**Ohjelmiston laadun parantaminen puutemittareiden avulla**

Teemu Kipponen

17.2.2005

Joensuun yliopisto
Tietojenkäsittelytieteen laitos

Pro gradu -tutkielma

**TIIVISTELMÄ**

Ohjelmiston laadukkuus on erittäin tärkeä tekijä nykypäivän ohjelmistotuotannossa. Ohjelmistojen laatua voidaan parantaa järjestelmällisellä mittaamisella. Puutemittareita käyttämällä ohjelmistojen kehitysprosessia voidaan kehittää laadukkaammaksi, jolloin kehitysprosessin lopputuloksena syntyy laadukas ohjelmisto. Tutkielman tarkoituksena on tarkastella ohjelmistoprosessia, ohjelmistomittausta, ohjelmiston laadun käsitteitä ja puutteiden terminologiaa sekä esitellä kirjallisuuden perusteella tutkimusten mukaan hyviksi havaittuja puutemittareita. Tutkielmassa käydään myös lyhyesti läpi puutteista saatavan mittaustiedon tallentaminen tietokantaan ja tarkastellaan ohjelmiston koon määrittämistä. Ohjelmiston koon laskenta on oleellinen osa ohjelmistomittausta, myös osana laadun mittausta.

**ACM-luokat:** (ACM Computing Classification System, 1998): D.2.8
**Avainsanat:** puutemittarit, ohjelmistomittaus, ohjelmiston laatu, ohjelmistoprosessi, ohjelmiston koko

**SISÄLTÖ**

## 1. JOHDANTO

Ohjelmistomittauksen (software measurement) tärkeyttä on aliarvioitu ohjelmistotuotannon alalla jo pitkään. Ohjelmistojen järjestelmällistä mittaamista ei ole osattu mieltää osaksi hyvää ohjelmistoprosessia, vaan sitä on pidetty erillisenä toimintona ja tavallaan erikoisuutena. Jos ohjelmistojen mittaamista on toteutettu, se on ollut epäsäännöllistä, hajanaista ja epätäydellistä [11]. Ohjelmistojen kehitysprosessia on kuitenkin mahdollista kehittää mittaamisen avulla laadukkaammaksi. Tämä onnistuu vain kontrolloidulla ja järjestelmällisellä mittaamisella, jolloin ohjelmistojen kehitysprosessi voidaan saada paremmin hallintaan. Myöskään ohjelmistoprojektin tilasta ei voida sanoa onko se kunnossa, jos sen kunnollisuutta ei mitata. Vain hallitun kehitysprosessin ja ohjelmistoprojektin lopputuloksena voi syntyä laadukas ohjelmisto.

Ohjelmistomittaus on Fentonin ja Pfleegerin [11] mukaan tärkeää seuraavista syistä: se auttaa *ymmärtämään* mitä ohjelmistokehityksen ja ylläpidon aikana tapahtuu, antaa mahdollisuuden *kontrolloida* mitä projekteissa tapahtuu sekä tukee prosessien ja tuotteiden *parantamista ja kehittämistä*. Ohjelmistomittausta tarvitaan siis ainakin projektien, tuotteiden, prosessien ja resurssien tilan selvittämiseksi [11]. Tämän tarpeen tiivistää melko osuvasti Fentonin ja Pfleegerin [11] mukaan DeMarcon [9] toteamus: "Et voi kontrolloida sitä mitä et voi mitata".

Tässä tutkielmassa ohjelmistomittausta tarkastellaan ohjelmiston puutteiden mittaamisen kautta. Puutteiden mahdollisimman täydellinen poistaminen ohjelmistosta on erittäin tärkeää ohjelmiston laadun kannalta. Ohjelmiston puutteet olisi lisäksi hyvä pystyä löytämään ja poistamaan mahdollisimman varhaisessa kehitysvaiheessa, koska niiden korjaaminen ohjelmiston myöhemmissä kehitysvaiheissa tai valmiista tuotteesta on vaikeampaa ja kalliimpaa. Boehm ja Basili [4] toteavatkin, että ohjelmistojen puutteiden löytäminen ja korjaaminen vasta ohjelmiston toimituksen jälkeen on 100 prosenttia kalliimpaa kuin puutteiden löytäminen ja korjaaminen määrittely- ja suunnitteluvaiheissa. Lisäksi ohjelmiston var-

haisissa kehitysvaiheissa syntyneet puutteet aiheuttavat hyvin usein lisää puutteita myöhemmissä kehitysvaiheissa. Mahdollisten puutteiden löydyttyä ne tulisi pystyä poistamaan mahdollisimman nopeasti ja tehokkaasti mm. kustannusten ja aikataulun takia, koska puutteiden korjaaminen on aina ylimääräistä työtä.

Puutteiden syntymistä on myös mahdollista ehkäistä aiemmin kerätyn puutetiedon avulla. Kerätyn puutetiedon avulla voidaan kiinnittää erityishuomiota ja varata aikaa niihin ohjelmiston kehitysvaiheiden kohtiin, joissa aiemmin on huomattu puutteiden esiintymisen olleen tiheää. Puutteiden syntymisen syitä on myös mahdollista tutkia puutteista kerätyn tietokannan avulla.

Tutkielman perusajatuksena on tarkastella ohjelmiston laadun parantamista puutemittareita käyttämällä ohjelmiston kehitysprosessin eri vaiheissa. Tutkielmassa esiteltävät mittarit on valittu Li'n ja Smidtsin [23] sekä Kan'in [21] perusteella. Artikkelissa esitetyssä tutkimuksessa 10 ohjelmistoalan asiantuntijaa arvioivat 30 ohjelmiston luotettavuuden parantamiseen tarkoitetun mittarin joukosta parhaiten soveltuvia mittareita ohjelmistoprosessin eri vaiheisiin. Ohjelmiston luotettavuus on tietenkin myös yksi tärkeimmistä ohjelmiston kokonaislaadun tekijöistä. Artikkelin pohjalta esitellään yksi tai useampi asiantuntijoitten hyvin soveltuvaksi arvioima mittari ohjelmiston kehitysvaihetta (määrittely, suunnittelu, toteutus ja testaus) kohden.

Tästä tutkielmasta on rajattu pois mittausohjelman tarkastelu. Mittausohjelma tarkoittaa ohjelmistomittauksen projektisuunnitelmaa, jossa määritellään mm. mittauksen tavoitteet, mittaukseen käytettävät resurssit, käytettävät mittarit, mittareiden käyttökohteet ja -tavat, saatujen mittaustulosten käsittely ja -analysointi. Van Solingenin ja Berghoutin [34] kirjassa on käsitelty esimerkkinä mittausohjelmaa luotettavuuden näkökulmasta.

Tutkielman toisessa luvussa tarkastellaan yleisesti ohjelmistoprosessia sekä ohjelmiston laadun käsitettä. Lisäksi luvussa käsitellään ohjelmistojen mittaamista

yleisesti ja ohjelmistojen laadun parantamista mittaamisen avulla. Luvussa esitellään myös lyhyesti ohjelmiston koon määrittelyitä sekä esimerkkejä ohjelmiston koon mittaamisesta. Tutkielman kolmannessa luvussa keskitytään puutteiden määrittelyyn, luokitteluun, mittaustiedon keräämiseen ja käsittelyyn sekä ohjelmiston laatuun puutteiden näkökulmasta. Luvussa esitellään Li'n ja Smidtsin [23] artikkelin pohjalta valitut kehitysvaiheisiin soveltuvat mittarit sekä niiden käyttötapoja. Käyttö- ja ylläpitovaiheen mittareita tarkastellaan Kan'in [21] periaatteiden pohjalta. Neljäs luku on yhteenveto, jossa pohditaan tässä tutkielmassa esiteltyjen mittareiden käytöllä saavutettavia hyötyjä.

# 2. OHJELMISTOPROSESSI JA OHJELMISTON LAATU

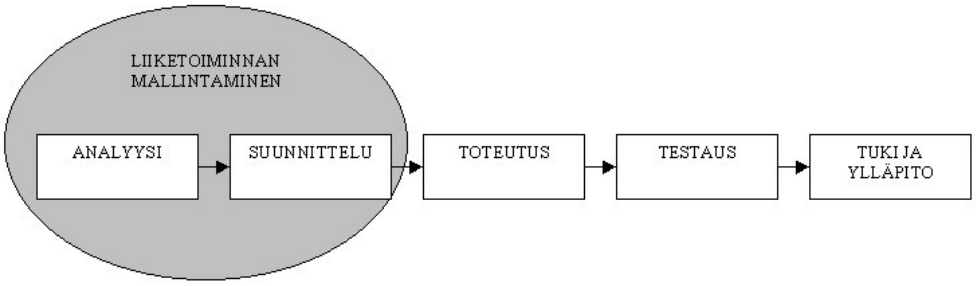
Ohjelmiston laatu on käsite, jolle löytyy hyvin useita määrittelyjä kirjallisuudesta. Ohjelmiston laatu vaihtelee myös näkökulmasta riippuen. Esimerkiksi ohjelmistopäällikkö saattaa pitää laadukkaana sellaista ohjelmistoa, joka on helposti muutettavissa, ylläpidettävissä ja uudelleenkäytettävissä. Tietenkin myös ohjelmiston menestyminen markkinoilla saattaa merkitä ohjelmistopäällikölle ohjelmiston laadukkuutta tai ainakin se merkitsee sitä yrityksen johdolle. Ohjelmiston käyttäjä saattaa taas pitää laadukkaana sellaista ohjelmistoa, joka toimii oikein ja sopii juuri hänen käyttötarkoituksiinsa.

Kohdassa 2.1 käydään suppeasti läpi ohjelmistoprosessin vaiheet prosessimallin avulla myöhemmin käsiteltävien asioiden tueksi. Kohdassa 2.2 tarkastellaan laadukkaan ohjelmiston käsitteitä ja määrittelyjä. Kohdassa 2.3 käsitellään lyhyesti ohjelmiston mittaamisen eri osa-alueita. Lopuksi kohdassa 2.4 käydään läpi ohjelmiston koon määrittelyjä, koska ohjelmiston koon määrittäminen on usein välttämätöntä muita ohjelmistomittareita käytettäessä.

## 2.1 Ohjelmistoprosessi

Floracin ja Carletonin [15] mukaan Pall [29] määrittelee käsitteen *prosessi* eräänlaiseksi loogiseksi ihmisten, materiaalien, energian, välineitten ja työtapojen muodostamaksi organisaatioksi, jonka tarkoituksena on tuottaa tarkasti ennalta määrätty lopputulos. Tämän prosessin määritelmän pohjalta Florac ja Carleton [15] määrittelevät *ohjelmistoprosessin* (software process) kokonaisuutena, johon kaikkien prosessi-määritelmään kuuluvien osatekijöiden lisäksi kuuluvat kaikki ohjelmiston tuottamiseen tarvittavat vaiheistetut tehtävät.

Lineaarinen peräkkäisen kehittämisen malli esitellään kuvassa 1. Se on yksinkertaisin ja vanhin, mutta myös laajimmin ohjelmistotuotannossa käytetty klassinen prosessimalli [31].




Kuva 1. Lineaarinen perättäisen kehittämisen malli [31].

Seuraavaksi käydään läpi kuvan 1 prosessimallin vaiheet ja tehtävät Pressmanin [31] mukaan hieman tarkemmin:

*Liiketoiminnan mallintaminen*. Ohjelmisto on aina osa suurempaa järjestelmää/liiketoimintaa, jonka vuoksi ohjelmistoprosessi voi alkaa liiketoiminnan mallintamisella. Tämän jälkeen vaatimukset kohdennetaan ohjelmistolle. Järjestelmänäkökohta on erittäin tärkeä varsinkin silloin kun ohjelmiston on oltava vuorovaikutuksessa laitteistojen, ihmisten ja tietokantojen kanssa.

*Ohjelmistovaatimusten analysointi*. Vaatimusten kokoamisprosessi on kohdistettu erityisesti ohjelmistoon. Ohjelmistosuunnittelijan täytyy ymmärtää ohjelmiston käsittelemät tiedot, toiminnot, käyttäytyminen, suorituskyky ja vuorovaikutus.

*Suunnittelu*. Ohjelmiston suunnittelu on yleensä neljä eri vaihetta käsittävä prosessi. Suunnitteluprosessin eri vaiheet kohdistuvat seuraaviin ohjelmiston ominaisuuksiin: tietorakenne, ohjelmistoarkkitehtuuri, käyttöliittymä ja algoritmit. Suunnitteluprosessi muodostaa vaatimuksista ohjelmiston kuvauksen, jota voidaan arvioida laadun varmistamiseksi ennen toteutusvaiheen alkua.

*Toteutus*. Toteutusvaiheessa suunnitteluprosessin tulos muunnetaan tietokoneen ymmärtämään muotoon ohjelmakoodiksi.

*Testaus*. Toteutusvaiheessa tehty ohjelmakoodi testataan testausprosessissa. Testausprosessi keskittyy ohjelman sisäiseen logiikkaan ja ulkoiseen toiminnallisuuteen.

*Tuki ja ylläpito*. Lähes jokaista käytössä olevaa ohjelmaa joudutaan muuttamaan asiakkaalle toimituksen jälkeen. Muutosten tarve saattaa aiheutua ohjelmistoon jääneistä virheistä, asiakkaan järjestelmäympäristön muuttumisesta tai asiakkaan muuttuneista ohjelmiston toiminnallisuus- tai tehokkuusvaatimuksista. Olemassaolevalle ohjelmalle muutosta tehtäessä tuki- ja ylläpitotoimet käyttävät uudestaan edellisiä vaiheita.

Kuvan 1 prosessimallissa on suurimpana epäkohtana se, että reaalimaailman ohjelmistoprosessit etenevät harvoin mallin esittämällä tavalla peräkkäisesti ja ohjelmistoon tehtävät muutokset voivat aiheuttaa tässä mallissa sekaannuksia. Lineaarinen peräkkäismalli vaatii myös, että kaikki ohjelmiston vaatimukset asetetaan tarkasti heti prosessin alussa, mikä saattaa usein asiakkaalle olla hyvin vaikeaa.

Pollicen et al. [30] mukaan perinteiset prosessimallit olettavat, että kaikki ohjelmiston kehitykseen tarvittavat tiedot ovat heti alussa tiedossa eikä ohjelmistoon kehityksen aikana tehdä muutoksia. Nykypäivänä ohjelmistoja kehitettäessä niihin joudutaan kuitenkin tekemään lähes jatkuvasti muutoksia kehitysprosessin aikana. Iteratiivinen prosessimalli tarjoaa tukea jatkuvien muutoksien hallintaan kehitysprosessin aikana. Esimerkiksi RUP (Rational Unified Process) -kehitysmallissa projekti on jaettu *iteraatioihin*, joista jokainen sisältää perinteisen prosessimallin vaiheet. Jokaisella iteraatiolla on oma tavoitteensa ja aikataulunsa. RUP-kehitysmallin mukaisen prosessin tavoitteena on tuottaa toimiva ohjelmisto, joka tarjoaa lisäarvoa ohjelmiston käyttäjälle. Jokainen iteraatio lisää edellisissä iteraatioissa tehtyyn työhön oman osansa, jonka johdosta RUP-kehitysmallin mukaisen pro-

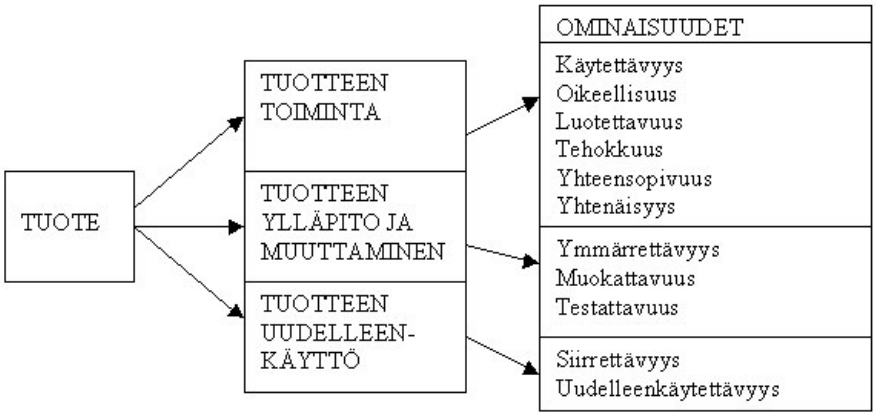
sessin voidaan sanoa olevan sekä *iteratiivinen* että *lisäävä* [30]. Lähtökohtana RUP-prosessissakin voi kuvan 1 mukaisesti olla liiketoiminnan mallintaminen.

### 2.2 Ohjelmiston laadun käsitteet

Kun ohjelmistoprosessia ja –projektia aletaan kehittää laadukkaammaksi, täytyy aivan ensimmäiseksi määritellä *ohjelmiston laatu*. Ongelmallista ohjelmiston laadun mittaamisessa on käsitteen laatu epäselvä määrittely ja täten laadun ymmärtäminen. Syynä huonoon määrittelyyn ja väärinymmärrykseen on laadun moniulotteisuus. Termiä laatu käytetään myös jokapäiväisessä elämässä, jolloin yleinen näkemys ja ammattimainen näkemys saattavat erota toisistaan hyvin voimakkaasti [21].

Ohjelmistotuotannon alalla laatua on kuvattu erilaisten mallien avulla, koska laatu on useiden eri tekijöiden ja niiden vuorovaikutusten yhdistelmä. Daily [8] kuvaa ohjelmiston laatua kuvan 2 mallin mukaan. Mallissa tuotteen eli ohjelmiston käyttö on jaettu kolmeen osa-alueeseen, jotka ovat tuotteen varsinainen käyttö, tuotteen ylläpito ja muuttaminen sekä tuotteen siirto toiseen järjestelmään. Kuvassa ohjelmiston toiminnan laatua kuvaavat ominaisuudet on lueteltu omassa laatikossaan:

- *Käytettävyys*: Voiko ohjelmistoa käyttää sille määriteltyyn tarkoitukseen?
- *Oikeellisuus*: Tekeekö ohjelmisto sitä mitä siltä vaaditaan?
- *Luotettavuus*: Jatkaako ohjelmisto sen tekemistä mitä siltä vaaditaan?
- *Tehokkuus*: Tekeekö ohjelmisto sitä mitä siltä vaaditaan minimiresursseilla (laite-, aika- ja kustannusresurssit)?
- *Yhteensopivuus*: Toimiiko ohjelmisto oikein muiden vaadittujen järjestelmien/ohjelmistojen kanssa?
- *Yhtenäisyys:* Kontrolloidaanko toimivan ohjelmiston käyttöoikeuksia ja sallivatko nämä käyttöoikeudet vain tarvittavat käyttömahdollisuudet?




Kuva 2. Ohjelmiston laatu [8].

Daily [8] asettaa kuvassa 2 ohjelmiston laadulle seuraavat ominaisuudet ohjelmiston ylläpidon ja ohjelmistoon tehtävien muutosten tekemisen kannalta:

- *Ymmärrettävyys*: Onko ohjelmiston toiminta ja rakenne selkeä?
- *Muokattavuus*: Voiko ohjelmistoa muuttaa?
- *Testattavuus*: Voiko ohjelmistoa testata?

Daily [8] asettaa kuvassa 2 ohjelmiston laadulle seuraavat ominaisuudet ohjelmiston uudelleenkäytön kannalta:

- *Siirrettävyys*: Voiko ohjelmisto toimia erilaisissa ympäristöissä (laite-, käyttöjärjestelmäympäristö tms.) ilman merkittävien muutosten tekemistä ohjelmiston vaatimuksiin?
- *Uudelleenkäytettävyys*: Soveltuuko ohjelmisto erilaisiin vaatimuksiin?

Ohjelmiston laadulle löytyy myös useita muita määritelmiä. Fenton ja Pfleeger [11] määrittelevät laadukkaalle ohjelmistolle seuraavia ominaisuuksia: ohjelmisto

soveltuu käyttötarkoitukseensa, ohjelmisto vastaa määrityksiään, ohjelmisto on erinomainen ja ohjelmiston käyttöikä on pitkä. IEEE:n standardissa 610.2 [19] ohjelmiston laatu määritellään:

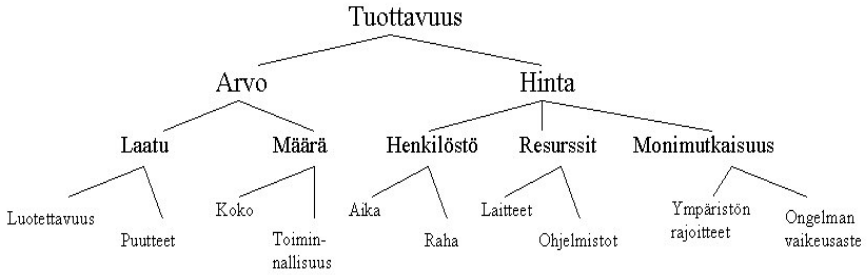
1) Aste (degree), jonka mukaisesti järjestelmä, komponentti tai prosessi täyttää sille asetetut vaatimukset.
2) Aste, jonka mukaisesti järjestelmä, komponentti tai prosessi täyttää asiakkaan tai käyttäjän tarpeet tai odotukset.

### 2.3 Ohjelmistomittaus

Ohjelmistoprosessit ja -projektit voidaan saada kunnolla hallintaan vain järjestelmällisellä ja kontrolloidulla mittaamisella ohjelmistomittareiden avulla. Ohjelmistomittarit (software metrics) kattavat Fentonin ja Pfleegerin [11] mukaan seuraavia ohjelmistomittauksen osa-alueita:

- *Hinta- ja työmääräarviot*: Yritysten johdolle on tärkeää pystyä ennustamaan ohjelmistoprojektin kokonaiskustannukset jo sen varhaisissa kehitysvaiheissa. Ohjelmistoprojektin kustannusten ja työmäärien arvioimiseksi on luotu useita malleja, kuten Boehmin [5] COCOMO-malli, Putnamin [32] SLIM-malli ja Albrechtin [2] toimintopistemalli.

- *Tuottavuusmittarit ja –mallit*: Ohjelmistoyrityksen johdolle on tärkeä tietää yrityksen henkilöstön tuottavuus ohjelmistoprosessien aikana. Kuvassa 3 on esitetty mahdolliset komponentit jotka vaikuttavat kokonaistuottavuuteen. Siinä tuottavuus on kuvattu arvon ja hinnan funktiona. Laatuun vaikuttaa luotettavuus ja puutteet.




Kuva 3. Tuottavuusmalli [11].

- *Mittaustietokanta*: Mittausohjelman [34] laadukkuus riippuu hyvin paljon mittaustietokannasta. Mittaustietokantaa käsitellään enemmän tutkielman kohdassa 3.2.

- *Laatumallit ja –mittarit*: Tuottavuudella ei ole mitään merkitystä, jos tuotteen laatu ei ole hyvä. Tästä syystä on kehitetty laatumalleja, joiden mittaustuloksia voidaan yhdistellä tuottavuusmallien kanssa. Laadun mittaamista puutemittareiden avulla käsitellään luvussa 3 Li'n ja Smidtsin [23] ja Kan'in [21] mukaisesti.

- *Luotettavuusmittarit*: Useimmat laatumallit sisältävät luotettavuuden yhtenä ohjelmiston laadun komponenttina. On kuitenkin ollut tarve ennustaa ja mitata ohjelmiston luotettavuutta myös itsenäisenä osana, jolloin tuotteen ymmärrettävyyttä ja kontrollia voidaan parantaa. Erästä Farrin [10] määrittämää luotettavuusmittaria tarkastellaan kohdassa 3.3.4.

- *Suorituskyvyn arviointimallit*: Suorituskyvyn arviointimalleilla pyritään arvioimaan järjestelmän suorituskykyä, kuten vasteaikoja ja suoritusnopeutta. Näitä malleja on esitellyt mm. Ferrari [12], Ferrari et al. [13] ja Kleinrock [22].

- *Rakenne- ja kompleksisuusmallit*: Ohjelmistoprojekteissa olisi hyvä pystyä ennustamaan mitkä kehitettävän järjestelmän osat tulevat olemaan vähemmän luotettavia, mitkä osat ovat vaikeampia testata tai mitkä osat vaativat enemmän ylläpitoa kuin toiset. Luotettavuutta ja ylläpidettävyyttä ei pystytä mittaamaan ennen kuin toimiva versio ohjelmakoodista on saatavilla. Klassisimmat esimerkit näistä malleista ovat Halsteadin [17] ja McCaben [24] rakenne- ja kompleksisuusmallit. Kohdassa 3.3.2 tarkastellaan McCaben [24] syklomaattista monimutkaisuutta.

- *Kyvykkyyden ja kypsyyden arviointi*: Yhdysvaltain ohjelmistotuotannon instituutti (US Software Engineering Institute, SEI) julkaisi Humphreyn [18] mukaan 1980-luvulla mallin nimeltä *prosessin kypsyyden arviointi*, jolla voitiin mitata ohjelmistotoimittajien kyky tuottaa laadukkaita ohjelmistoja Yhdysvaltojen parlamentille. Malli sisälsi useita ohjelmistokehitykseen liittyviä ominaisuuksia, kuten työkalujen käyttö, standardikäytännöt jne. Mallin ensimmäisessä versiossa ohjelmistotuottajat vastasivat yli sataan kysymykseen, jotka oli suunniteltu siten, että tuottajien todelliset käytännöt saatiin selville. Malli antoi tuloksena ”arvosanan” asteikolla 1-5 ohjelmistotuottajan käytännöistä. Arvo 1 vastaa ”ad hoc” -kehitystä kun taas arvo 5 vastaa kehitysprosessia, jota voidaan optimoida jatkuvaan palautteeseen perustuen. SEI on julkaissut mallista useita uusia versioita, joista uusin on nimeltään *kyvykkyyden- ja kypsyyden arviointi*. Malli perustuu avainkäytäntöihin, joita jokaisen hyvään ohjelmistotuotantoprosessiin pyrkivän ohjelmistotuottajan tulisi käyttää.

- *Johtaminen mittareiden avulla*: Mittaamisesta on tullut tärkeä osa ohjelmistoprojektin hallinnassa ja johtamisessa. Asiakkaat ja kehittäjät haluavat nähdä mittauksien pohjalta luotuja kaavioita ja kuvaajia varmistuakseen siitä, että projekti on aikataulussa ja kunnossa.

- *Menetelmien ja työkalujen evaluointi*: Uusia menetelmiä ja työkaluja markkinoidaan organisaation tai projektin tuottavuuden ja tuotteiden laadun parantamiseksi. On kuitenkin vaikeaa arvioida pelkkien myyntipuheiden perusteella, miten lupaukset toimivat käytännössä. Tästä syystä useat organisaatiot tekevät kokeita ja tapaustutkimuksia todetakseen menetelmän tai työkalun sopivuuden juuri heidän käyttötarkoitukseensa. Tällaiset tutkimukset vaativat tarkkaa ja hallinnoitua mittaamista ja analyysia.

Ohjelmistomittarit voidaan Kan'in [21] mukaan jakaa myös karkeasti seuraaviin luokkiin:

1) Tuotemittarit
    - kuvaavat ohjelmiston ominaisuuksia, kuten koko, monimutkaisuus, suunnittelupiirteet, suorituskyky ja laatu
2) Prosessimittarit
    - kuvaavat ohjelmiston kehitys- ja ylläpitoprosessin tilaa
    - mitataan esim. puutteiden poistamisen tehokkuutta kehityksen aikana
3) Projektimittarit
    - kuvaavat projektin ominaisuuksia ja etenemistä
    - mitataan esim. projektin hintaa, aikataulua ja tuottavuutta.

Kan'in [21] jaottelu vastaa Fentonin ja Pfleegerin [11] jaottelua, jonka mukaan ohjelmistossa on kolme mitattavaa pääkohdetta: *prosessit, tuotteet* ja *resurssit*. Prosessit vaativat resursseja voidakseen tuottaa tuotteita. Kaikilla näillä kohteilla on sekä *sisäiset* että *ulkoiset* ominaisuudet. Sisäiset ominaisuudet voidaan mitata tutkimalla tuotetta, prosessia tai resurssia itsessään erillään sen käyttäytymisestä. Jos ajatellaan esim. ohjelmamoduuleita tuotteina, ilman koodin suorittamista, voidaan määritellä tärkeitä sisäisiä ominaisuuksia, kuten ohjelmamoduulin koko koodiriveinä, kompleksisuus ja yhteydet muihin moduuleihin. Ohjelmamoduulien ulkoisia ominaisuuksia voidaan mitata vain kun ohjelmakoodia suoritetaan. Ul-

koisia ominaisuuksia tässä tapauksessa voisivat olla esim. käyttäjän kohtaamien häiriöiden määrä ohjelmistossa tai aika, joka ohjelmalta kuluu etsiä ja palauttaa haluttu tieto tietokannasta [11].

## 2.4 Ohjelmiston koko

Ohjelmiston koon mittaaminen on välttämätöntä useimpia ohjelmistomittareita käytettäessä. Esimerkiksi kohdissa 3.3.2.1 ja 3.3.3.1 esiteltävät suunnitelman ja koodin puutetiheysmittarit vaativat ohjelmiston koon määrittämistä. Tässä kohdassa esitellään kaksi yleisesti käytettyä tapaa määrittää ohjelmiston koko: koodirivien määrä ja toimintopisteet. Lisäksi käydään lyhyesti läpi COSMIC-FFP -menetelmä ohjelmiston koon määrittämiseksi. Luvussa käsitellään myös liitteen 1 esimerkkisovelluksen koon määrittäminen sekä toimintopistelaskennalla että COSMIC-FFP-menetelmällä.

### *2.4.1 Koodirivien määrä*

Proseduraalisia ohjelmointikieliä, kuten C, käytettäessä koodirivien laskeminen on perinteinen tapa laskea ohjelmiston koko. Koodirivien laskemiskäytännöissä on kuitenkin suuria eroja, jotka johtuvat lähinnä siitä kuinka seuraavassa lueteltaviin ohjelmariveihin suhtaudutaan koodirivien laskemisessa: tyhjät rivit, kommenttirivit, tiedon määrittelyrivit ja rivit, jotka sisältävät useita eri käskyjä. Conte et al. [7] määrittelevät koodirivin (LOC) seuraavasti:

> Koodirivi on jokainen rivi ohjelmatekstissä, joka ei ole kommentti- tai tyhjä rivi, riippumatta siitä, onko rivillä useita ohjelmalauseita tai ohjelmalauseen osia. Koodiriveihin lasketaan myös kaikki ohjelman otsikot (header), määrittelyt, esittelyt ja suorittuvat sekä ei-suorittuvat lauseet.

Määritelmä on Fentonin ja Pfleegerin [11] mukaan laajasti hyväksytty, ja siitä käytetään lyhennystä NCLOC (non-commented lines of code). Kommentoimattomien rivien määrä on järkevä ja hyödyllinen mittauskohde esimerkiksi, kun ohjelmarivien määrää verrataan työmäärään tuottavuuden kannalta. Määritelmän ongelmana on se, että määritelmä NCLOC olettaa, että kommenttien kirjoittamiseen ei kulu lainkaan ohjelmoijien työpanosta, joka ei tietenkään pidä paikkaansa.

Kohdissa 3.3.2.1 ja 3.3.3.1 esiteltävien suunnitelman ja koodin puutetiheysmittareiden vaatimat määritelmät KSLOC ja KDLOC ovat muunnoksia perinteiseen koodirivien laskemiseen. KSLOC tarkoittaa suoritettavien ohjelmarivien ja ei-suoritettavan tiedon esittelyyn käytettävien ohjelmarivien määrää. KSLOD tarkoittaa suunnitteludokumenttien rivimäärää.

### *2.4.2 Toimintopisteet*

Toimintopisteillä (function points, FP) tarkoitetaan ohjelmiston määrittelyissä kuvattua toiminnallisuuden määrää [11]. Ohjelmiston toiminnallisuuden määrän määrittämistä tarvitaan ainakin oliosuuntautuneita ja visuaalisia ohjelmointikieliä, käytettäessä, koska perinteinen koodirivien laskeminen ei anna luotettavia tuloksia näitä ohjelmointikieliä käytettäessä [11]. Väite perustuu siihen, että sovelluskehitin muodostaa suuren osan ohjelmiston toiminnallisuudesta eikä tämän toiminnallisuuden ohjelmarivien määrää pystytä laskemaan.

#### 2.4.2.1 Sovittamaton ja sovitettu toimintopistemäärä

Jotta toimintopisteet (FP) voidaan laskea, täytyy ensin laskea sovittamaton toimintopistemäärä, UFC (unadjusted function point count). Tämä saadaan laskemalla ohjelmistosta seuraavien osien määrä:

- ulkoiset syötteet käyttäjältä: esim. menuvalinnat
- ulkoiset tulosteet käyttäjälle: esim. raportit ja ilmoitukset

- ulkoiset kyselyt käyttäjältä: esim. interaktiiviset syötteet, vaativat vasteen
- ulkoiset tiedostot: konekieliset rajapinnat muihin järjestelmiin
- sisäiset tiedostot: suorituksen aikana tarvittavat sisäiset loogiset tiedostot

Seuraavaksi jokaiselle osalle määritellään monimutkaisuus asteikolla: "helppo", "keskiverto", "vaikea". Painoarvo määritellään jokaiselle osalle erikseen taulukon 1 mukaan. Garmus ja Herron [16] esittävät kirjassaan miten painoarvot määräytyvät.

Taulukko 1. Toimintopisteiden monimutkaisuuden painoarvot [31].

| Parametri | Lukumäärä | | Painoarvo Helppo | Keskiverto | Vaikea | |
|---|---|---|---|---|---|---|
| Ulkoiset syötteet (EI) | | x | 3 | 4 | 6 | |
| Ulkoiset tulosteet (EO) | | x | 4 | 5 | 7 | |
| Ulkoiset kyselyt (EQ) | | x | 3 | 4 | 6 | |
| Ulkoiset tiedostot (EIF) | | x | 7 | 10 | 15 | |
| Sisäiset tiedostot (ILF) | | x | 5 | 7 | 10 | |
| Kaikki yhteensä (UFC) | | | | | | |

Toimintopisteet lasketaan käyttämällä seuraavaa kaavaa:

$$FP = UFC \times TCF \text{ eli}$$
$$FP = UFC \times \left[0.65 + 0.01 \times \sum (F_i)\right] \qquad (1)$$

Kaavassa (1) UFC on taulukossa 1 laskettu painoarvojen summa. Systeemipiirteiden $F_i$ ($i$=1,..,14) summa on osa monimutkaisuuden sovitustekijää (TCF), joka lasketaan ohjelmistolle seuraavien kysymysten ilmaisemien systeemipiirteiden pohjalta [3, 16]:

1) Vaatiiko järjestelmä tietoliikennettä?
2) Onko järjestelmässä hajautettua tiedonkäsittelyä?
3) Onko järjestelmän suorituskyky kriittinen tekijä?
4) Toimiiko järjestelmä raskaasti kuormitetussa käyttöympäristössä?

5) Kuinka korkeita asetetut tapahtumatiheysvaatimukset ovat?
6) Vaatiiko järjestelmä suoraa (online) tiedonsyöttöä?
7) Kuinka onnistunut järjestelmä on loppukäyttäjän työtehon kannalta?
8) Suoritetaanko päivitys reaaliaikaisesti?
9) Onko tiedonkäsittely monimutkaista?
10) Onko koodi suunniteltu uudelleenkäytettäväksi?
11) Kuinka helppoa ohjelmiston muuntaminen ja asentaminen on?
12) Vaatiiko järjestelmä luotettavaa varmistusta ja toipumista?
13) Onko järjestelmä suunniteltu useisiin asennuksiin eri organisaatioissa?
14) Kuinka helppoa muutosten tekeminen järjestelmään on?

Jokaiseen ylläolevaan 14 kysymykseen vastataan käyttäen asteikkoa: 0 (ei tärkeä tai ei sovellettavissa) – 5 (ehdottomasti välttämätön). Vastauksista saatu summa kerrotaan siis vakiolla 0.01 ja lisätään tuloon vakio 0.65 jolloin saadaan TCF. Toimintopisteet saadaan sitten kertomalla UFC-arvo TCF:n arvolla.

#### 2.4.2.2 Esimerkki toimintopistelaskennasta

Tässä esimerkissä lasketaan liitteen 1 esimerkkisovelluksen [14] toiminnallinen koko toimintopisteinä. Taulukossa 2 näkyy sovelluksen toimintojen kuvaukset, joista nähdään kuinka moneen tiedostoon (FTR) ja järjestelmärajan ylittävään tietokenttään (DET) toiminto viittaa.

Taulukko 2. Esimerkkisovelluksen toimintojen kuvaukset.

| Toiminto | Tyyppi | DET | FTR |
|---|---|---|---|
| | | | |
| Lisää asiakas | EI | Nimi, osoite | Asiakas |
| | | | |
| Tulosta lasku | EO | Nimi, osoite, maksumäärä, tuotteiden kokonaismäärä | Asiakas, Tuote |
| | | | |
| Kysely tuotteista | EQ | Sijainti, tila, kuvaus, kuorma | Paikka, Tuote |

Taulukossa 3 näkyy esimerkkisovelluksen toimintojen monimutkaisuus ja toimintopisteet taulukon 2 toimintojen kuvausten ja taulukon 1 painoarvojen perusteella muodostettuna. Tietokenttien (DET) osalta on otettu huomioon järjestelmärajan ylittävien virheilmoitusten ja käyttäjän suorittamien toimintojen osuus Garmusin ja Herronin [16] mukaisesti. Sisäisten tiedostojen tietokenttien lukumäärä vastaa liitteen 1 tietokantaulujen attribuuttien lukumäärää. Kullakin sisäisellä tiedostolla on vain yksi tietuetyyppi (RET).

Taulukko 3. Esimerkkisovelluksen toimintojen toimintopisteet.

| Toiminto | Tyyppi | DET | FTR | RET | Toimintopisteet |
|---|---|---|---|---|---|
| Lisää asiakas | EI | 4 | 1 | | 3 |
| Tulosta lasku | EO | 6 | 2 | | 5 |
| Kysely tuotteista | EQ | 6 | 2 | | 4 |
| Asiakas | ILF | 3 | | 1 | 7 |
| Tuote | ILF | 6 | | 1 | 7 |
| Paikka | ILF | 2 | | 1 | 7 |
| | | | | | 33 |

Seuraavaksi lasketaan sovelluksen monimutkaisuuden sovitustekijä. Systeemipiirteet $F_i$ ($i$=1,..,14) voisivat tässä tapauksessa olla (4+0+0+4+1+3+4+3+3+2+2+3+ 2+3) = 34. Näin ollen voidaan laskea:

$$FP = UFC \times TCF \text{ eli}$$
$$FP = UFC \times \left[0.65 + 0.01 \times \sum (F_i)\right]$$

$$FP = 33 \times \left[0.65 + (0.01 \times 34)\right] = 32{,}67$$

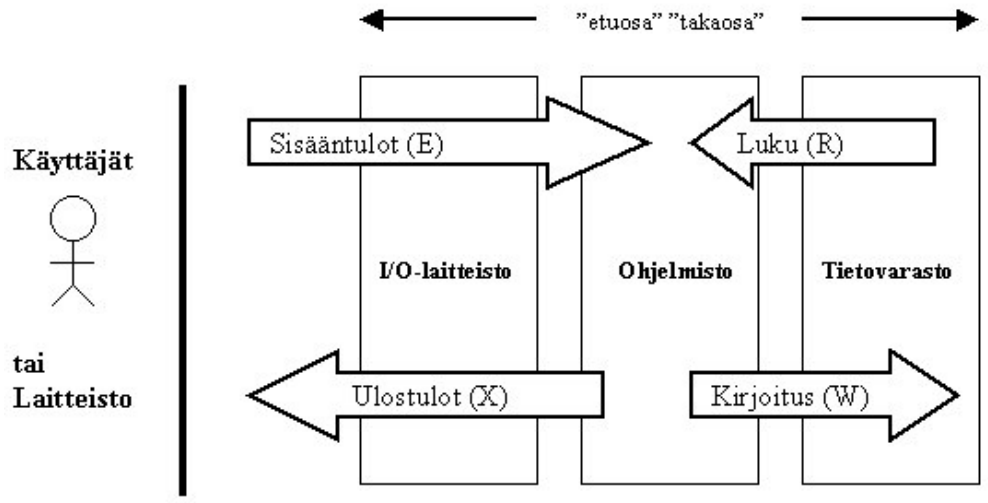
*2.4.3 COSMIC-FFP*

COSMIC-FFP on uudempi ja yksinkertaisempi menetelmä ohjelmiston toiminnallisen koon laskemiseksi kuin toimintopisteanalyysi. COSMIC-FFP jakautuu kahteen vaiheeseen: vaatimusten *kuvausvaihe* (mapping phase) ja *mittaamisvaihe* (measurement phase). Kuvausvaiheessa sovelluksesta muodostetaan käyttäjän

toiminnallisia vaatimuksia (Functional User Requirements, FUR) vastaava ohjelmistomalli (COSMIC-FFP software FUR model), josta sitten selviää sovelluksen toiminnallinen rakenne mittausta varten. Mittaamisvaiheessa ohjelmistomallia käytetään mittauksen suorittamiseksi. Vaiheiden tuloksena syntyy sovelluksen koon ilmoittava numeerinen arvo [1].

2.4.3.1 Laskennan periaate

Kuvasta 4 selviää, mistä COSMIC-FFP:ssä on pohjimmiltaan kysymys. Keskeisenä COSMIC-FFP:ssä on *tiedon siirtäminen* (data movement), joka jaetaan neljään *aliprosessiin*: sisääntulo (entry, E), ulostulo (exit, X), kirjoitus (write, W) ja luku (read, R). Sisään- ja ulostulojen kanssa käyttäjä vaihtaa tietoja sovelluksen kanssa, kun taas kirjoittamisen ja lukemisen avulla tietoja siirretään sovelluksen ja tietovarastojen välillä.




Kuva 4. Tiedon siirtäminen ohjelmistossa toiminnallisesta näkökulmasta [1].

Toiminnallinen prosessi voi koostua yhdestä tai useammasta aliprosessista ja kukin tiedon siirtämisen aliprosessi sisältää täsmälleen yhden *tietoryhmän* (data group). Sovellus voidaan jakaa myös useampaan *kerrokseen*, jolloin mittaaminen

voidaan suorittaa tiedon siirtämisen aliprosessien avulla myös sovelluksen kerrosten välillä [1]. Aliprosessit määritellään Abranin et al. [1] mukaan tarkemmin seuraavasti:

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

Kaikkiin neljään aliprosessiin kuuluu aina tiedon käsittelyä. Jokaisen aliprosessin käsittelemän tietoryhmän tietoattribuuttien joukko on yksilöllinen verrattuna muihin samaan toiminnalliseen prosessiin kuuluviin saman tyyppisiin aliprosesseihin [1].

COSMIC-FFP:n mittaamisvaiheessa käytetty standardimitta on 1 *Cfsu* (Cosmic Functional Size Unit). Jokainen aliprosessi (E, X, R ja W) vastaa yhtä *Cfsu*:ta [1]. Aliprosessien yhteenlaskettu lukumäärä ilmaisee Abranin et al. [1] mukaan sovelluksen (kerroksen) koon kaavan (2) mukaisesti:

$$\text{koko}_{\text{Cfsu}}(\text{kerros}_i) = \sum \text{koko}(E_i) + \sum \text{koko}(X_i) + \sum \text{koko}(R_i) + \sum \text{koko}(W_i) \qquad (2)$$

Sovellukseen (kerroksiin) tehtyjen muutosten koko lasketaan Abranin et al. [1] mukaan kaavalla (3):

$$koko_{Cfsu}(muutos(kerros_i)) = \sum koko(lisätyt\ aliprosessit_i) + \sum koko(muokatut\ aliprosessit_i) + \sum koko(poistetut\ aliprosessit_i) \quad (3)$$

2.4.3.2 Esimerkki ohjelmiston koon laskemisesta COSMIC-FFP:llä

Taulukossa 4 on laskettu liitteen 1 mukaisen esimerkkisovelluksen [14] toiminnallinen koko COSMIC-FFP:n mukaisesti.

Taulukko 4. Esimerkkisovelluksen (liite 1) toiminnallinen koko.

| Nimi | Tiedon siirtymä | Tyyppi | FFP |
|---|---|---|---|
| Lisää asiakas | | | 4 |
| | Käyttäjän syöte | Sisääntulo | 1 |
| | Nimen tarkistus | Luku | 1 |
| | Virheilmoitus | Ulostulo | 1 |
| | Tallenna asiakas | Kirjoitus | 1 |
| Tulosta lasku | | | 5 |
| | Käyttäjän syöte | Sisääntulo | 1 |
| | Hae asiakastiedot | Luku | 1 |
| | Hae tuotetiedot | Luku | 1 |
| | Virheilmoitus | Ulostulo | 1 |
| | Tulosta lasku | Ulostulo | 1 |
| Kysely varastoiduista tuotteista | | | 5 |
| | Käyttäjän syöte | Sisääntulo | 1 |
| | Hae paikkatiedot | Luku | 1 |
| | Virheilmoitus | Ulostulo | 1 |
| | Hae tuotetiedot | Luku | 1 |
| | Näytä tuotetiedot | Ulostulo | 1 |
| | | YHTEENSÄ | 14 |

# 3. PUUTEMITTAREIDEN SOVELTAMINEN

Mittaaminen kokonaisuudessaan voi parantaa ohjelmiston kokonaislaatua, mutta on myös olemassa mittareita, jotka keskittyvät ainoastaan ohjelmiston laatunäkökohtiin. Ohjelmiston laatumittarit ovat niiden ohjelmistomittareiden joukko, jotka keskittyvät tuotteen, prosessin ja projektin laatunäkökohtiin. Laatumittarit voidaan Kan'in [21] mukaan jakaa kolmeen kategoriaan:

- kehitysprosessin laadun mittarit,
- lopputuotteen laadun mittarit ja
- ylläpidon laadun mittarit.

Ohjelmistojen puutteet aiheutuvat erilaisista virheistä ohjelmistoprosessin aikana. Yleisimpiä virheitä ovat määrittelyvaiheessa tapahtuneet väärinymmärrykset ja huolimattomuudet. Virheet ovat hyvin usein inhimillisiä ja yleensä suurimmat puutteet ohjelmiin aiheuttavat nimenomaan analyysivaiheessa tehdyt virheet.

Tässä luvussa käsitellään ohjelmiston laadun parantamista puutteita mittaamalla perustuen Li' n ja Smidtsin [23] sekä Kan'in [21] käsityksiin. Aluksi määritellään ohjelmistojen puutteiden käsitteet sekä käydään läpi miten ohjelmiston puutteista kerättyjä tietoja kerätään, varastoidaan ja käsitellään.

## 3.1 Puutteiden määrittely ja luokittelu

Ohjelmistojen puutteiden termejä on määritelty ja luokiteltu kirjallisuudessa monin eri tavoin. Usein terminologia on varsin sekavaa ja jopa ristiriitaista. Terminologia onkin usein mittaamista tekevissä yrityksissä mennyt usein sekaisin. Terminologian yhtenäisyys olisi äärimmäisen tärkeää varsinkin mittaamista aloitettaessa, jotta kaikki puutteiden mittaamiseen osallisena olevat puhuisivat samaa kieltä.

Usein suomen kielessä, kuten tässäkin tutkielmassa, ohjelmiston kaikista ongelmista käytetään yhteistä termiä *puute*. IEEE:n standardissa 982.2 [20], joka ohjelmiston ongelmien termien osalta esitellään seuraavassa, termi *puute* (defect) määritellään poikkeamana ohjelmistossa. Määritelmän perusteella termin puute voidaankin siis ajatella kattavan kaikki erilaiset ohjelmistojen ongelmatyypit.

IEEE:n standardissa 982.2 [20] ohjelmistojen puutteet määritellään seuraavasti:

- *Puute* (defect): Poikkeama tuotteessa. Esimerkiksi
  1) laiminlyönnit ja epätäydellisyydet varhaisissa kehitysvaiheissa ja
  2) viat testaukseen tai käyttöön hyväksytyssä ohjelmistossa.
- *Virhe* (error): Ihmisen toiminto, jonka seurauksena ohjelmistoon syntyy vika. Esimerkiksi laiminlyönnit ja epätäydellisyydet vaatimusmäärittelyssä.
- *Häiriö* (failure):
  1) Tila, jossa toiminnalliselta yksiköltä eli esim. ohjelmamoduulilta loppuu kyky suorittaa vaadittu toiminto.
  2) Tapahtuma, jossa järjestelmä tai järjestelmän osa ei suoriudu vaadituista toiminnoista esim. annetussa aikarajassa. Häiriö voi syntyä kun vika kohdataan.
- *Vika* (fault):
  1) Odottamaton tila, joka aiheuttaa toiminnalliselta yksiköltä eli esim. ohjelmamoduulilta vaaditun toiminnon epäonnistumisen.
  2) Virheen ilmentymä ohjelmistossa. Vika voi aiheuttaa häiriön.

Fenton ja Pfleeger [11] määrittelevät ohjelmiston puutteiden terminologian yhdenmukaisesti IEEE:n [20] kanssa kuvan 5 mukaisesti tulkittuna.


Kuva 5. Ohjelmiston puutteiden terminologia.

*Vika* aiheutuu ohjelmistoon siis inhimillisestä *virheestä*. Vika on ihmisen virheen ilmentymä ohjelmistossa. Esimerkiksi suunnittelija saattaa ymmärtää väärin käyttöliittymävaatimukset ja tehdä väärinymmärryksen sisältävän suunnitelman. Tämä suunnitteluvirhe aiheuttaa myös viallisen ohjelmakoodin ja sitä myöten viallisen ohjelmiston, kuten myös viallisen käyttöohjeen. Yksittäinen inhimillinen virhe voi siis aiheuttaa monta vikaa ohjelmistoon. Viat voivat aiheuttaa tietyissä olosuhteissa *häiriön*. Häiriön tapahtuessa ohjelmiston toiminta poikkeaa sen vaaditusta toiminnasta. Häiriöitä voi esiintyä missä tahansa vaiheessa ohjelmiston kehitystä ja usein vasta ohjelmiston toimituksen jälkeen. Kaikki viat eivät johda aina häiriöihin, koska olosuhteita, joita tarvittaisiin häiriön syntymiseen, ei ehkä koskaan tule (esim. vian sisältämää koodinosaa ei koskaan suoriteta) [11]. Vikoja ja häiriöitä voidaan ajatella myös systeemin sisäisinä ja ulkoisina 'ominaisuuksia'. Viat ovat ongelmia, jotka sovelluskehittäjät havaitsevat, kun taas häiriöt ovat ongelmia, jotka käyttäjät havaitsevat.

Lisäksi Fenton ja Pfleeger [11] määrittelevät termit *ohjelmavirhe* (bug), *romahdus* (crash) ja *poikkeama* (anomaly). Termi ohjelmavirhe tarkoittaa koodissa ilmenevää vikaa. Termi romahdus tarkoittaa häiriön erikoistyyppiä, jossa järjestelmä menee tilaan, jossa se lakkaa toimimasta kokonaan. Termi poikkeama tarkoittaa yleensä vikaa, joka ei todennäköisesti aiheuta häiriöitä itsessään, mutta saattaa joskus aiheuttaa häiriöitä epäsuorasti. Esimerkiksi hyvän ohjelmointitavan vastainen muuttujien nimeäminen mielletään usein poikkeamaksi, koska esim. myö-

hemmässä vaiheessa ohjelmaan muutosta tehtäessä huonosti nimetyt muuttujat voivat aiheuttaa väärinymmärryksiä ja aiheuttaa siten häiriöitä.

## 3.2 Puutetiedon kerääminen ja tallentaminen

Ohjelmistoissa löydetyistä ja havaituista puutteista on syytä kerätä tietoja ja tallentaa ne mittaustietovarastoon. Tietovarasto voi yksinkertaisimmillaan olla manuaalinen kortisto, mutta suositeltavaa olisi perustaa tietovarasto tietokantaan. Tietojen ollessa tietokannassa niiden käsittely ja analysointi helpottuu huomattavasti.

### *3.2.1 Puutetiedon kerääminen*

Kun puute havaitaan ohjelmistossa, siitä täytyy tallettaa tärkeimmät tiedot puutetyyppikohtaisille lomakkeille, jotta puutteen syitä ja sen seurauksia voidaan myöhemmin tutkia. Tärkeimmät kerättävät tiedot puutteesta ovat Fentonin ja Pfleegerin [11] mukaan:

- sijainti (missä puute esiintyi)
- aika (milloin puute esiintyi)
- oire (mitä havaittiin)
- lopputulos (mitä tapahtui)
- mekanismi (miten puute esiintyi)
- syy (miksi ongelma esiintyi)
- vakavuus (paljonko puute vaikuttaa käyttäjään)
- hinta (kuinka paljon puute maksoi)

Seuraavaksi käydään läpi puuteraporttien tyypit ja raporttityyppikohtaisesti kerättävät tiedot Fentonin ja Pfleegerin [11] mukaan. *Häiriöraportti* keskittyy systeemin ulkoisiin puutteisiin, siis niihin puutteisiin jotka useimmiten ohjelmiston

käyttäjät havaitsevat. Häiriöraporttiin voidaan Fentonin ja Pfleegerin [11] mukaan kerätä seuraavia tietoja:

- Sijainti: esim. ohjelman tietty asennus, missä häiriö havaittiin
- Aika: esim. prosessoriaika tai kellonaika
- Oire: virheilmoituksen tyyppi tai häiriön ilmenemistapa
- Lopputulos: häiriön kuvaus, esim. "käyttöjärjestelmän romahdus", "palvelun alentuminen", "tiedon katoaminen", "väärä tulostus" tai "ei tulostusta"
- Mekanismi: tapahtumaketju, joka johti häiriöön (sisältäen näppäinkomennot ja tilatiedon)
- Syy: viittaus mahdolliseen vikaan, joka johti häiriöön
- Vakavuus: viittaus asteikkoon, esim. "kriittinen", "tärkeä", "ei-tärkeä"
- Hinta: häiriön korjaamisen hinta

Syy-kategoria kannattaa ristiviitata vika- ja muutosraportteihin, jolloin raporttikokoelma antaa ilmenneestä puutteesta koko kuvan siten, että häiriöraportti kertoo *mitä* tapahtui, vikaraportti kertoo *mikä* sen aiheutti ja muutosraportit kertovat *miten* se korjattiin. Häiriöstä pystytään tallettamaan sen tapahtuessa vain sijainti, aika, oire, lopputulos ja vakavuus. Muut tiedot saadaan vasta diagnoosin jälkeen. Puutetiedon keräyslomakkeen pitää sisältää vähintään nämä viisi kategoriaa.

*Vikaraporttiin* kerätään tietoa systeemin sisäisissä osissa ilmenneestä viasta. Vikaraporttiin kerätään tietoa viasta yksittäisen moduulin tarkkuudella sekä vian paikantamisen ja sen korjaamisen hinnasta. Viat ovat puutteita, jotka yleensä löydetään tarkastusmenettelyin kehitysvaiheessa. Vikaraportti on rakenteeltaan hyvin samankaltainen kuin häiriöraportti, mutta siinä samoihin kysymyksiin saadaan täysin erilaisia vastauksia. Fentonin ja Pfleegerin [11] mukaan vikaraporttiin kerätään seuraavia tietoja:

- Sijainti: moduuli, dokumentin nimi

- Aika: kehitysvaiheet, jossa vika syntyi, havaittiin ja korjattiin
- Oire: virheilmoituksen tyyppi tai vian paljastanut toimenpide (esim. katselmointi)
- Lopputulos: vian aiheuttama häiriö
- Mekanismi: kuinka vika luotiin, havaittiin ja korjattiin
- Syy: ihmisen tekemän virheen tyyppi, joka johti vikaan
- Vakavuus: viittaus viasta seuranneeseen tai potentiaalisen häiriön vakavuuteen
- Hinta: vian paikallistamiseen ja korjaamiseen kuluva aika tai työpanos

Kun häiriö on kohdattu ja sen syy on selvitetty, puute korjataan tekemällä muutoksia tuotteeseen. Muutokset voivat olla muokkauksia yhteen tai kaikkiin kehityksen aikaisiin tuotteisiin sisältäen määrittelyt, suunnitelmat, ohjelmakoodit, testitapaukset, testitulokset ja dokumentit. *Muutosraporttia* käytetään tehtyjen muutosten kirjaamiseen ja se kertoo miten vian aiheuttama häiriö korjattiin. Muutosraporttiin kerätään Fentonin ja Pfleegerin [11] mukaan seuraavia tietoja:

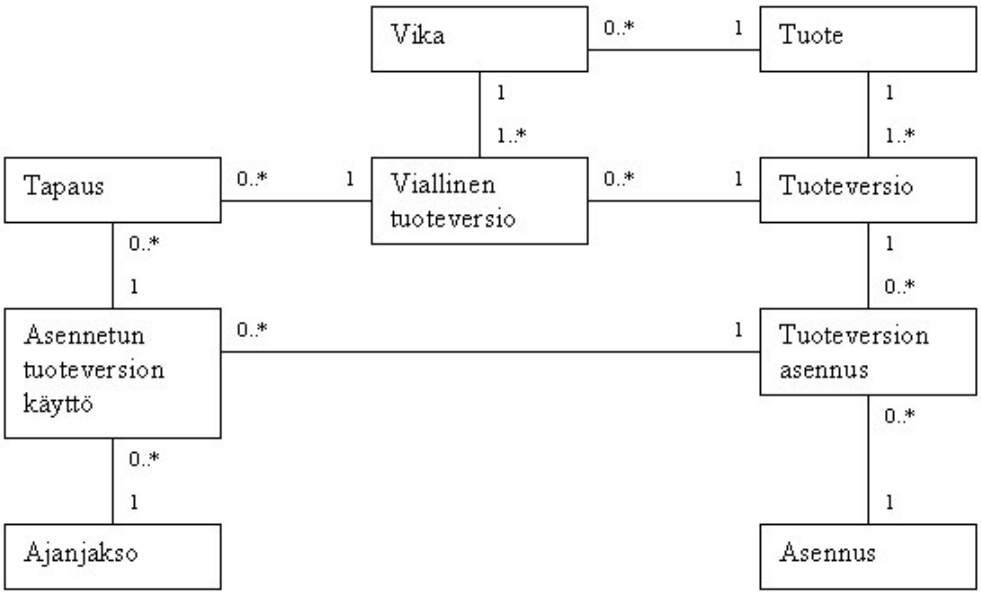
- Sijainti: muutetun dokumentin tai moduulin tunniste
- Aika: milloin muutos tehtiin
- Oire: muutoksen tyyppi
- Lopputulos: muutoksen onnistuminen, todetaan esim. taantumatestauksella
- Mekanismi: miten ja kuka muutoksen teki
- Syy: korjaava, mukauttava, ehkäisevä tai täydentävä
- Vakavuus: kuinka paljon muutos vaikuttaa systeemiin
- Hinta: muutoksen tekemiseen ja testaamiseen kulunut aika ja työpanos

### *3.2.2 Puutetiedon tallentaminen*

Puutetiedot kannattaa tallentaa tietokannanhallintajärjestelmää (DBMS, database management system) käyttävään tietokantaan. Tietokannanhallintajärjestelmän suurimpina etuina paperilomakkeisiin ja rakenteettomiin tiedostoihin verrattuna on kieli, jolla voidaan määritellä tietorakenteet ja tiedon lisäys-, muokkaus- ja poisto-operaatiot. Rajoitteet, kuten viittausten tarkistukset tietueitten välillä, voidaan määritellä tiedon eheyden varmistamiseksi. Myös syötteiden muodot, arvoalueet ja sallitut arvot voidaan tarkistaa automaattisesti jo tietoja syötettäessä [11].

Fentonin ja Pfleegerin [11] mukaan useissa tutkimuksissa, kuten Comerin et al. [6] tutkimuksessa, on yritetty kehittää kaikkiin käyttötarpeisiin soveltuvaa tietomallia ohjelmistotuotannossa syntyvälle mittaus- ym. datalle. Vaarana tällaisen mallin kehittämisessä on, että siitä tulee liian laaja ja monimutkainen. Fentonin ja Pfleegerin [11] mukaan kaikissa kirjallisuudessa raportoiduissa projekteissa on ollut käytössä sovellettu tietomalli, joka sopii tietylle tietojoukolle.

Kuva 6 esittää tietokantarakenteen, joka tukee ohjelmiston luotettavuuden mittaamista. Kuvaus on muokattu Fentonin ja Pfleegerin [11] käyttämästä notaatiosta UML-notaation [28] edellyttämään muotoon.




Kuva 6. Esimerkki puutetietokantakaavasta [11].

Tuote-taulussa on mitattavat tuotteet ja Tuoteversio-taulussa on kaikki tuotteen eri versiot. Asennus-taulu sisältää kaikki asennukset, missä tiettyä tuoteversiota ajetaan. Tuoteversion asennus -tauluun tallennetaan tieto tuotteen versiosta asennuksessa. Ajanjakso-taulussa on tallennettuna kalenteriajan peräkkäisten ajanjaksojen alku- ja loppuajat, jona aikana tuotetta on määrä mitata. Asennetun tuoteversion käyttö -tauluun tallennetaan jokaisen tuoteversion käytön ajanjakson alku- ja loppuaika kalenteriaikana sekä käyttöaika kyseisenä ajanjaksona. Vika-tauluun tallennetaan kaikki yhdestä tai useammasta tuoteversiosta löytyneet viat. Viallinen tuoteversio -taulu sisältää vikatunniste-tuoteversio pareja. Tapaus-tauluun tallennetaan kaikki viat, jotka havaittiin tiettyä tuoteversiota käytettäessä tietyssä asennuksessa.

Kun puutetietokanta on suunniteltu ja sinne on viety kerättyä dataa, voidaan puutetietoja analysoida. Oletetaan, että meillä on kerättynä luotettavuusdataa kuvan 6

mukaiseen tietokantaan. Jos on tarve saada selville yksittäisen tuoteversion luotettavuus kaikissa sen asennuksissa, voidaan noudattaa seuraavia askelia:

1. Valitse jokainen **tapaus** joka viittaa kyseiseen **tuoteversioon.**
2. Ryhmittele valitut **tapaukset** niiden viittaamien **vikojen** perusteella ja lajittele jokainen ryhmä esiintymisajan perusteella.
3. Poista kaikki muut paitsi ensimmäinen **tapaus** jokaisesta ryhmästä.
4. Laske jäljelle jääneiden **tapausten** lukumäärä jokaisessa **ajanjaksossa.**
5. Laske asennetun tuoteversion käytön kokonaismäärä kullekin ajanjaksolle.

Tuloksena saadaan lista numeropareista: tietyssä tuoteversiossa ensiksi havaittujen erilaisten vikojen lukumäärä ja kyseisen tuoteversion käytön kokonaismäärä kunakin ajanjaksona. Tulos ilmentää tuotteen stabiilisuutta. Jos haluttaisiin suhteuttaa vikojen lukumäärä tuotteen kokoon, pitäisi kerättyjen vikojen lukumäärä jakaa tuotteen koolla, esimerkiksi toiminnallista kokoa käyttäen.

### 3.3 Ohjelmiston kehitysvaiheisiin parhaiten soveltuvat mittarit

Tässä kohdassa esitellään Li' n ja Smidtsin [23] artikkelin pohjalta valitut mittarit. Kutakin ohjelmiston kehitysvaihetta kohden esitellään yksi tai kaksi asiantuntijoiden mielestä hyvin kyseiseen kehitysvaiheeseen soveltuva mittari.

Li' n ja Smidtsin [23] artikkelissa 10 tiukoin kriteerein valittua ohjelmistoalan asi antuntijaa arvioivat ohjelmistomittareiden soveltuvuutta ohjelmistoprosessin eri vaiheisiin. Asiantuntijoiden valintakriteereinä oli mm. kokemus ja tehtyjen tutkimusten monipuolisuus. Pääpaino artikkelissa oli ohjelmiston luotettavuuden lisääminen, joka on myös yksi tärkeimpiä ohjelmiston kokonaislaadun tekijöitä.

Mittareiden valintakriteereinä artikkelissa olivat seuraavat ominaisuudet:

- *hinta*: mittarin käyttöönoton ja käyttämisen vaatima työpanos
- *hyöty*: mittarin käytöllä saavutettava ajansäästö
- *uskottavuus*: täyttääkö mittari sille asetetut tavoitteet
- *käyttökokemukset*: mittarin käytön laajuus teollisuudessa
- *toistettavuus*: mittarin tulosten pysyvyys samana, vaikka mittausta suorittavat henkilöt vaihtuvat
- *hyväksyntä*: mittarin hyväksymisen taso ohjelmistotuotannon alalla
- *sopivuus luotettavuuden arviointiin*: mittarin soveltuvuus luotettavuuden arviointiin.

Taulukossa 5 on esitetty asiantuntijoiden arvioimat kolme parhaiten soveltuvaa mittaria vaihetta kohden.

Taulukko 5. Kolme parhaiten soveltuvaa mittaria per vaihe [23].

| **Analyysi** | **Suunnittelu** | **Toteutus** | **Testaus** |
|---|---|---|---|
| Vikatiheys | Suunnitelman puutetiheys | Koodin puutetiheys | Häiriösuhde |
| Määrittelyihin kohdistuvat muutospyynnöt | Syklomaattinen kompleksisuus | Suunnitelman puutetiheys | Koodin puutetiheys |
| Virheiden jakauma | Vikatiheys | Syklomaattinen kompleksisuus | Kattavuusaste |

Kuten taulukosta 5 huomataan, esiintyy samoja mittareita useissa vaiheissa. *Vikatiheys*-mittari soveltuu hyvin sekä vaatimusmäärittely- että suunnitteluvaiheeseen. Myös sekä *syklomaattinen kompleksisuus*- että *suunnitelman puutetiheys* –mittarit on arvioitu hyvin soveltuvaksi sekä suunnittelu- että toteutusvaiheeseen. Myös *koodin puutetiheys* –mittari soveltuu hyvin sekä toteutus- että testausvaiheeseen.

Seuraavaksi esitellään Li' n ja Smidtsin [23] artikkelin mukaan kehitysvaiheisiin parhaiten soveltuvina puutemittareina *vikatiheys, suunnitelman puutetiheys, koodin puutetiheys*- ja *häiriösuhde*. Lisäksi esitellään suunnittelu- ja toteutusvaihee-

seen sopiva *syklomaattinen kompleksisuus*- ja määrittelyvaiheeseen sopiva *määrittelyihin kohdistuvat muutospyynnöt* –mittarit.

### *3.3.1 Määrittelyvaihe*

#### 3.3.1.1 Vikatiheys

*Vikatiheys* (fault density) -mittari ilmaisee ohjelmiston vikatiheyden vakavuusasteineen. Määrittelyvaiheen vikatiheyttä voidaan verrata aiemmissa samankaltaisissa projekteissa saatuihin vikatiheyksiin. Vikatiheys-mittaria voidaan käyttää myös suunnitteluvaiheessa. Mittaria voidaan Smidtsin ja Li' n [33] mukaan IEEE:n standardin 982.2 [20] mukaisesti käyttää seuraaviin toimintoihin:

1) Jäljellä olevien vikojen määrän ennustamiseen vertaamalla odotettuun vikatiheyteen.
2) Ennalta asetettujen vakavuusastetavoitteiden pohjalta suoritettavan testaamisen riittävyyden toteamiseen.
3) Perusvikatiheyksien muodostamiseen vertailuja ja ennustuksia varten.

Vikatiheysmittarin käyttö vaatii seuraavat toimenpiteet [33]:

1) Vakavuusasteiden määrittäminen sekä häiriö- ja vikatyyppien muodostaminen. Häiriötyyppi voi olla syöttö-, tulostusoperaatio tai molemmat sekä käyttäjä. Vikatyyppi voi olla: suunnittelu, koodaus, dokumentaatio tai käyttöönotto.
2) Kirjaa ylös jokainen häiriö.
3) Selvitä ohjelmaviat, jotka aiheuttivat häiriön ja luokittele viat tyypin mukaan.
4) Selvitä suorittuvan ohjelmakoodin ja ei-suorittuvien määrittelyjen kokonaisrivimäärä tuhansina riveinä (*KSLOC*).
5) Laske vikatiheys annetuille vakavuusasteille seuraavalla kaavalla (4):

$$F_d = F / KSLOC \quad (4)$$

jossa $F$ on annetulla aikavälillä löydettyjen ja häiriöön johtaneiden vikojen lukumäärä.

Jos KSLOC-arvoa ei voida laskea, voidaan siinä tapauksessa käyttää jotain muuta normalisointitekijää, esimerkiksi määrittelydokumenttien rivimäärää kokomittarina. Mahdollista on myös toiminnallisen koon laskenta kohtien 2.4.2 ja 2.4.3 mukaisesti.

Vertaamalla kehitettävän järjestelmän vikatiheyksiä aiemmin tehtyihin samankaltaisten järjestelmien vikatiheyksiin, voidaan saada selville onko nykyistä järjestelmää testattu riittävästi. Vikatiheys-mittarin käyttöönotto ei vaadi juurikaan koulutusta, mutta aikaisempaa kokemusta ohjelmistojen testaamisesta tarvitaan vika- ja virheluokittelujen muodostamiseksi.

Esimerkki vikatiheyden laskemisesta:

Häiriöiden lukumäärä on 21, joista saatujen yksilöllisten vikojen määrä on F=29 ja lähdekoodin rivimäärä on 6000. Vikatiheys on siis

$F_d = F / KSLOC$ = 29/6 = 4.8 vikaa/KSLOC.

3.3.1.2 Määrittelyihin kohdistuvat muutospyynnöt

Määrittelyvaiheen toisena mittarina esitellään *vaatimusmäärittelyihin kohdistuvat muutospyynnöt* (requirements specification change requests) –mittari, vaikka se ei varsinainen puutemittari olekaan. Toki muutospyynnöt usein johtuvat puutteista

vaatimusmäärittelyissä. Möllerin ja Paulishin [27] mukaan mittarin määrittely on yksinkertaisesti:

- vaatimusmäärittelyihin kohdistuneiden muutospyyntöjen lukumäärä.

Muutospyynnöt lasketaan vaatimusmäärittelydokumenttien ensimmäisten versioiden ja ohjelmiston julkaisun väliseltä ajalta.

Jatkuvasti muuttuvat vaatimusmäärittelyt ovat suurin syy ohjelmistoprojektien häiriöihin [27]. Tämän vuoksi vaatimusmäärittelyjen versionhallinta sekä vaatimusmäärittelyihin kohdistuvien muutoksien hallittu tekeminen ovat tärkeitä tekijöitä ohjelmistoprojektin laadun kannalta. Tätä helppokäyttöistä mittaria voidaan käyttää vaatimusmäärittelyjen muutoksien määrien vertaamiseen edellisten projekteihin muutoksien määriin. Näin voidaan saada selville, onko määrittelyvaiheen työskentelyssä edistytty.

### *3.3.2 Suunnitteluvaihe*

#### 3.3.2.1 Suunnitelman puutetiheys

*Suunnitelman puutetiheys* (design defect density)- mittarilla voidaan osoittaa ovatko suunnitelman tarkastusmenettelyt tehokkaita [33]. Tarkastusmenettelyt vaativat lisätarkkuutta, jos puutetiheys on asetettua normia korkeampi usean tarkastuskierroksen jälkeen.

Suunnitelman puutetiheysmittarin käyttö vaatii seuraavat toimenpiteet [33]:

1) Määritä puutteiden vakavuusasteet.
2) Muodosta luokittelujärjestelmä puutteen vakavuutta ja luokkaa varten.
3) Kirjaa jokaisessa tarkastuksessa tuotteen koko ja löytyneiden puutteiden kokonaismäärä.

4) Laske suunnitelman puutetiheydelle suhde seuraavalla kaavalla (5):

$$DD = \frac{\sum_{i=1}^{I} D_i}{KSLOD} \qquad (5)$$

jossa

$D_i$ = tarkastuskierroksella i havaittujen suunnitelman puutteiden lukumäärä

$I$ = tarkastusten lukumäärä

$KSLOD$ = suunnitteludokumenttien kokonaisrivimäärä tuhansina riveinä.

Esimerkki puutetiheyden laskemisesta:

I = 7

KSLOD = 8, eli

$$\sum_{i=1}^{7} D_i = 78$$

$$DD = \frac{78}{8} = 9.8$$

Puutetiheys on helppo laskea ja sen avulla voidaan arvioida tarkastusmenettelyprosessin tehokkuutta. Mittarin tuloksissa on kuitenkin hieman tulkinnanvaraa. Esimerkiksi alhainen puutetiheys-arvo voi merkitä hyvää prosessia ja tuotetta tai se voi merkitä huonoa prosessia. Tarkastusmenettelyprosessia pitää tutkia, jos puutetiheys-arvo on alhainen verrattuna aiempiin samanlaisiin projekteihin. Jos tarkastusmenettely todetaan samankaltaiseksi kuin aiempiin projekteihin verrattuna, voidaan kehitysprosessin todeta tuottaneen hyvin vähän puutteita sisältävän tuotteen. Jos puutetiheys on liian suuri verrattuna edellisiin samanlaisiin projekteihin, tällöin pitää tutkia kehitysprosessia ja selvittää onko huonon ohjelmistotuotteen takana huonot ohjelmointitavat. Tällaisessa tapauksessa puutteet kannattaa jakaa kategorioihin puutteen tyypin mukaan. Tällainen jaottelu voisi olla esim.

suunnittelu, logiikka, standardit, käyttöliittymä, kommentit ja vaatimukset. Tällöin voidaan nähdä, missä puutteita esiintyy eniten ja tehdä tarvittavat toimenpiteet sen perusteella. Esimerkiksi jos puutetiheys on suuri vaatimuksissa, on syytä keskeyttää ohjelmiston kehitys kokonaan kunnes vaatimukset on korjattu. Jos kehitysprosessi todetaan vastaavaksi kuin aiemmin, silloin tarkastusmenettelyprosessi voidaan todeta tehokkaaksi.

Puutetiheysmittarin käyttö edellyttää tietämystä suunnitelman ja koodin tarkastusmenettelyprosesseista. Se vaatii myös yhtenäistä puutetietokantaa ja tietojen kirjaamista. Tarkastusmenettelyn pitäisi osoittautua tehokkaaksi, jos siihen sitoudutaan täysin ja se saa johdon tuen. Mitä aikaisemmin tarkastusmenettelydataa on saatavilla, sitä suurempi on saatava hyöty.

#### 3.3.2.2 Syklomaattinen monimutkaisuus

Suunnittelu- ja toteutusvaiheeseen sopivana mittarina esitellään *syklomaattinen monimutkaisuus* (cyclomatic complexity)–mittari, jonka kehitti McCabe [24] vuonna 1976. Mittarin tarkoituksena on osoittaa ohjelmiston testattavuus, ymmärrettävyys ja ylläpidettävyys. Se ei varsinaisesti ole puutemittari, mutta sen käytön avulla ohjelmisto voidaan saada selkeämmin ymmärrettäväksi ja paremmin testattavaksi. Selkeästi ymmärrettävissä olevassa ja paremmin testattavassa ohjelmistossa puutteita esiintyy varmasti vähemmän kuin monimutkaisessa ja vaikeasti testattavassa ohjelmistossa. Mittari pohjautuu klassiseen graafiteorian mukaiseen malliin. Syklomaattinen numero tarkoittaa lineaarisesti riippumattomien polkujen lukumäärää ohjelmassa. Ohjelman proseduurit/moduulit esitetään vahvasti kytkettyinä graafeina yksiselitteisillä sisään- ja ulostulopisteillä. Yleinen kaava syklomaattisen numeron laskemiseen on:

$$M = v(G) = e - n + 2p \qquad (6)$$

jossa

$v(G)$ = proseduurin $G$ syklomaattinen numero

$e$ = kaarien lukumäärä kaaviossa

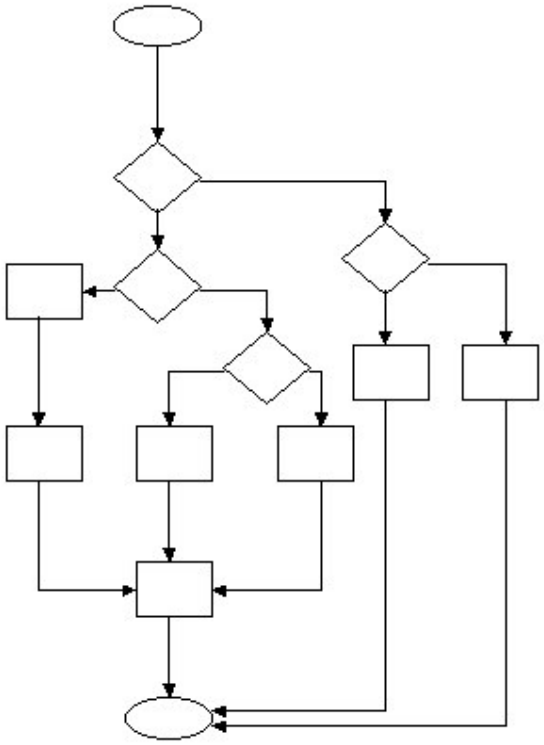
$n$ = solmujen lukumäärä kaaviossa

$p$ = kytkemättömien osien määrä kaaviossa.

Syklomaattinen kompleksisuus voidaan laskea myös yksinkertaisemmin, jos $d$ on haarautumissolmujen lukumäärä $G$:ssä niin

$$v(G) = d + 1 \qquad (7)$$

Esimerkkinä syklomaattisen numeron eli $v(G)$:n laskemisesta lasketaan kuvan 7 ohjausvuokaaviona kuvaaman proseduurin syklomaattinen numero.



Kuva 7. Esimerkki syklomaattisen numeron laskemisesta [11].

Kuvan 7 esimerkin kaaviossa on 16 kaarta ($e$=16), 13 solmua ($n$=13) ja 1 kytkemätön osa ($p$=1), joten kaavalla (6) tai (7) proseduurin syklomaattiseksi numeroksi eli $v(G)$:ksi saadaan 5.

McCaben [24] mukaan moduulin syklomaattinen numero ei saisi ylittää arvoa 10. Jos arvo ylittää arvon 10, moduuli pitää suunnitella uudestaan syklomaattisen numeron laskemiseksi. McCaben mielestä moduulit, joilla syklomaattinen numero on korkea, tulevat hyvin todennäköisesti sisältämään puutteita ja ovat myös huonosti ylläpidettäviä.

Syklomaattinen numero ei kuitenkaan Fentonin ja Pfleegerin [11] mielestä ole yksinään luotettava ohjelmiston monimutkaisuuden mittari. On nimittäin olemassa ohjelmia, joissa on suuri määrä haarautumissolmuja, mutta jotka silti ovat helppoja ymmärtää, koodata ja ylläpitää. Kuitenkin syklomaattinen numero on hyödyllinen mittari ohjelman tai moduulin testattavuuden ja ylläpidon vaikeuden selville saamiseksi. Kan'in [21] mukaan useat testauksen ammattilaiset suosittelevat McCaben mittarin käyttöä testauksen riittävän kattavuuden toteamiseksi.

### *3.3.3 Toteutusvaihe*

#### 3.3.3.1 Koodin puutetiheys

*Koodin puutetiheys* (code defect density)- mittarilla voidaan osoittaa ovatko ohjelmakoodin tarkastusmenettelyt tehokkaita [33]. Tarkastusmenettelyt vaativat lisätarkkuutta, jos puutetiheys on asetettua normia korkeampi usean tarkastuskierroksen jälkeen.

Koodin puutetiheysmittarin käyttö vaatii seuraavat toimenpiteet [33]:

1) Määritä puutteiden vakavuusasteet.

2) Muodosta luokittelujärjestelmä puutteen vakavuutta ja luokkaa varten.
3) Kirjaa jokaisessa tarkastuksessa tuotteen koko ja puutteiden kokonaismäärä.
4) Laske koodin puutetiheydelle suhde seuraavalla kaavalla (8):

$$DD = \frac{\sum_{i=1}^{I} D_i}{KSLOC} \qquad (8)$$

jossa

$D_i$ = tarkastuskierroksella i havaittujen ohjelmakoodin puutteiden lukumäärä

$I$ = tarkastusten lukumäärä

$KSLOC$ = suorittuvan ohjelmakoodin ja ei-suorittuvien määrittelyjen kokonaisrivimäärä tuhansina riveinä.

### *3.3.4 Testausvaihe*

#### 3.3.4.1 Häiriösuhde

*Häiriösuhde* (failure rate) –mittaria käytetään osoittamaan ohjelmiston luotettavuuden kasvamista testausajan funktiona [33].

Häiriösuhde $\lambda(t)$ voidaan arvioida käyttämällä ohjelmiston luotettavuuden kasvumallina (software reliability growth model) esimerkiksi epähomogeenista Poisson-prosessimallia (NHPP, nonhomogeneous Poisson process model).

NHPP-mallin perusoletukset ovat Farrin [10] mukaan:

1) Kumulatiivinen häiriöitten lukumäärä ajassa $t$, $M(t)$, noudattaa Poisson-prosessia keskiarvofunktiona $\mu(t)$. Keskiarvofunktiossa virheiden odotettu lukumäärä ajasta $t$ aikaan $t + \Delta t$ on verrannollinen ei-havaittuihin virhei-

siin ajassa *t*. Lisäksi sen oletetaan olevan rajattu ja ei-vähenevä ajan funktio s.e. $\lim_{t\to\infty}\mu(t) = N$, eli se on äärellinen häiriömalli.

2) Virheiden lukumäärät ($f_1, f_2, ..., f_n$), jotka on havaittu aikaväleillä [($t_0 = 0$, $t_1$), ($t_1$, $t_2$),...,($t_{i-1}$, $t_i$),...,($t_{n-1}$, $t_n$)] ovat riippumattomia mille tahansa äärelliselle aikajoukolle, $t_1 < t_2 < ... < t_n$.

Edellä olevien oletusten perusteella voidaan Farrin [10] mukaan osoittaa, että keskiarvofunktion tulee olla muotoa

$$\mu(t) = N(1 - e^{-bt}) \qquad (9)$$

jollekin vakioille $b > 0$ ja $N > 0$. $N$ on odotettavissa olevien ja myöhemmin havaittavien virheiden kokonaismäärä. Koska häiriöiden tiheysfunktio (failure intensity function) on derivaatta $\mu(t)$:stä, niin

$$\lambda(t) = Nbe^{-bt} \qquad (10)$$

joka Misran [25] mukaan on kaava häiriösuhteen laskemiseen. Farrin [10] mukaan on huomattava, että häiriöiden tiheysfunktio on voimakkaasti laskeva kun $t > 0$.

Farrin [10] mukaan vakioille $N$ ja $b$ saadaan lasketuksi estimaatit seuraavan yhtälöparin (11) ratkaisuna, kun häiriöiden lukumäärät ja aikavälit tunnetaan:

$$\hat{N} = \frac{\sum_{i=1}^{n} f_i}{\left(1 - e^{-\hat{b}t_n}\right)} \qquad \text{ja} \qquad \frac{t_n e^{-\hat{b}t_n} \sum_{i=1}^{n} f_i}{\left(1 - e^{-\hat{b}t_n}\right)} = \sum_{i=1}^{n} \frac{f_i\left(t_i e^{-\hat{b}t_i} - t_{i-1}e^{-\hat{b}t_{i-1}}\right)}{e^{-\hat{b}t_{i-1}} - e^{-\hat{b}t_i}} \qquad (11)$$

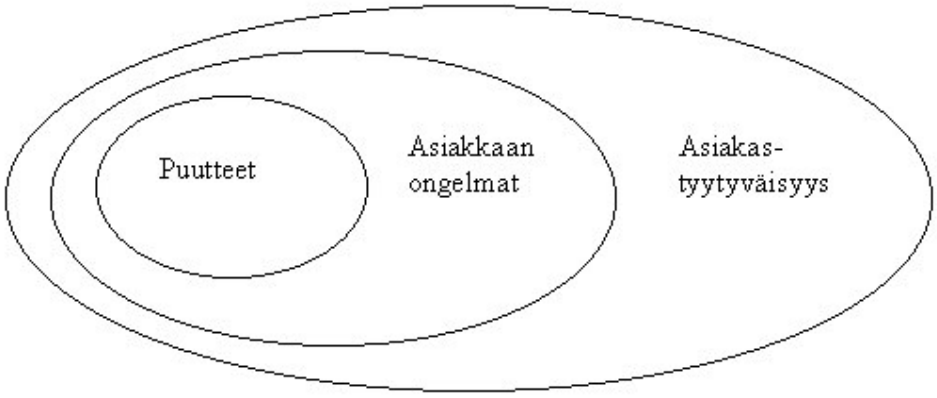
Jälkimmäinen yhtälö kaavasta (11) estimaatin *b* löytämiseksi ratkaistaan numeerisin menetelmin ja ratkaisu sijoitetaan sitten edelliseen yhtälöön estimaatin *N* löytämiseksi.

## 3.4 Käytön ja ylläpidon laadun mittaaminen

Käyttö- ja ylläpitovaiheessa ilmenevät ongelmat ja puutteet johtuvat kehitysvaiheen aikana tapahtuneista virheistä. Eli ohjelmiston ylläpitovaiheessa ei Kan'in [21] mukaan ohjelmiston laatuun voi vaikuttaa enää millään tavalla. Ylläpitovaiheen aikana havaitut ja asiakkaiden raportoimat puutteet voidaan vain korjata niin nopeasti ja laadukkaasti kuin mahdollista.

### *3.4.1 Lopputuoteen laatu*

Käytön aikana lopputuotteen laatua voidaan Kan'in [21] mukaan ajatella kahdella tasolla: sisäisenä tuotteen laatuna ja asiakastyytyväisyytenä. Sisäinen tuotteen laatu voidaan mitata puutetiheyden avulla. Nopeasti ja onnistuneesti korjatut puutteet voivat vaikuttaa positiivisesti asiakastyytyväisyyteen. Turvallisuuskriittisissä järjestelmissä, kuten lennonohjaus- ja asejärjestelmissä käytetään usein puutetiheyden sijaan *keskimääräistä vikaantumisaikaa* (Mean Time To Failure, MTTF), jonka laskemiseksi pidetään kirjaa häiriöiden välisistä ajoista. Kuva 8 havainnollistaa lopputuotteen laadun ilmenemistä.




Kuva 8. Lopputuotteen laatu [21].

*Asiakkaan kohtaamat ongelmat* -mittarilla mitataan asiakkaitten kohtaamien ongelmien määrää ohjelmistotuotetta käyttäessään. Ongelmat voivat olla vikojen aiheuttamia häiriöitä tai sitten käytettävyysongelmia, epäselvyyksiä dokumentaatiossa tai jopa käyttäjävirheitä. *Asiakkaan kohtaamat ongelmat* –mittari ilmaistaan usein muodossa: ongelmat per käyttäjäkuukausi (problems per user month, PUM).

$$PUM = \frac{\text{käyttäjien ilmoittamien ongelmien kokonaismäärä aikajaksolla}}{\text{ohjelmiston lisenssikuukausien lukumäärä aikajaksolla}} \quad (12)$$

missä lisenssikuukausien lukumäärä = asennettujen lisenssien lukumäärä * kuukausien lukumäärä ajanjaksolla.

PUM lasketaan yleensä jokaiselle kuukaudelle sen jälkeen kun ohjelmisto on julkaistu. Alhaiseen PUM -arvoon voidaan päästä seuraavilla toimenpiteillä:

- kehittämällä ohjelmiston kehitysprosessia, jolloin tuotteen puutteet vähenevät
- kehittämällä ohjelmiston käytettävyyttä, dokumentaatiota, asiakkaitten koulutusta ja asiakastukea
- lisäämällä tuotteen myyntiä.

Kaikki yllä olevat toimenpiteet ovat hyödyllisiä laadun parantamista ja liiketoiminnan kehittämistä ajatellen. Tästä syystä PUM:in voidaan sanoa olevan hyvä ja hyödyllinen mittari.

*Asiakastyytyväisyys*: Asiakkaitten tyytyväisyys ohjelmistotuotteen kokonaislaatuun ja sen tiettyihin ominaisuuksiin saadaan selville asiakaskyselyin. Kyselyissä voidaan painottaa erilaisia ominaisuuksia. Esimerkiksi Kan'in [21] mukaan IBM käyttää kyselyissään CUPRIDISMO-luokittelua (kyvykkyys, toiminnallisuus, käytettävyys, tehokkuus, luotettavuus, asennettavuus, ylläpidettävyys, dokumentaatio, palvelut ja kokonaislaatu), kun taas Hewlett-Packard käyttää FURPS-luokittelua (toiminnallisuus, käytettävyys, luotettavuus, tehokkuus ja palvelut). Usein kyselyissä on käytössä seuraavanlainen asteikko:

- hyvin tyytyväinen
- tyytyväinen
- ei osaa sanoa, neutraali
- tyytymätön
- täysin tyytymätön.

Yllä olevaan asteikkoon perustuen voidaan muodostaa useita mittareita, riippuen analyysin käyttötarkoituksesta, esimerkiksi:

1) Täysin tyytyväiset asiakkaat (%)
2) Tyytyväiset asiakkaat (tyytyväiset ja täysin tyytyväiset) (%)
3) Tyytymättömät asiakkaat (tyytymättömät ja täysin tyytymättömät) (%)
4) Ei-tyytyväiset asiakkaat (neutraalit, tyytymättömät ja täysin tyytymättömät) (%)

*3.4.2 Ylläpidon laatu*

Tärkeitä ylläpidon laadun mittareita ovat Kan'in [21] mukaan: keskeneräisten korjausten ja keskeneräisten korjausten hallinnan indeksi (fix backlog and backlog management index), korjausten vasteaika ja korjausten vastaavuus (fix response and fix responsiveness), myöhästyneiden korjausten määrä prosentteina (percent delinquent fixes) ja korjausten laatu (fix quality).

*Keskeneräisten korjausten ja niiden hallinnan indeksi*: *Keskeneräisten korjausten indeksi* on yksinkertaisesti niiden raportoitujen ongelmien lukumäärä, jotka ovat korjaamatta kuukauden/viikon lopussa. *Keskeneräisten korjausten hallinnan indeksi* (backlog management index, BMI) on toinen mittari avoimien ja korjaamattomien ongelmien hallintaan. BMI lasketaan seuraavasti:

$$BMI = \frac{\text{Käsiteltyjen ongelmien lukumäärä kuukauden aikana}}{\text{Saapuneitten ongelmien lukumäärä kuukauden aikana}} \text{x}100\% \quad (13)$$

Jos BMI-arvo on suurempi kuin sata, keskeneräiset korjaukset ovat vähentyneet. Jos arvo on taas pienempi kuin sata, keskeneräiset korjaukset ovat lisääntyneet.

*Korjausten vasteaika ja korjausten vastaavuus*: Useissa ohjelmistoja kehittävissä organisaatioissa on asetettu aikarajat, joiden puitteissa korjausten pitää olla saatavilla raportoituihin puutteisiin. Aikarajat ovat usein riippuvaisia puutteen vakavuudesta, eli mitä vakavampi puute on sitä tiukempi on aikaraja sen korjaamiseen. Korjausten lyhyt vasteaika voi johtaa hyvään asiakastyytyväisyyteen, jos korjaukset vastaavat asiakkaan odotuksia. Eli on erittäin tärkeää, että korjaukset korjaavat raportoidut puutteet.

*Myöhästyneiden korjausten määrä prosentteina*: Korjaus on myöhästynyt, jos korjaukseen kuluva aika ylittää selvästi vaaditun vasteajan. Myöhästyneiden kor-

jausten määrä prosentteina (percent delinquent fixes, DF) lasketaan seuraavalla kaavalla:

$$DF = \frac{\text{Korjausten lkm, joiden vasteaika ylitti aikarajan}}{\text{Korjausten lkm, jotka valmistuivat ajallaan}} \text{ x } 100\ \% \qquad (14)$$

*Korjausten laatu*: Korjausten laatu eli puutteellisten korjausten määrä on tärkeä laatumittari ohjelmiston ylläpitovaiheessa. Asiakkaan kannalta on jo tarpeeksi ikävää ylipäätään kohdata toiminnallisia puutteita ohjelmistoissa. Vielä pahempaa on, jos puutteiden korjauksetkin ovat puutteellisia. Korjaus on puutteellinen, jos se ei korjaa raportoitua ongelmaa tai jos se korjaa alkuperäisen ongelman, mutta aiheuttaa uusia puutteita. Puutteellisten korjausten määrä on yksinkertaisesti puutteellisten korjausten määrä kaikista korjauksista prosentteina. Jos puutteiden ja niiden korjausten lukumäärä on suuri, pieni puutteellisten korjausten prosentti voi antaa liian positiivisen kuvan, vaikka puutteellisten korjausten lukumäärä olisikin melko suuri. Tästä syystä on parempi käyttää suoraan puutteellisten korjausten lukumäärää tunnuslukuna. Ylläpitovaiheen laatutavoitteena on tietenkin: ei yhtään puutteellista eikä myöhästynyttä korjausta.

## 4. YHTEENVETO

Ohjelmiston mittaaminen kaikissa ohjelmistoprosessin vaiheissa on hyödyllistä, koska sen avulla ohjelmistoprosessia ja sen lopputuloksena syntyvää ohjelmistoa on mahdollista kehittää laadukkaammaksi. Laadukas ohjelmisto on erittäin tärkeä ja arvokas kilpailutekijä ohjelmistomarkkinoilla.

Ohjelmistoprosessia voidaan kehittää laadukkaammaksi puutteita mittaamalla ja keräämällä kohdatuista puutteista tietoa. Tutkielmassa tarkastellut puutemittarit ovat Li' n ja Smidtsin artikkelissa [23] arvioitu ohjelmistoalan asiantuntijoiden toimesta parhaiten soveltuviksi mittareiksi ohjelmistoprosessin kehitysvaiheisiin. Tarkastellut lopputuotteen ja ylläpidon laadun mittarit perustuvat Kan'in [21] esittämiin mittareihin.

Mittaamisen toteuttaminen esitellyillä mittareilla tulee olla järjestelmällistä, jotta niiden käytöstä saadaan hyötyä. Mittaamista aloitettaessa on muistettava, että tuloksia ohjelmiston laadun paranemisen suhteen ei voi saada heti, vaan tulokset alkavat näkyä vasta muutaman mitatun projektin jälkeen.

Tässä tutkielmassa ei käsitelty esiteltyjen mittareiden käytännön toteutusta, niiden käytöstä saatuja kokemuksia eikä niiden käytön vaatimia resursseja. Jatkotutkimuksen aiheena voisikin olla mittaamisen toteuttamisesta esitellyillä mittareilla saatujen kokemusten tarkastelu sekä mittaamisen toteuttaminen esitellyillä mittareilla pienyrityksessä mahdollisimman pienin kustannuksin ja resurssein.

**VIITTEET**

[1] Abran A., Desharnais J., Oligny S., St-Pierre D., Symons C. (2001) *COSMIC-FFP Measurement Manual, Version 2.1*. Internet WWW-sivu, URL: http://www.lrgl.uqam.ca/cosmic-ffp/ffp21.jsp.

[2] Albrecht, A.J. (1979): Measuring Application Development. *Proceedings of IBM Applications Development Joint SHARE/GUIDE Symposium*. Monterey, CA, 83-92.

[3] Arthur, L.J. (1985) *Measuring Programmer Productivity and Software Quality*. John Wiley & Sons Inc., New York.

[4] Boehm, B., Basili, V.R. (2001) Software Defect Reduction Top 10 List. *Computer*, **34**(1), 135-137.

[5] Boehm, B.W. (1981): *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ.

[6] Comer, P., Bradley, P., Ross, N. (1987): *Software Data Collection Manual*. Issue 2, ALV/PRJ/SE/078: Software Data Library Project.

[7] Conte, S.D., Dunsmore, H.E., Shen, V.Y. (1986) *Software Engineering Metrics and Models*. Benjamin/Cummings, Menlo Park, California.

[8] Daily, K. (1992) *Quality Management for Software*. NCC Blackwell Limited, Oxford.

[9] DeMarco, T. (1982) *Controlling Software Projects*. Yourdon Press, New York.

[10] Farr, W. (1996) Software Reliability Modelling Survey. *Handbook of Software Reliability Engineering* (ed. Lyu M.). McGraw-Hill, New York.

[11] Fenton, N. E., Pfleeger S.L. (1997) *Software Metrics: A Rigorous and Practical Approach*. International Thomson Publishing Europe, London.

[12] Ferrari, D. (1978): *Computer System Performance Evaluation*. Prentice Hall, Englewood Cliffs, NJ.

[13] Ferrari, D., Serazzi, G., Zeigner, A. (1983): *Measurement and Tuning of Computer Systems*. Prentice Hall, New York.

[14] Fetcke, T. (2001): *The Warehouse Software Portfolio, A Case Study in Functional Size Measurement*. WWW-sivu, http://www.lrgl.uqam.ca/cosmic-ffp/casestudies/Fetcke1999b.pdf (5.2.2005).

[15] Florac, W.A., Carleton, A.D. (1999) *Measuring the Software Process –Statistical Process Control for Software Process Improvement.* Addison Wesley Inc., Massachusetts.

[16] Garmus D., Herron D. (2001): *Function Point Analysis, Measurement Practices for Successful Software Projects*. Addison-Wesley, Boston.

[17] Halstead, M. (1977): *Elements of Software Science*. Elsevier, North Holland.

[18] Humphrey, W.S. (1989): *Managing the Software Process*. Addison-Wesley, Reading, MA.

[19] IEEE (1990) *IEEE standard glossary of software engineering terminology*. IEEE Std 610.12. IEEE Computer Society Press.

[20] IEEE (1988) *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*. IEEE Std 982.2. New York:IEEE.

[21] Kan, S.H. (2003) *Metrics and Models in Software Quality Engineering*. Addison-Wesley, Boston.

[22] Kleinrock, L. (1975): *Queueing Systems, Volume 1: Theory, and Volume 2: Computer Applications*. Wiley, New York.

[23] Li, M., Smidts, C.S. (2003) A Ranking of Software Engineering Measures Based on Expert Opinion. *IEEE Transactions on Software Engineering*, **29**(9), 811-824.

[24] McCabe, T.J. (1976) A Complexity Measure. *IEEE Transactions of Software Engineering*, **2**(4), 308-320.

[25] Misra, P.N. (1983) Software Reliability Analysis. *IBM Systems Journal*, **22**(3), 262-270.

[26] Myers, G.J. (1979): *The Art of Software Testing*. John Wiley & Sons, New York.

[27] Möller, K.H., Paulish, D.J. (1993) *Software Metrics: A Practitioner's Guide to Improved Product Development*. Chapman & Hall, London.

[28] OMG (2003): *Unified Modeling Language 1.5*. WWW-sivusto, http://www.omg.org/technology/documents/formal/uml.htm (5.2.2005).

[29] Pall, G.A. (1987) *Quality Process Management*. Prentice-Hall, New Jersey.

[30] Pollice, G., Augustine, L., Lowe, C., Madhur, J. (2004) *Software Development for Small Teams: A RUP-Centric Approach.* Addison-Wesley, Boston.

[31] Pressman, R.S. (2000) *Software Engineering: A Practitioner's Approach*. McGraw-Hill Publishing Company, Glasgow.

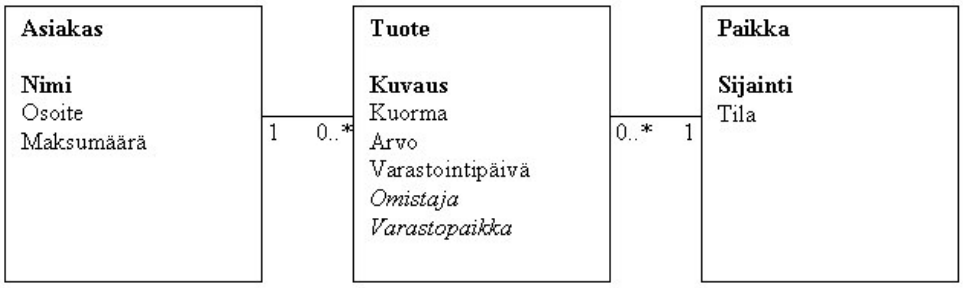
[32] Putnam, L.H. (1978): A General Empirical Solution to the Macrosoftware Sizing and Estimating Problem. *IEEE Transactions of Software Engineering*, **4**(4), 345-361.

[33] Smidts, C., Li, M. (2000*) Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems*. Technical Report, NUREG/GR-0019, University of Maryland, Washington D.C.

[34] Solingen R. van, Berghout E. (1999): *The Goal/Question/Metric Method: a Practical Guide for Quality Improvement of Software Development*. McGraw-Hill Publishing Company, London.

# LIITE 1 ESIMERKKISOVELLUS TOIMINNALLISEN KOON LASKEMISEKSI

Esimerkkinä on osa yksinkertaisesta varastotietojen hallintasovelluksesta Fetcken [14] mukaan. Sovellusta käyttävät yritykset, joilla on varastotilaa vuokrattavana muille yrityksille. Sovellus käyttää tietokantatauluja Asiakas, Tuote ja Paikka:

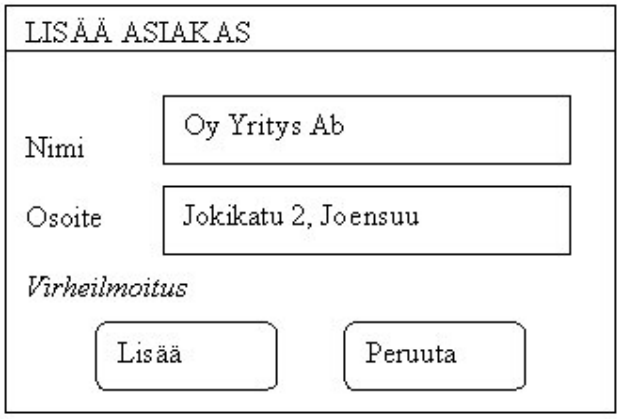



Tietokantataulussa Asiakas on varastotilaa haluavan asiakkaan tiedot: Nimi (perusavain), Osoite sekä Maksu. Maksu-kentässä on tieto siitä, kuinka paljon asiakkaalla on maksettavanaan varastotilasta. Tuote-taulun perusavain on Kuvaus-kenttä. Kuorma on varastointikoon yksikkö. Tuotetta voi olla varastossa esimerkiksi kaksi kuormallista. Arvo-kenttä tarvitaan esimerkiksi vakuutusmaksuja varten. Sen lisäksi Tuote-taulussa on varastointipäivälle oma kenttänsä sekä viiteavaimet Omistaja ja Varastopaikka. Paikka-taulu 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.

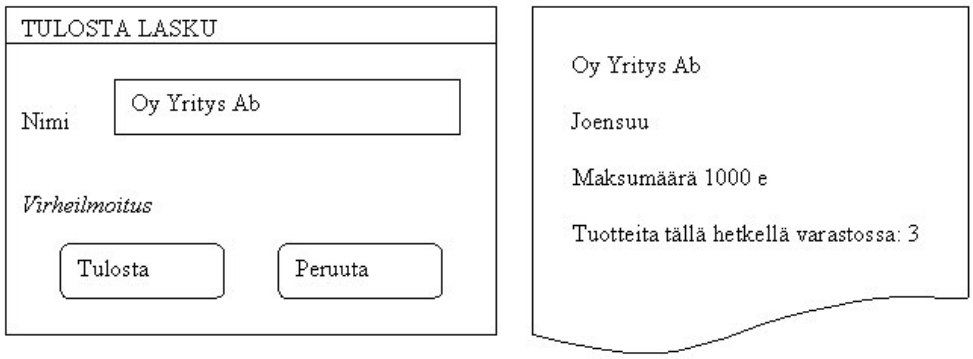
Tarkastellaan seuraavia sovellukseen liittyviä tapahtumia:

1. Lisää asiakas
2. Tulosta lasku
3. Kysely varastoiduista tuotteista.

**Lisää asiakas:** Lisää asiakas -tapahtuma lisää rivin Asiakas-tauluun tietokantaan. Käyttäjän on syötettävä tiedot nimi ja osoite asiakasta lisättäessä. Maksumäärä alustetaan ohjelmallisesti nollaksi. Kun käyttäjä painaa Lisää-painiketta, lisätään tiedot tietokantaa. Jos käyttäjän antamalla nimellä on jo olemassa asiakas tietokannassa, tietoja ei lisätä ja annetaan virheilmoitus. Käyttäjä voi peruuttaa toiminnon Peruuta-painikkeella. Tapahtumaan liittyvä käyttöliittymälomake voisi olla muotoa:




**Tulosta lasku:** Asiakkaalle lähetettävä lasku sisältää nimen ja maksettavan laskun määrän. Lasku sisältää myös asiakkaan sillä hetkellä varastoitujen tuotteiden lukumäärän. Jos annetulla asiakkaan nimellä ei löydy asiakasta, tulee virheilmoitus. Tapahtumaan liittyvä käyttöliittymälomake ja lasku voisivat olla muotoa:




**Kysely varastoiduista tuotteista:** Tapahtuman avulla voidaan tulostaa tietyssä varastopaikassa olevien tuotteiden tiedot. Käyttäjä syöttää varastopaikan sijainnin. Jos kyseinen varastopaikka on olemassa, tulostetaan varastopaikan Sijainti- ja Tila-tiedot. Muutoin annetaan virheilmoitus. Käyttäjä voi peruuttaa toiminnon Peruuta-painikkeella. Kyselyn syötettä ja vastausta voidaan havainnollistaa: