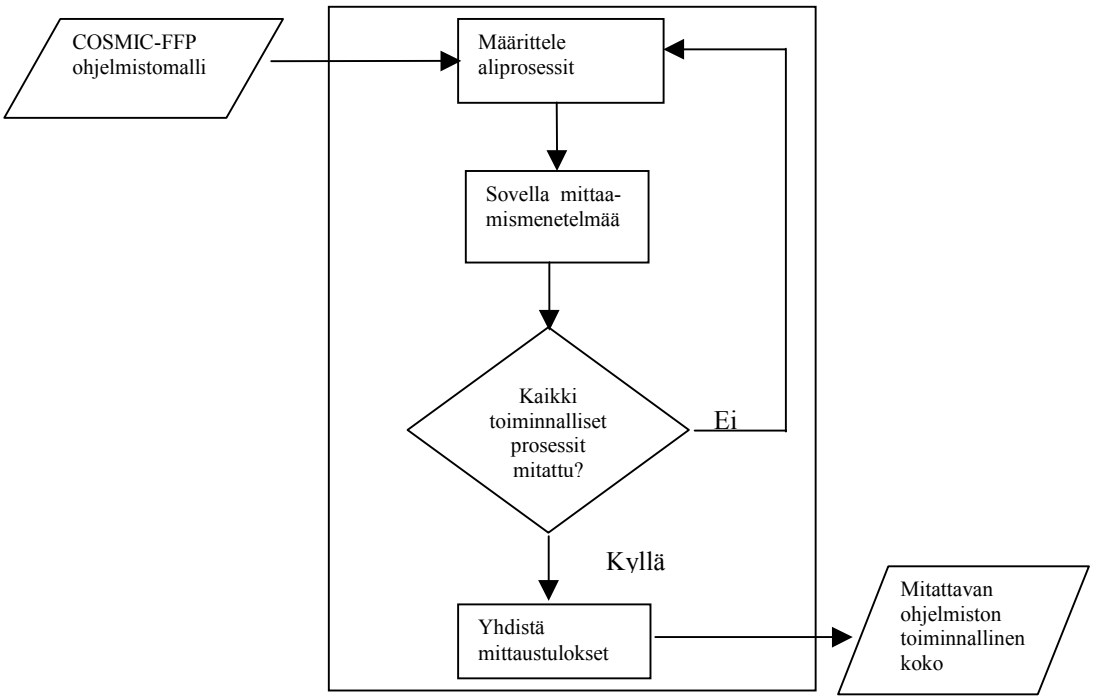
Tietoryhmä voi materialisoitua kolmella tavalla (Abran et al., 2001b):

1. Tietuerakenne pysyvällä muistilaitteella (tiedostot, tietokannan taulut, ROM-muisti, jne). Tämän tyyppisen tietoryhmän pysyvyys on joko lyhyt tai määrittämätön.
2. Fyysinen tietorakenne tietokoneen tilapäisessä muistissa (dynaamisesti varattu rakenne). Tämän tyyppisen tietoryhmän pysyvyys on lyhyt tai hetkellinen.
3. Toiminnallisesti yhteenkuuluvien attribuuttien ryhmittely oheislaitteelle (esimerkiksi näyttölaite, kirjoittimelle tulostetut raportit, ohjauspaneeli). Tämän tyyppisen tietoryhmän pysyvyys vaihtelee riippuen tiedon käytöstä ja oheislaitteesta.

Taulukossa 4.3 on lueteltu liitteen 2 esimerkkisovelluksesta löytyvät tietoryhmät. Esimerkiksi keittotilaa käytetään asetettaessa tavoitelämpötilaa ja kontrolloitaessa merkkivaloja.

4.4 Mittaamisvaihe

Ohjelmiston toiminnallisen koon mittaaminen on riippumaton siitä kuinka paljon itse ohjelmiston kehittäminen on vaatinut työtä tai mitä välineitä ja menetelmiä on käytetty (Abran et al., 2001b). Kuvassa 4.8 on esitetty COSMIC-FFP:n kuvausvaiheen jälkeen seuraavan mittaamisvaiheen jäsentely.



Kuva 4.8 COSMIC-FFP:n mittaamisvaihe yleisellä tasolla (Abran et al., 2001b).

4.4.1 Aliprosessien määrittäminen

COSMIC-FFP:n aliprosessi tarkoittaa tiedon siirtämistä toiminnallisen prosessin suorituksen aikana (Abran et al., 2001b). Aliprosesseja on neljää eri tyyppiä: sisääntulo, ulostulo, luku sekä kirjoitus. Aliprosessit ilmaisevat ainoastaan käyttäjän toiminnallisia vaatimuksia, eivät laatua tai teknisiä vaatimuksia. Aliprosessit määritellään seuraavasti (Abran et al., 2001b):

- Sisääntulo (E): Yhteen tietoryhmään kuuluvien tietoattribuuttien siirtäminen käyttäjän puolelta ohjelmistorajan sisäpuolelle. Sisääntulo ei päivitä siirtämäänsä tietoa. Toiminnallisesti ajatellen aliprosessi sisääntulo tuo tiedon käyttäjän puolelta ohjelmistorajan sisäpuolelle osana toiminnallista prosessia.
- Ulostulo (X): Yhteen tietoryhmään kuuluvien tietoattribuuttien siirtäminen ohjelmistorajan sisältä käyttäjän puolelle. Ulostulo ei lue siirtämäänsä tietoa. Toiminnallisesti ajatellen aliprosessi ulostulo tuo ohjelmistorajan sisäpuolella olevan tiedon käyttäjän puolelle osana toiminnallista prosessia.
- Luku (R): Viittaa yhteen tietoryhmään kuuluviin tietoattribuutteihin. Toiminnallisesti ajatellen aliprosessi luku tuo tiedon tietovarastosta osana toiminnallista prosessia.

- Kirjoitus (W): Viittaa yhteen tietoryhmään kuuluviin tietoattribuutteihin. Toiminnallisesti ajatellen aliprosessi kirjoitus vie tiedon tietovarastoon osana toiminnallista prosessia.

Kaikkiin neljään aliprosessiin kuuluu aina tiedon käsittelyä (Abran et al., 2001b). Sisääntuloon kuuluu muun muassa tiedon kelpuutus ja muotoilu, jotta tieto saataisiin oikeaan muotoon. Ulostuloon kuuluu esimerkiksi tiedon muotoilu raportissa esitettävään muotoon. Ulostuloon ei kuitenkaan kuuluu sisääntulo, joka saatetaan vaatia esimerkiksi raporttiin tulevan tiedon kuvaukseksi. Lukuun/kirjoitukseen kuuluu mukaan myös käsittely, joka vaaditaan tiedon hankkimiseksi/päivittämiseksi.

Jokaisen aliprosessin käsittelemä tietoattribuuttien joukko on yksilöllinen verrattuna muihin samaan toiminnalliseen prosessiin kuuluviin saman tyyppisiin aliprosesseihin. Esimerkiksi sisääntulon käsittelemien tietoattribuuttien joukko on yksilöllinen muihin samaan toiminnalliseen prosessiin kuuluviin sisääntuloihin verrattuna. Yhteenkään aliprosessiin ei kuulu mitään toista aliprosessia, esimerkiksi sisääntuloon ei kuulu ulostulo, kirjoitus eikä luku.

Sisääntuloja määriteltäessä kannattaa ottaa myös huomioon, että *ajoitettuja* (clock triggered) tapahtumia pidetään ulkoisina tapahtumina. Toiminnallista prosessia, joka generoi ajoitetun tapahtuman, ei lasketa mukaan, koska se näin ollen tapahtuu sovellusalueen ulkopuolella. (Abran et al., 2001b). Tästä syystä liitteen 2 esimerkkisovelluksessa aloita-kytkimen painamisen laukaisemaa ajastimen kontrollointia ei kelpuutettu toiminnalliseksi prosessiksi taulukkoon 4.3, koska tapahtuma generoi muita laukaisevia tapahtumia (5 ja 30 sekunnin signaalit).

4.4.2 Mittaamisvaiheen soveltaminen

COSMIC-FFP mittaamisvaiheessa käytetty standardimitta on 1 *Cfsu* (Cosmic Functional Size Unit). Jokainen määritelty aliprosessi (sisääntulo, ulostulo, kirjoitus ja luku) vastaa yhtä Cfsu:ta (Abran et al., 2001b). Käytännössä tämä tarkoittaa sitä, että aliprosessien yhteenlaskettu lukumäärä ilmaisee sovelluksen koon kaavan (4.1) mukaisesti[9].

$$\text{Koko}_{\text{Cfsu}}(\text{kerros}_i) = \text{Koko}\sum\text{koko}(\text{sisääntulot}_i) + \sum\text{koko}(\text{ulostulot}_i) + \sum\text{koko}(\text{luvut}_i) + \sum\text{koko}(\text{kirjoitukset}_i) \quad (4.1)$$

[9] Lukumäärään lisätään myös mahdollisten paikallisten laajennusten toiminnallinen koko.

Ohjelmistoon (kerroksiin) tehtyjen muutosten koko lasketaan kaavalla (4.2) (Abran et al., 2001b).

$$\text{Koko}_{\text{Cfsu}}(\text{Muutos}(\text{kerros}_i)) = \qquad (4.2)$$
$$\sum\text{koko}(\text{lisätyt aliprosessit}_i) + \sum\text{koko}(\text{muokatut aliprosessit}_i) + \sum\text{koko}(\text{poistetut aliprosessit}_i)$$

Abranin et al. (2001b) mukaan on myös hyödyllistä laskea sovelluksen koko aliprosesseittain (kuinka paljon sisääntuloja, ulostuloja, lukuja ja kirjoituksia), kerroksittain tai vaikkapa yksittäisen prosessin suhteen, koska tämä saattaa antaa lisätietoa sovelluksen luonteesta. Sovelluksen koko voidaan ilmaista muodossa *x* Cfsu(v. *y*), jossa *x* edustaa numeerista arvoa, joka on laskettavana olevan kerroksen yhteenlaskun tulos ja *y* edustaa laskennassa käytetyn COSMIC-FFP:n versionumeroa.

Taulukossa 4.2 on liitteen 1 MIS-tyyppisen sovellusesimerkin toiminnallisen koon laskenta COSMIC-FFP:llä[10]. Sovellusalueen raja näkyy IFPUG:n toimintopisteanalyysin yhteydessä esitetystä kuvasta 3.2. Esimerkkisovelluksen toiminnallinen koko ilmaistaan ohjeiden mukaisesti seuraavasti: 81_{Cfsu} (v. 2.0).

Taulukko 4.2 Esimerkkisovelluksen (liite 1) toiminnallinen koko (Fetcke, 1999).

No	Nimi	Tiedon siirtymä	Tyyppi	DET	FFP
1	Lisää Asiakas			7	4
		Käyttäjän syöte	Sisääntulo	2	1
		Nimen tarkistus	Luku	1	1
		Virheilmoitus	Ulostulo	1	1
		Tallenna asiakas	Kirjoitus	3	1
2	Muuta asiakkaan tietoja			16	8
		Käyttäjän syöte	Sisääntulo	1	1
		Hae asiakastiedot	Luku	3	1
		Virheilmoitus	Ulostulo	1	1
		Näytä asiakastiedot	Ulostulo	3	1
		Syötä muutettu tieto	Sisääntulo	3	1
		Hae tuotetiedot	Luku	1	1
		Tallenna tuotetiedot	Kirjoitus	1	1
		Tallenna muutettu tieto	Kirjoitus	3	1
3	Poista asiakas			8	5
		Käyttäjän syöte	Sisääntulo	1	1
		Hae asiakastiedot	Luku	2	1

[10] Fetcke (1999) on käyttänyt esimerkissään (liite 1) COSMIC-FFP:n versiota 2.0, joka ei kuitenkaan eroa laskennassa versiosta 2.1

Taulukko 4.2 Esimerkkisovelluksen (liite 1) toiminnallinen koko (jatk.)

		Virheilmoitus	Ulostulo	1	1
		Hae tuotetiedot	Luku	1	1
		Poista asiakas	Kirjoitus	3	1
4	Ota maksu			6	4
		Käyttäjän syöte	Sisääntulo	1	1
		Hae asiakastiedot	Luku	2	1
		Virheilmoitus	Ulostulo	1	1
		Päivitä tiedot	Kirjoitus	1	1
5	Tallenna tuote			17	6
		Käyttäjän syöte	Sisääntulo	5	1
		Hae asiakastiedot	Luku	1	1
		Hae paikkatiedot	Luku	2	1
		Hae tuotetiedot	Luku	2	1
		Virheilmoitus	Ulostulo	1	1
		Tallenna tiedot	Kirjoitus	6	1
6	Hae tuote			17	6
		Käyttäjän syöte	Sisääntulo	2	1
		Hae tuotetiedot	Luku	5	1
		Hae asiakastiedot	Luku	2	1
		Virheilmoitus	Ulostulo	1	1
		Päivitä asiakastietue	Kirjoitus	1	1
		Poista tuote	Kirjoitus	6	1
7	Lisää paikka			6	4
		Käyttäjän syöte	Sisääntulo	2	1
		Tarkista nimi	Luku	1	1
		Virheilmoitus	Ulostulo	1	1
		Tallenna tiedot	Kirjoitus	2	1
8	Muuta paikan tietoja			13	8
		Käyttäjän syöte	Sisääntulo	1	1
		Hae paikkatiedot	Luku	2	1
		Virheilmoitus	Ulostulo	1	1
		Näytä paikkatiedot	Ulostulo	2	1
		Syötä paikkatiedot	Sisääntulo	2	1
		Hae tuotetiedot	Luku	2	1
		Tallenna tuotetiedot	Kirjoitus	1	1
		Tallenna paikkatiedot	Kirjoitus	2	1
9	Poista paikka			6	5
		Käyttäjän syöte	Sisääntulo	1	1
		Hae paikkatiedot	Luku	1	1
		Hae tuotetiedot	Luku	1	1
		Virheilmoitus	Ulostulo	1	1
		Poista paikka	Kirjoitus	2	1
10	Tulosta asiakkaan tuotteet			13	5
		Käyttäjän syöte	Sisääntulo	1	1
		Hae asiakastiedot	Luku	1	1
		Hae tuotetiedot	Luku	4	1
		Virheilmoitus	Ulostulo	1	1
		Tulosta lista	Ulostulo	6	1
11	Tulosta lasku			10	5
		Käyttäjän syöte	Sisääntulo	1	1
		Hae asiakastiedot	Luku	3	1
		Hae tuotetiedot	Luku	1	1
		Virheilmoitus	Ulostulo	1	1
		Tulosta lasku	Ulostulo	4	1
12	Tulosta lista varastoiduista tuotteista			8	3
		Käyttäjän toiminta	Sisääntulo	1	1
		Hae tuotetiedot	Luku	3	1
		Tulosta lista	Ulostulo	4	1

Taulukko 4.2 Esimerkkisovelluksen (liite 1) toiminnallinen koko (jatk.)

13	Asiakaskysely			8	4
		Käyttäjän syöte	Sisääntulo	1	1
		Hae asiakastiedot	Luku	3	1
		Virheilmoitus	Ulostulo	1	1
		Näytä asiakastiedot	Ulostulo	3	1
14	Kysely asiakkaan tuotteista			13	5
		Käyttäjän syöte	Sisääntulo	1	1
		Hae asiakastiedot	Luku	1	1
		Virheilmoitus	Ulostulo	1	1
		Hae tuotetiedot	Luku	5	1
		Näytä tuotetiedot	Ulostulo	5	1
15	Kysely paikoista			6	4
		Käyttäjän syöte	Sisääntulo	1	1
		Hae paikkatiedot	Luku	2	1
		Virheilmoitus	Ulostulo	1	1
		Näytä paikkatiedot	Ulostulo	2	1
16	Kysely varastoiduista tuotteista			11	5
		Käyttäjän syöte	Sisääntulo	1	1
		Hae paikkatiedot	Luku	2	1
		Virheilmoitus	Ulostulo	1	1
		Hae tuotetiedot	Luku	3	1
		Näytä tuotetiedot	Ulostulo	4	1
			YHTEENSÄ	165	81

Taulukossa 4.3 on esitetty liitteen 2 reaaliaikaisen sovelluksen toiminnallisen koon laskenta (Abran et al., 2001a). Esimerkkisovelluksen toiminnallinen koko on 12_{Cfsu}(v. 2.0).

Taulukko 4.3 Liitteessä 2 kuvatun reaaliaikaisen sovelluksen toiminnalliset prosessit yksityiskohtaisesti sekä toiminnallinen koko (Abran et al., 2001a).

Prosessin kuvaus	Laukaiseva tapahtuma	Aliprosessi	Tietoryhmä	T	FFP	
Keittotilan valinta	Tilakytkimen painaminen	Vastaanota keittotila	Keittotila	E	1	
		Kirjoita keittotila	Keittotila	W	1	2
Aseta kohdelämpö	30 sekunnin signaali	Vastaanota kulunut aika	Kulunut aika	E	1	
		Lue keittotila	Keittotila	R	1	
		Kirjoita tavoitelämpötila	Tavoitelämpötila	W	1	3
Lämmittimen kontrollointi	5 sekunnin signaali	Vastaanota laukaiseva tekijä	5 sekunnin signaali	E	1	
		Vastaanota nykyinen lämpö	Nykyinen lämpötila	E	1	
		Lue tavoitelämpötila	Tavoitelämpötila	R	1	
		Aseta lämmitin ON/OFF	Lämmittimen tila	W	1	4
Merkkivalojen kontrollointi	30 sekunnin signaali	Vastaanota kulunut aika	Kulunut aika	E	1	
		Lue keittotila	Keittotila	R	1	
		Lähetä keittäjän tila merkkivaloille	Keittimen tila	X	1	3
			YHTEENSÄ			12

Taulukosta 4.3 näkyy hyvin mitä asioita sovelluksesta COSMIC-FFP:n mukaan halutaan mitata: prosessit, niihin kuuluvat aliprosessit ja laukaisevat tekijät. Myös tieto siitä, mihin tietojoukkoon aliprosessi viittaa, on oltava selvillä. Taulukossa 4.4 näkyy Abranin et al. (2001a)

suosittelema tapa esittää laskennan lopputulos. Taulukossa on myös mainittu prosessin id -sarake muodossa *x.y*, jossa *x* tarkoittaa kerrosta ja *y* on prosessin numero kyseisen kerroksen sisällä.

Taulukko 4.4 Yhteenveto liitteen 2 sovelluksen toiminnallisesta koosta COSMIC-FFP:llä laskettuna (Abran et al., 2001a).

Nro	Prosessin ID	Prosessin kuvaus	E	X	R	W	FFP Pisteet (Cfsu)
1	1.1	Keittotilan valinta	1	0	0	1	2
2	1.2	Aseta kohdelämpötila	1	0	1	1	3
3	1.3	Lämmittimen kontrollointi	2	1	1	0	4
4	1.4	Merkkivalojen kontrollointi	1	1	1	0	3
Yhteenveto:		4 prosessia	5	2	3	2	12

4.5 Tutkimustulos COSMIC-FFP:stä

Abran et al. (2001c) ovat tutkineet kuinka hyvin COSMIC-FFP soveltuu sovelluksen toiminnallisen koon mittaamiseen[11]. Tutkimuksen tarkoituksena oli mm.:

- mitata, onko COSMIC-FFP:n dokumentointi ymmärrettävää ja menetelmä toistettavissa riippumatta toimialasta tai käytettävästä teknologiasta.
- selvittää, vastaako sovelluksen toiminnallinen koko sovelluksen kehittämiseen tarvittua työmäärää.

Tarkoituksena oli myös saada COSMIC-FFP käyttöön niissä yrityksissä, jotka olivat mukana tehdyssä tutkimuksessa. Kokeeseen osallistuneet yritykset saivat myös käyttöönsä tutkimustulokset, joista kävi ilmi mm. kyseisen yrityksen toimintateho verrattuna muihin kokeeseen osallistuneisiin yrityksiin. Kokeessa COSMIC-FFP:tä sovellettiin hyvin erilaisille sovellusalueille ja alustoille sekä kehitys- että ylläpitoprojekteissa (Abran et al., 2001c).

COSMIC-FFP:n toistettavuudesta järjestettiin kaksi koetta (Abran et al., 2001c). Toistettavuudella tarkoitetaan sitä, kuinka samanlaisia tuloksia saavutettiin samasta sovelluksesta eri mittaajien välillä. Ensimmäisessä kokeessa käytettiin ilmailualaan liittyvän ohjelmiston dokumentaatiota. Ohjelmiston kokoa mitanneet suunnittelijat, joista jokainen määritteli kahden do-

[11] Tutkimuksessa käytettiin COSMIC-FFP:n versiota 2.0.

kumentaatiossa olleen osan koon, tunsivat sovellusalueen. Lähes täydellinen toistettavuus saavutettiin.

Toisessa testissä käytettiin prosessin hallintaan liittyvän sovelluksen määrittelyjä. Niiden suunnittelijoiden kohdalla, joilla oli kokemusta prosessin hallinnasta, saavutettiin hyvä toistettavuus. Kokemattomat ohjelmoijat, joilla ei ollut kokemusta mitattavasta sovellusalasta, saavuttivat selvästi huonomman toistettavuuden. Loppupäätelmä on, että kokemus sekä COSMIC-FFP:stä että sovellusalasta ovat kumpikin välttämättömiä hyvän toistettavuuden saavuttamiseksi. Mukana olleiden mielestä COSMIC-FFP oli "helppo" tai "kohtuullisen helppo" käyttää.

Abranin et al. (2001c) tutkimuksessa mitattiin myös ohjelmiston kehittämiseen/ylläpitoon vaaditun työmäärän ja toiminnallisen koon välistä korrelaatiota. Taulukossa 4.5 näkyy tutkittavana ollut aineisto sekä käytetty määrittely-, ohjelmointi- ja testausaika. Mukana oli 12 ohjelmistoprojektia, joiden yhteinen koko oli 13 $961_{Cfsu.}$

Tutkittavaan aineistoon liittyi seuraavia ongelmia (Abran et al., 2001c):

- Aineisto koostui 12 projektista, mikä ei välttämättä ole riittävästi.
- Joidenkin avainmuuttujien epähomogeenisuus. Aineistoon kuului 12 projektia, mutta näistä kahden projektin yhteenlaskettu koko oli 11 255_{Cfsu}, jonka jälkeen muiden 10 projektin yhteenlasketuksi kooksi jää vain 2 706_{Cfsu}. Tämä epäsuhta vaikeuttaa tilastollisten menetelmien käyttöä.
- Epähomogeenisuus tavassa, jolla mukana olleet yritykset kehittävät ohjelmistojaan. Vaikka yrityksen sisäinen tapa kehittää ohjelmia olisi hyvinkin standardoitu, erilaisuutta on silti yritysten välillä.

Taulukosta 4.5 voidaan korrelaatio koon ja vaadittavan työmäärän välillä havaita silmämääräisestikin, mutta taulukossa 4.6 näkyy tarkat arvot. Korrelaatiot ovat suuria ottaen huomioon tiedon alkuperän epähomogeenisuuden (kolme organisaatiota kolmesta eri maasta erilaisine ohjelmiston kehittämismenetelmineen) (Abran et al., 2001c). Tutkimustulokset ovat kuitenkin vasta alustavia ja niihin tulee suhtautua tietyllä varauksella.

Taulukko 4.5 COSMIC-FFP:n testaamiseen käytetty aineisto (Abran et al., 2001c).

Projekti	Tyyppi	Koko ($_{Cfsu}$)	Määrittelyaika (henkilö/tunti)	Ohjelmointiaika (henkilö/tunti)	Testausaika (henkilö/tunti)
A	Kehitys	32	-	252	-
B	Kehitys	76	381	1 457	401
C	Kehitys	56	68	487	335
D	Kehitys	142	136	643	-
E	Kehitys	8	115	116	-
F	Ylläpito	142	2 060	1 487	5 055
G	Kehitys	332	468	11 382	254
H	Ylläpito	624	-	15 815	1 372
I	Kehitys	810	1 304	10 903	4 548
J	Kehitys	484	-	20 808	6 772
K	Kehitys	8 251	49 000	66 000	93 000
L	Kehitys	3 004	32 000	27 000	20 000
Yhteensä		13 961			

Taulukko 4.6 Tutkimuksessa ilmenneet korrelaatiot (Abran et al., 2001c).

	Ohjelmiston toiminnallinen koko
Määrittelyjen tekemiseen kulunut aika	0.7824
Ohjelmiston valmistamiseen kulunut aika	0.8971
Testaamiseen kulunut aika	0.7019

5. JOHTOPÄÄTÖKSET

Mielestäni hyvä lähtökohta sovelluksen koon arvioimiselle on sovelluksen käyttäjälle tarjoama toiminnallisuus. Ohjelmiston kehittäjä voi laskea asiakkaalle toimitettavan sovelluksen koon samalla tavalla riippumatta kehityksessä käytetyistä välineistä. Vastaavasti asiakas edellyttää tilaamaltaan ohjelmistolta tiettyä toiminnallisuutta riippumatta siitä, millä välineillä ohjelmisto on toteutettu. Ohjelmiston toiminnallinen koko voidaan myös arvioida heti, kun käyttäjän vaatimukset ovat selvillä. Riippumattomuus ohjelmointivälineistä antaa lisäksi hyvät mahdollisuudet erilaisten ohjelmistokehitystä koskevien ja keskenään vertailukelpoisten tunnuslukujen laskemiseen.

Luvussa 3 esitelty IFPUG 4.1 on osoittautunut varsin hyväksi siinä suhteessa, että näin lasketu toimintopistearvo korreloi ohjelmiston kehityksessä vaadittavan työmäärän kanssa. IFPUG 4.1:n ongelmana pidetäänkin sitä, ettei se ei ole riittävän yksinkertainen suorittaa. Monimutkaisuuden takia eri tahot saavat laskelmissaan erilaiset toimintopistearvot saman toiminnallisuuden omaavista ohjelmistoista. Näitä mittauseroja syntyy erityisesti vertailtaessa yritysten välisiä mittaustuloksia. Yritysten sisäisenä arviointimenetelmänä IFPUG 4.1 voi kuitenkin osoittautua hyväksi ja toistettavissa olevaksi menetelmäksi varsinkin, kun yritys on saanut siitä kokemusta muutaman ohjelmiston verran. Sen sijaan menetelmään kuuluvat neljätoista sovitustekijää ovat osoittautuneet tutkimuksien valossa hyödyttömiksi nykyisessä muodossaan käytettyinä. Sovitustekijöitä olisikin syytä käyttää erillisinä arvoina antamaan sovelluksen luonnetta kuvaavaa informaatiota.

IFPUG:n toimintopistelaskennan ongelmana on pidetty yksiselitteisyyden puutetta ja sitä, ettei se sovellu kaiken tyyppisten ohjelmistojen arviointiin. Luvussa 4 esitellyn COSMIC-FFP:n on tarkoitus tarjota ratkaisu näihin ongelmiin. COSMIC-FFP:n yksi perusajatus on se, että sovellus koostuu eri kerroksista. Tämä antanee joustavan tavan mitata ohjelmiston kokoa eri näkökulmista katsottuna, mutta vaatii tutkimustuloksia. Alustavana tutkimuksena on kuitenkin huomattu, että menetelmällä saatu arvio ohjelmiston koosta korreloi ohjelmiston määrittelyjen tekemiseen, kehittämiseen ja testaamiseen käytetyn työmäärän kanssa. Samoin COSMIC-FFP näyttäisi olevan arviointimenetelmänä helposti toistettavissa. Edellä mainittujen korrelaation ja toistettavuuden saavuttaminen ovat toimintopistelaskennan perustavoitteet. COSMIC-FFP:kään ei silti tarjoa selkeää ratkaisua ohjelmistossa esiintyvien algoritmien monimutkaisuuden arviointiin. On ehkä lisäksi hyvä palauttaa mieliin tutkimustuloksen (Abran et al.,

2001c) loppupäätelmä, jonka mukaan sekä kokemus COSMIC-FFP:stä että sovellusalasta ovat kumpikin välttämättömiä hyvän toistettavuuden saavuttamiseksi. Vaikka sovellusalueesta olisikin kokemusta, ei reaaliaikaisen sovelluksen arviointi ole välttämättä mikään triviaali asia.

Horganin et al. (1998) mielestä on olemassa todisteita siitä, että yksinkertaiset menetelmät toimivat laskennan alkuvaiheessa yhtä hyvin, kuin varsinainen toimintopisteanalyysi. Tämän väitteen perusteella IFPUG:in käytettävyys näyttää huomattavasti paremmalta. Horganin et al. (1998) ehdottama malli on jopa yksinkertaisempi suorittaa kuin COSMIC-FFP. Jotta IFPUG saataisiin nykyajan vaatimuksia vastaavaksi, pitäisi siihen kuitenkin mielestäni ehdottomasti lisätä kerroksen käsite. Toisaalta, sisäiset loogiset tiedostot ja ulkoiset liitäntätiedostot aiheuttaisivat tässä "kerrosten maailmassa" lisäsekaannusta jo muutenkin yksiselitteisyyden puutteesta kärsivään menetelmään. Pelkkä käyttäjän näkökulma ei jatkossa kuitenkaan tule riittämään, joten IFPUG:in kehittäjillä riittää haasteita tulevaisuudessa. Menetelmän yksinkertaisuuden vuoksi COSMIC-FFP:hen kerrokset sopivat sen sijaan hyvin. COSMIC-FFP:n ongelma voikin olla sen liiallinen suoraviivaisuus. Laskennan pitää toki olla yksinkertaista, mutta sen pitää antaa myös oikea kuva ohjelmiston koosta. IFPUG:in tapa käsitellä tietoalkiotyyppejä saattaisi antaa viitteitä suunnasta, johon COSMIC-FFP:tä joudutaan jatkossa kehittämään. Vaikka Abranin et al. (2001c) tutkimus kertookin COSMIC-FFP:llä saatujen tulosten korreloivan ohjelmistoon käytetyn suunnittelun, työajan ja testaamisen kanssa, ei yhden tutkimuksen perusteella tutkimuksen tekijöidenkään mielestä voi tehdä liian pitkälle meneviä johtopäätöksiä. COSMIC-FFP:n käyttäjien kannattaakin mielestäni tarkkailla, antaako laskenta todellisuutta vastaavan kuvan.

Abran et al. (2001c) ja Symons (2001) suosittelevat molemmat UML:ää ja etenkin käyttötapauskaaviota COSMIC-FFP:n toiminnallisten prosessien tunnistamisessa. Käyttötapauskaavion suunnittelu on hyvä omaksua myös IFPUG:in alkeisprosesseja tunnistettaessa. Myös monet muut UML:n kaaviotyypit sopivat käytettäväksi toimintopistelaskennan yhteydessä. Näistä mainittakoon käyttötapauskaavion lisäksi luokka- ja sekvenssikaavio. Uemuran et al. (1999) esittämä mallinnuskielen hyväksikäyttö toimintopisteanalyysissa onkin mielenkiintoinen. Mallinnuskielen käyttö on helppo tapa lisätä systemaattisuutta ja toistettavuutta toimintopistelaskentaan. Uemura et al. olettavat laskennassaan tietuealkiotyyppien (RET) lukumäärän aina yhdeksi; ja tässä voisikin olla myös tapa yksinkertaistaa toimintopisteanalyysiä.

Symons (2001) arvioi, että tarvetta toimivalle arviointimenetelmälle olisi, koska toimintopistelaskennan suosio on ollut laskusuunnassa 1990-luvulla. COSMIC-FFP tarjoaa ratkaisun toimintopistelaskennan suurimpiin ongelmiin ja voisi näin ollen kääntää kehityssuunnan. Toisaalta, jos vanhat mittausmenetelmät ovat niin auttamattomasti vanhentuneita, kuin Symons antaa ymmärtää, aiheuttaa se myös ongelmia. Ohjelmistojen valmistajat pitää saada vakuuttuneiksi siitä, että uusi menetelmä on luotettava, yksinkertainen ja että sitä voidaan käyttää pitkälle tulevaisuuteen. Oman ongelmansa muodostavat vanhoilla arviointimenetelmillä saadut tulokset, joita ei voida konvertoida yhteensopiviksi uuden menetelmän kanssa. Vanhojen laskentatulosten osoittautuminen hyödyttömiksi tuskin lisää halukkuutta minkään uuden menetelmän käyttöönottoon.

VIITELUETTELO

Abran A., Robillard P.N.: Function Points Analysis: An Empirical Study of Its Measurement Processes. *IEEE Transactions on software engineering.* 22(12), 895-910, 1996.

Abran A., Desharnais J., Oligny S., St-Pierre D., Symons C., Fagg P.: *Case Study, Rice Cooker, Version 2.1.* Internet WWW-sivu, URL: http://www.lrgl.uqam.ca/cosmic-ffp/ casestudies/version2.1RiceCookerCaseStudy.PDF (07.12.2001a).

Abran A., Desharnais J., Oligny S., St-Pierre D., Symons C.: *COSMIC-FFP Measurement Manual, version 2.1*. Internet WWW-sivu, URL: http://www.lrgl.uqam.ca/publications/ private/446.pdf (7.12.2001b).

Abran A., Symons C., Oligny S.: *An overview of COSMIC-FFP field trial results*. Internet WWW-sivu, URL: http://www.lrgl.uqam.ca/publications/pdf/614.pdf (07.12.2001c).

Booch G., Rumbaugh J., Jacobson I.: *The Unified Modelling Language User Guide*, Addison-Wesley, Reading, Massachusets, 1999.

Dreger J.B.: *Function point analysis*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

Elmasri R., Navathe S.B.: *Fundamentals of Database Systems.* Addison-Wesley, Reading Massachusetts, 2000.

Fetcke T.: *The Warehouse Software Portofolio, A Case Study in Functional Size Measurement,* Internet WWW-sivu, URL: http://www.lrgl.uqam.ca/cosmic-ffp/casestudies/ Fetcke1999b.pdf (07.12.2001).

Furey S.: Why we should use function points. *IEEE Software*, 14(2), 28-30, 1997.

Garmus D., Herron D.: *Function Point Analysis, Measurement Practices for Successful Software Projects*. Addison - Wesley, USA, 2000.

Horgan G., Souheil K., Forte P.: Construction of an FPA-type metric for early lifecycle estimation. *Information and Software Technology*, 40, 409-415, 1998.

Jones C.: Software sizing. *IEE Review*, 45(4), 165-167, 1999.

Kitchenham B.: The Problem with Function Points. *IEEE Software*, 14(2), 29-31, 1997.

Lokan C.J.: An empirical study of the correlations between function point elements. *Proceedings of the sixth International Software Metrics Symposium 1999,* 200-206, 1999.

Lokan C.J.: An empirical analysis of function point adjustment factors. *Information and Software Technology*, 42, 649-660, 2000.

Low G.C., Jeffery D.R.: Function Points in the Estimation and Evaluation of the Software Process. *IEEE Transactions on software engineering.* 16(1), 64-71, 1990.

Pfleeger S.L.: *Software engineering, Theory and practice.* Prentice Hall, Upper Saddle River, New Jersey, 1998.

Symons C. *Come back function point analysis (modernised) - all is forgiven!*. Internet WWW-sivu, URL: http://www.lrgl.uqam.ca/publications/pdf/647.pdf (07.12.2001).

Uemura T., Kusumoto S., Inoue K.: Function point measurement tool for UML design specification. *Proceedings of the sixth International Software Metrics Symposium 1999,* 62-69, 1999.

Esimerkki MIS-tyyppisestä sovelluksesta (Fetcke, 1999)

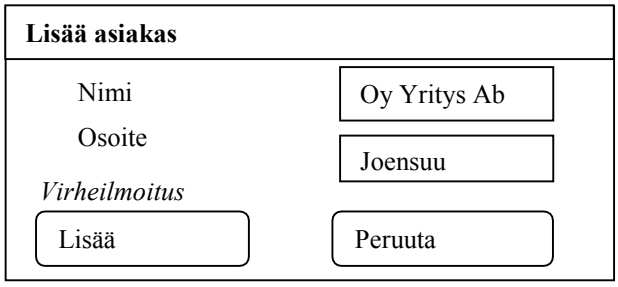
Esimerkkinä on yksinkertainen sovellus, jota käyttävät ne yritykset, joilla on varastotilaa vuokrattavana muille yrityksille. Sovellus käyttää ja päivittää tiedostoja Asiakas, Tuote ja Paikka:

Tiedostossa Asiakas on varastotilaa haluavan asiakkaan tiedot: Nimi (perusavain), Osoite sekä Maksu. Maksu-kentässä on tieto siitä, kuinka paljon asiakkaalla on maksettavaan varastotilasta. Tuote-tiedoston perusavain on Kuvaus-kenttä. Kuorma on varastointikoon yksikkö. Tuotetta voi olla varastossa esimerkiksi kaksi kuormallista. Arvo-kenttä tarvitaan esimerkiksi vakuutusmaksuja varten. Sen lisäksi Tuote-tiedostossa on varastointipäivälle oma kenttänsä sekä viiteavaimet Omistaja ja Varastopaikka. Paikka-tiedosto sisältää kaksi kenttää: Sijainti (perusavain) ja Tila. Tila-kentässä ilmoitetaan, kuinka monta kuormallista tuotteita mahtuu kyseiseen paikkaan (Sijainti). Asiakkaalla voi olla varastossa useita tuotteita ja yhdessä varastopaikassa voi olla useita tuotteita varastoituina.

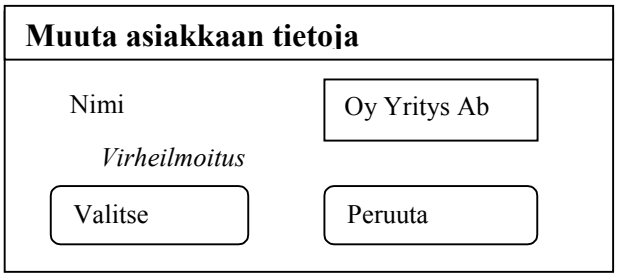
Sovelluksessa voidaan tunnistaa 16 tapahtumaa:

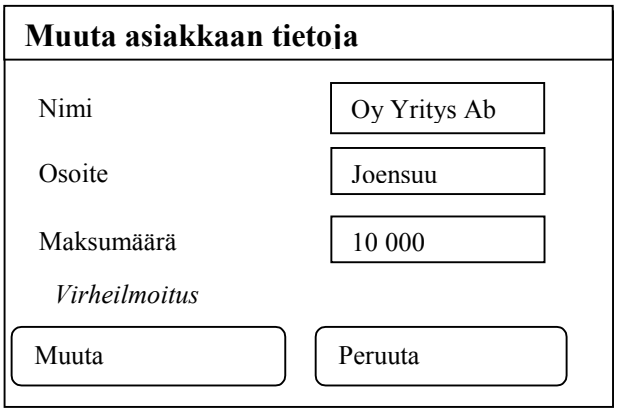
1 Lisää asiakas
2 Muuta asiakkaan tietoja
3 Poista asiakas
4 Ota maksu
5 Tallenna tuote (asiakas tuo tuotteen varastoitavaksi)
6 Nouda tuote (asiakas hakee tuotteen varastosta)
7 Lisää uusi paikka
8 Muuta paikan tietoja
9 Poista paikka
10 Tulosta asiakkaan tuotteet
11 Tulosta lasku
12 Tulosta lista varastoiduista tuotteista
13 Asiakaskysely
14 Kysely asiakkaan tuotteista
15 Kysely paikoista
16 Kysely varastoiduista tuotteista

Lisää asiakas: Asiakastietoja syötettäessä käyttäjän on syötettävä kenttiin nimi ja osoite. Maksukenttä alustetaan ohjelmallisesti nollaksi. Kun käyttäjä painaa Lisää-painiketta lisätään tietue tietokantaan. Asiakastietoja syötettäessä tarkistetaan, onko mahdollisesti kyseinen nimi olemassa, jos on, tulee virheilmoitus. Käyttäjä voi peruuttaa toiminnon Peruuta-painikkeella.

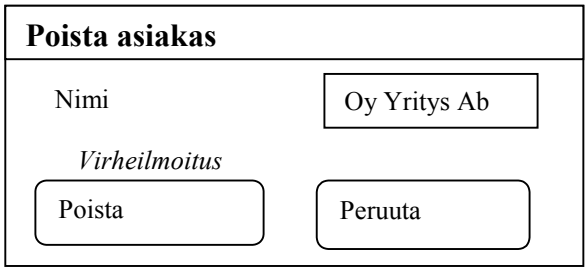


Muuta asiakkaan tietoja: Käyttäjä valitsee haluamansa asiakkaan nimen perusteella. Näkyviin tulee kaikki Asiakas-tiedoston tietokentät muokkaamista varten. Käyttäjä voi muuttaa minkä tahansa kentän arvoa. Jos käyttäjä muuttaa nimikentän arvoa, muutetaan kaikkiin vastaaviin Tuote-tiedostoissa oleviin omistaja-kenttiin vastaava nimi. Jos uusi nimi on jo olemassa, tulee virheilmoitus.

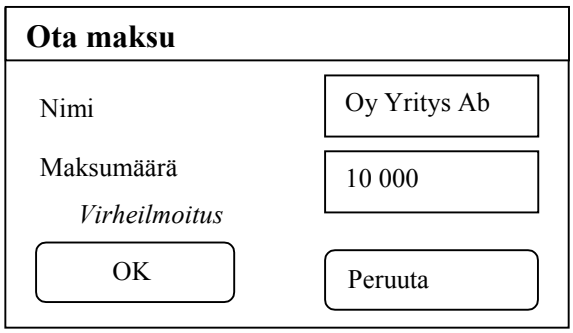




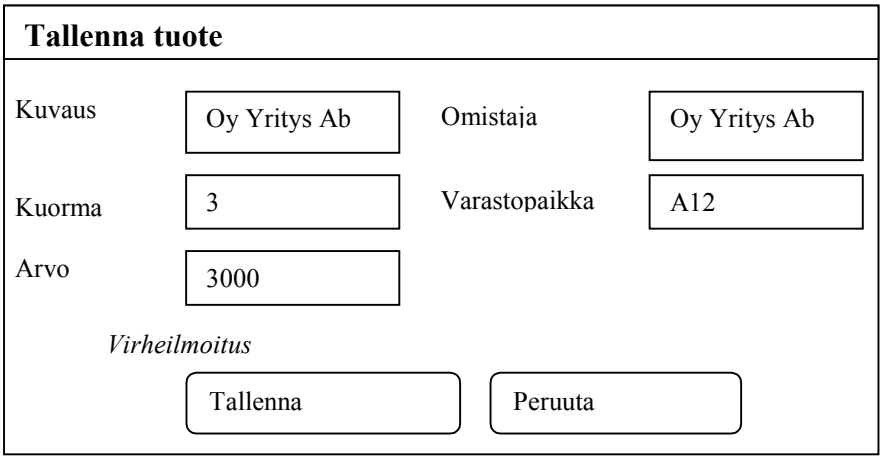
Poista asiakas: Poistettavan asiakkaan tiedot haetaan asiakkaan nimen perusteella. Jos maksu-kenttä ei ole nolla tai jos asiakkaalla on tuotteita varastossa, tietuetta ei voi poistaa ja tulee virheilmoitus. Virheilmoitus tulee myös, jos haettua asiakasta ei ole tietokannassa.



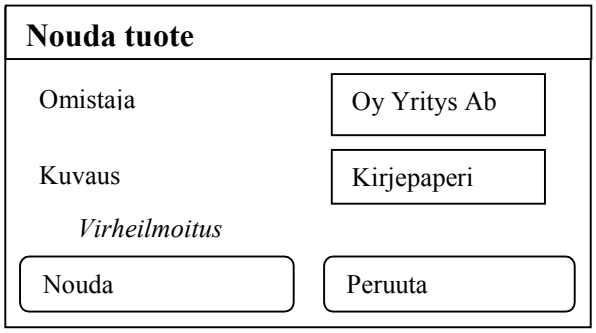
Ota maksu: Asiakkaita velotetaan sen mukaan, kuinka paljon heillä on tuotteita varastossa. Kun asiakas maksaa laskunsa, suoritetaan tapahtuma "Ota maksu". Asiakkaan tiedot haetaan nimen perusteella. Maksu-kentästä vähennetään asiakkaan maksama määrä ja asiakkaan tiedot päivitetään tietokantaan. Jos kyseistä asiakasta ei ole, tulee virheilmoitus.



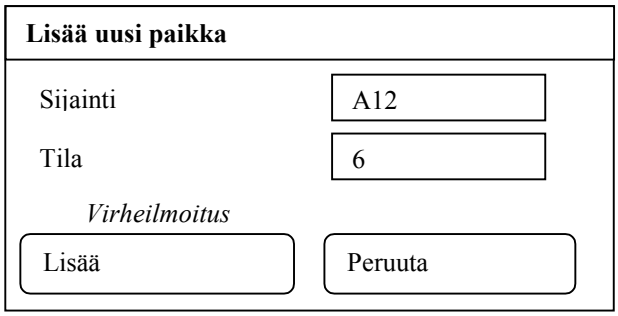
Tallenna tuote: Kun asiakas tuo tuotteen varastoitavaksi, viedään tiedot Tuote-tiedostoon. Asiakas tunnistetaan omistajakentän avulla; samanniminen asiakas on löydyttävä Asiakas-tiedostosta. Tuote varastoidaan tiettyyn paikkaan varastossa ja vastaava tieto on löydyttävä Paikka-tiedostosta. Tuote-tiedostosta käyttäjä syöttää tiedot kaikkiin tietokenttiin. Virheilmoitus tulee, jos vastaavaa Asiakas-tietuetta ei löydy, vastaavaa Paikka-tietuetta ei löydy tai vastaava Kuvaus on jo olemassa.



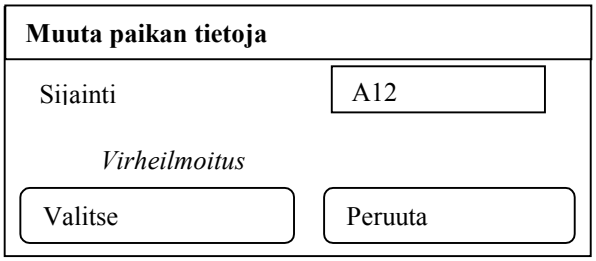
Nouda tuote: Kun asiakas hakee tuotteensa pois varastosta, vastaava Tuote-tietue poistetaan tietokannasta. Asiakas tunnistetaan omistaja-kentän perusteella ja Asiakas-tietueeseen lisätään maksu-kenttään kyseisen tuotteen varastointimaksu (Asiakas-tiedostoa päivitetään). Tapahtuma epäonnistuu, jos vastaavaa kuvaustietoa ei löydy Tuote-tiedostosta tai kyseinen asiakas ei omista poistettavaksi haluttua tuotetta. Tapahtuman epäonnistumisesta tulee virheilmoitus. Maksumäärä lasketaan kiinteällä kaavalla, joka perustuu varastointiaikaan ja siihen tilaan, jonka varastointi on vaatinut.

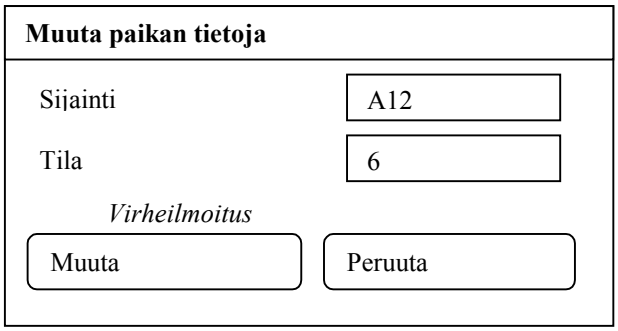


Lisää uusi paikka: Kun käyttöön tulee uusi varastopaikka, pitää se kirjata tietokantaan uutena Paikka-tietueena. Päivitettävät kentät ovat Sijainti ja Tila. Jos kyseistä Sijaintia ei ole olemassa, päivitetään uusi tietue, muutoin annetaan virheilmoitus.

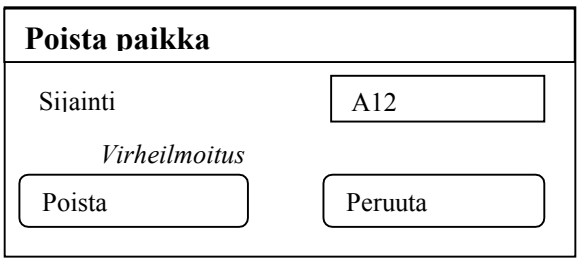


Muuta paikan tietoja: Käyttäjä voi myös muuttaa varastopaikan tietokentän Sijainti avulla. Jos kyseistä sijaintia ei ole olemassa, tulee virheilmoitus. Myöskään jo olemassa olevaa sijaintia ei hyväksytä. Tila-kentän arvoa ei voida asettaa pienemmäksi, kuin varastoidut tuotteet vaativat, muutoin tulee virheilmoitus.

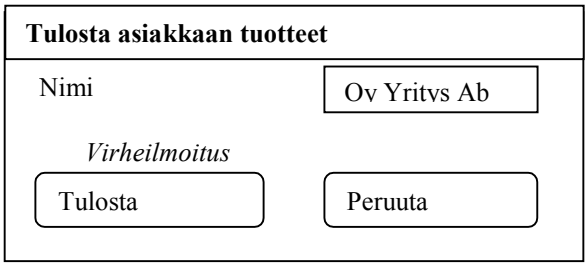


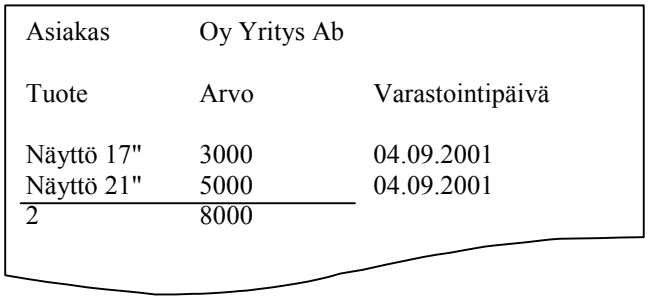


Poista paikka: Varastopaikka voidaan myös poistaa varastopaikan avulla. Jos kyseisessä varastopaikassa on varastoituna tuotteita, tulee virheilmoitus.

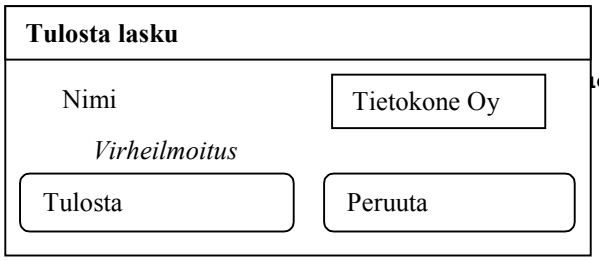


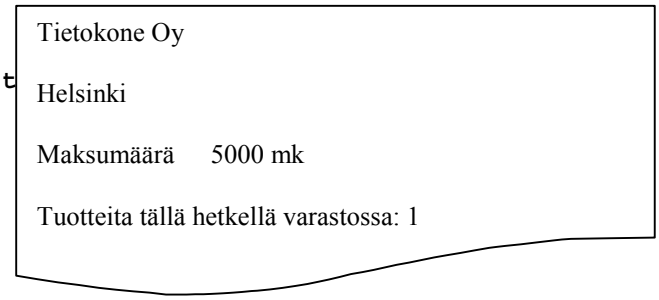
Tulosta asiakkaan tuotteet: Asiakkaan tiedot voidaan tulostaa asiakkaan nimen perusteella. Jos kyseinen asiakas on tallennettu tietokantaan, tulostetaan lista, jossa on tuotteen Kuvaus, Arvo, Varastointipäivä sekä summataan loppuun tuotteiden yhteisarvo.





Tulosta lasku: Asiakkaalle lähetettävä lasku sisältää nimen ja osoitteen ja laskun määrän. Lasku sisältää myös sillä hetkellä varastoituna olevat asiakkaan tuotteet. Jos sennimistä asiakasta ei ole, tulee virheilmoitus.

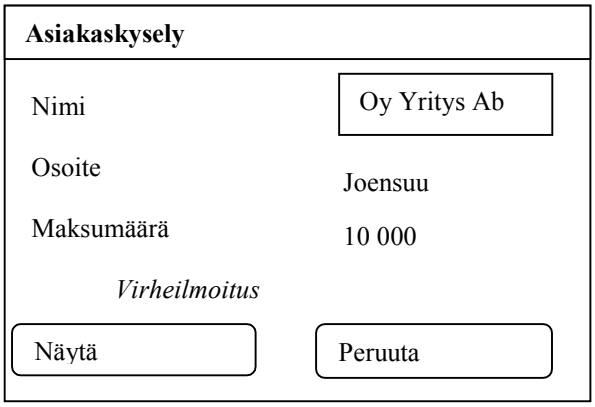




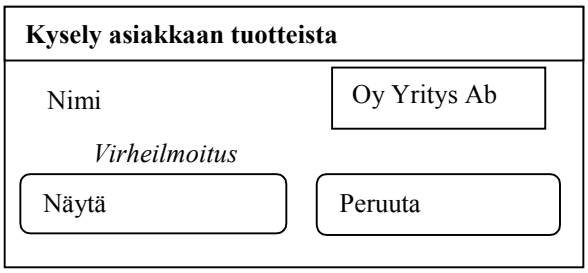
Tulosta lista varastoiduista tuotteista: Ohjelman avulla voidaan myös tulostaa lista kaikista varastossa olevista tuotteista. Jokaisesta tuotteesta tulostetaan Kuvaus, Kuorma ja Sijainti. Myös varastossa olevien tuotteiden kokonaismäärä tulostetaan.

Tuote	Kuorma	Varastopaikka
Kirjepaperi	2	A22
Näyttö 17"	2	A33
Näyttö 21"	2	A34
Paperiliittimiä	5	A35
	11	

Asiakaskysely: Asiakaskyselyllä saadaan tulostettua käyttäjän syöttämää Nimeä vastaavan asiakkaan tiedot. Jos kyseinen asiakastietue on olemassa, tulostetaan asiakkaan Nimi, Osoite ja Maksumäärä (eli määrä, joka asiakkaalla on maksettavaan varastoinnista). Jos kyseistä asiakastietuetta ei löydy, tulostetaan virheilmoitus.



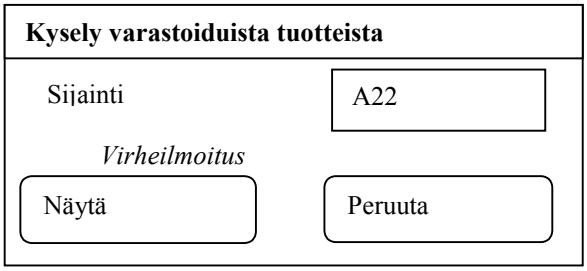
Kysely asiakkaan tuotteista: Tämän tapahtuman avulla voidaan tulostaa kaikki tietyn asiakkaan varastossa olevat tuotteet. Käyttäjän on syötettävä haluamansa asiakkaan nimi. Jos kyseinen asiakastietue on olemassa, tulostetaan Kuvaus, Kuorma, Arvo ja Varastointipäivä kaikista asiakkaan omistamista tuotteista.



Asiakas	Oy Yritys Ab		
Tuote	Kuorma	Arvo	Varastointipäivä
Paperiliitin	3	400	04.05.2001
Saksia	3	200	04.05.2001

Kysely paikoista: Varastopaikkojen tila saadaan myös selville. Käyttäjä syöttää Sijainnin. Jos kyseinen Sijainti on olemassa, tulostetaan Sijainti- ja Tila-tiedot. Muussa tapauksessa tulostetaan virheilmoitus. Käyttöliittymä on samanlainen kuin "Muuta paikan tietoja"-tapahtumassa, mutta käyttäjä ei voi tehdä muutoksia paikkojen tietoihin tässä kohdassa.

Kysely varastoiduista tuotteista: Tämän tapahtuman avulla voidaan tulostaa tietyssä varastopaikassa olevien tuotteiden tiedot. Käyttäjä syöttää Sijainnin. Jos kyseinen varastopaikka on olemassa, tulostetaan Sijainti- sekä varastopaikan Tila-tiedon. Muussa tapauksessa tulostetaan virheilmoitus.

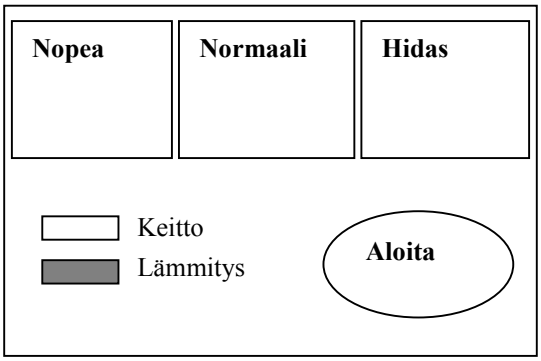


Sijainti	A22
Tila	6
Tuote	Kuorma
Näyttö 17"	2
Näyttö 21"	2
Paperiliitin	2

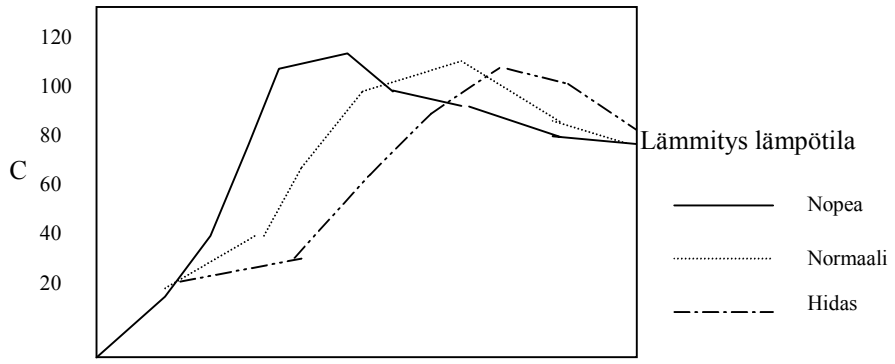
Esimerkki reaaliaikaisesta sovelluksesta (Abran, 2001a)

Riisinkeitin-sovelluksen määrittelyt

1. Riisin keittämisen pitää olla mahdollista kolmessa eri tilassa: nopea, normaali ja hidas.



2. Riisin keittäminen alkaa, kun käyttäjä on painanut Aloita-kytkintä. Normaalisti tämä tapahtuu keittotilan valitsemisen jälkeen.
3. Jos riisin keittäminen alkaa ilman, että käyttäjä on valinnut tilaa, riisin keitto alkaa automaattisesti normaalitilassa.
4. Kun riisin keittäminen on valmis, siirtyy riisinkeitin automaattisesti lämmitystilaan (kaikissa kolmessa eri tilassa).
5. Oikean merkkivalon pitää palaa etupaneelissa. Se kertoo käyttäjälle, onko menossa keittämis- vai lämmitysvaihe.
6. Lämmitystä pitää voida kontrolloida "tilan/lämpötilan/kuluneesn ajan perusteella (katso kuva 2). Ohjelmiston pitää määritellä lämpötila valitun tilan ja käytetyn ajan perusteella.



7. Ohjelmistossa pitää olla operaatio, joka pitää yllä tietoa ajan kulumisesta siitä lähtien, kun Aloita-kytkintä painettiin. Operaation on tarjottava kolme asiaa; kulunut aika 1 sekunnin välein sekä signaali 5 ja 30 sekunnin välein.
8. Joka 30. sekunti asetetaan uusi kohdelämpötila.
9. Joka 5. sekunti ulkoinen sensori tarkkailee lämpötilaa ja vertaa sitä kohdelämpötilaan. Lämpöä tarkkaillaan vähentämällä kohdelämpötilasta nykyinen lämpötila. Jos tulos on positiivinen, asetetaan lämmittäjä asentoon ON. Jos tulos on puolestaan negatiivinen, asetetaan lämmittäjä asentoon OFF.
10. Joka 30. sekunti päivitetään keitto- ja lämmitysmerkkivalot. Keitto- ja lämmitysmerkkivalot kytketään päälle tai pois päältä riippuen kuluneesta ajasta ja valitusta tilasta. Jokaisella tilalla on oma esimääritelty aika, jota käytetään merkkivalojen yhteydessä. Keittovalo palaa esimääritellyn ajan. Muina aikoina se on sammuksissa. Lämmitysvalo palaa esimääritellyn ajan jälkeen keittimen virran katkaisemiseen asti.