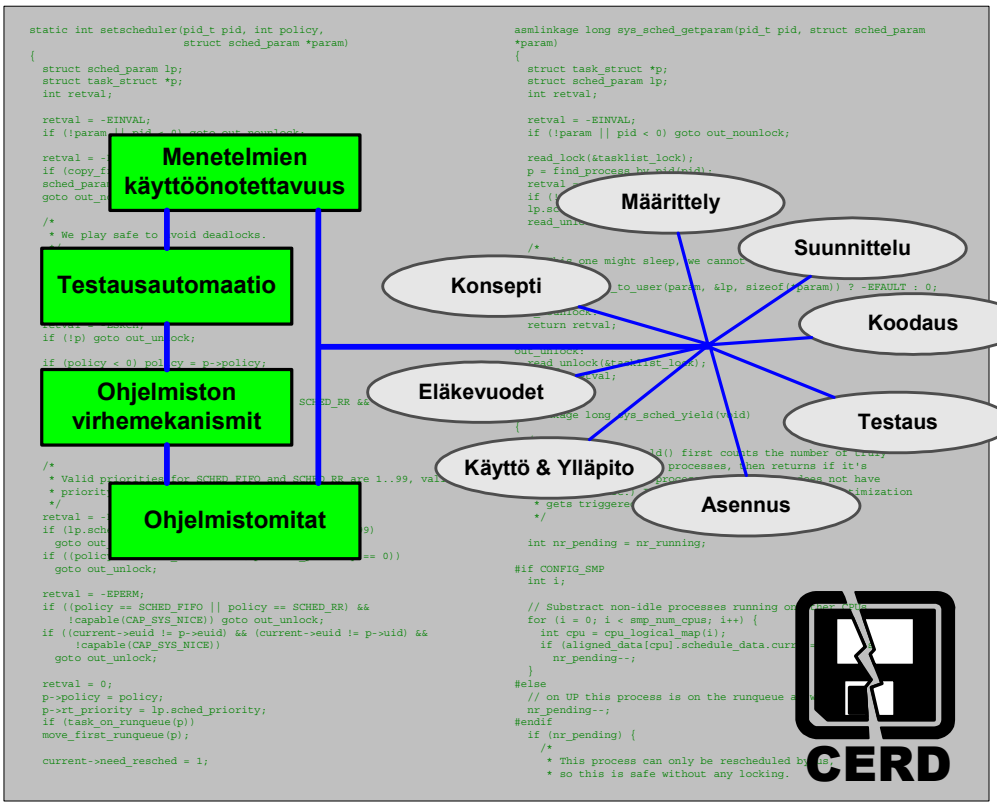


Hannu Harju & Mika Koskela

Kustannustehokas ohjelmiston luotettavuuden suunnittelu ja arviointi

Osa 2

VTT TIEDOTTEITA – RESEARCH NOTES 2193

Kustannustehokas ohjelmiston luotettavuuden suunnittelu ja arviointi

Osa 2

Hannu Harju & Mika Koskela

VTT Tuotteet ja tuotanto

ISBN 951–38–6135–X (nid.)
ISSN 1235–0605 (nid.)

ISBN 951–38–6136–8 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1235–0605 (URL: http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Tuotteet ja tuotanto, Tekniikantie 12, PL 1301, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 6752

VTT Industriella System, Teknikvägen 12, PB 1301, 02044 VTT
tel. växel (09) 4561, fax (09) 456 6752

VTT Industrial Systems, Tekniikantie 12, P.O.Box 1301, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 6752

Otamedia Oy, Espoo 2003

Harju, Hannu & Koskela, Mika. Kustannustehokas ohjelmiston luotettavuuden suunnittelu ja arviointi. Osa 2 [Cost-effective reliability design and assessment of software. Part 2]. Espoo 2003. VTT Tiedotteita – Research Notes 2193. 107 s.

Avainsanat software dependability assessment, software metrics, software reliability engineering, automated software testing, software measurement data

Tiivistelmä

Ohjelmistojen käyttäminen kriittisiin sovelluksiin on jatkuvassa kasvussa. Päinvastoin kuin laitteistoviat, ohjelmistoviat ovat systemaattisia ja ne voivat piileksiä pitkiä aikoja ennen paljastumistaan. Tämä tiedote on toinen osa tutkimussarjassa, jossa käsitellään ohjelmiston luotettavuuden kustannustehokasta suunnittelua ja arviointia. Osan kaksi teemoina ovat uusien menetelmien vähäisen käytön syyt, automaattinen testaaminen luotettavuuden ilmaisijana, ohjelmiston virhemekanismit sekä ohjelmistomittojen käyttö ohjelmiston luotettavuuden arvioinnin apuna.

Kaikki kehittyneet ohjelmistoprosessit ja -menetelmät lupaavat vähentää kustannuksia, vaivannäköä tai virheitä sekä parantaa laatua ja kasvattaa luotettavuutta. Huolimatta näistä toivottavista ominaisuuksista yritykset eivät ole omaksuneet nykyaikaisia menetelmiä käyttöönsä. Formaalit menetelmät, mittausprosessit, standardit ja ohjeet sekä jopa automaattiset testausmenetelmät eivät ole useimpien ohjelmistokehittäjien suosiossa.

Testaaminen on taitolaji. Jos oletetaan, että pienellä joukolla testitapauksia on löydettävä useimmat ohjelmistovirheet, testitapausten valitseminen on tärkeässä asemassa. Automaattiseen testaamiseen asetetaan suuria odotuksia. Sen odotetaan kasvattavan testikattavuutta ja siten parantavan luotettavuuden osoittamista, mutta automaatiossa taidontarve on toinen verrattuna perinteiseen testaamiseen.

Kattavuus on monitahoinen käsite. Puhutaan esimerkiksi testikattavuudesta ja koodikattavuudesta, jotka kummatkin sisältävät useita ominaisuuksia. Eroa kattavuuden ja kattavuusolettamuksien välillä ei kuitenkaan tehdä, koska virhemekanismin teoreettinen tuntemus ei ole hyvin kehittynyt. Virhemekanismi on kuitenkin kaiken testaamisen ja suunnittelun perustekijöitä etsittäessä virheitä ja suojauduttaessa niiltä.

Ohjelmistomittoja on perinteisesti käytetty ohjelmistoprosessin ja -projektin hallinnallisiin toimintoihin. Mittojen käyttöä ohjelmiston luotettavuuden arvioinnin apuna on tutkittu runsaasti, mutta käytännön ohjelmistotyöhön sopivia menetelmiä on verrattain vähän, mikä johtuu mitattavissa olevien ohjelmiston ominaisuuksien epämääräisestä suhteesta luotettavuuteen. Mittaustieto on kuitenkin merkittävä informaation lähde varsinkin ohjelmiston varhaisissa elinkaaren vaiheissa tapahtuvien ohjelmiston riskiosien tunnistamisessa ja korjaavien toimenpiteiden kohdentamisessa.

Harju, Hannu & Koskela, Mika. Kustannustehokas ohjelmiston luotettavuuden suunnittelu ja arviointi. Osa 2 [Cost-effective reliability design and assessment of software. Part 2]. Espoo 2003. VTT Tiedotteita – Research Notes 2193. 107 p.

Keywords software dependability assessment, software metrics, software reliability engineering, automated software testing, software measurement data

Abstract

Software is increasingly being used in critical applications. Unlike most hardware failures, software failures are systematic and software faults may lie hidden for a long time before being revealed. This publication is a second part of research project which study cost effective design and assessment of software dependability. Three specific themes are introduced: why software methods are not used in practice, test automation in demonstrating software dependability, failure mechanisms and software metrics utilised in software dependability assessment.

All advanced software prosesses and methods offer to reduce cost or effort, or defects, and to increase quality and reliability. In spite of these desireable features, most modern software methods are not adopted by organisations. Formal methods, measurement prosesses, standards and guidelines, and even automatic testers have not won favour with most of the software developers.

Testing is a skill. If a small number of test cases is expected to find most of the faults in the software, the task of selecting test cases is an important one. Automating tests is expected to increase testing coverage which means better demonstration for dependability, but the skill that automation needs differs from the skill of traditional testing.

Coverage is a multidimensional concept. We speak about testing coverage or code coverage, which both include tens of features, but we seldom make difference with coverage and coverage assumption. It is because the software failure mechanism, that is, the sequence fault – error – failure is not theoretically very well known, even if the very same failure mechanism is the basis for finding bugs by testing and shielded from their concequenses.

Software metrics has been traditionally used in issues of software process development and project management. Research of utilising software metrics in software dependability assessment has been done but few practical methods exist because of difficulties to describe dependability in terms of measurable properties. Software measurement data however offers valuable information in the early phases of software lifecycle in particular, where error-prone component identification and corrective operations are taken place.

Alkusanat

Tutkimushanke toteutettiin ABB Industryn, Fortumin, Teollisuuden Voima Oy:n ja VTT Tuotteet ja tuotannon yhteistyönä. Näiden yritysten lisäksi rahoituksesta vastasi Teknologian kehittämiskeskus (Tekes). Varsinaisen tutkimustyön hoiti pääasiallisesti VTT Tuotteet ja tuotanto. Tiedote on tutkimusprojektin "Kustannustehokas ohjelmistojen luotettavuuden suunnittelu ja arviointi, CERD" toinen osa.

Kiitämme kaikkia osallistuneita tahoja ja henkilöitä arvokkaasta panoksesta. Erityiskiitokset johtoryhmän edustajille Kari Rintalalle, Tekes, Jari Pesoselle, Teollisuuden Voima Oy, Martti Välisuolle, Fortum, Jari Yli-Juutille, ABB Industry, Marja-Leena Järviselle ja Heimo Takalalle, Säteilyturvakeskus, Irmeli Lambergille, Innopoli Oy sekä Olli Ventälle, VTT.

Hannu Harju ja Mika Koskela

Sisällysluettelo

Symboliluettelo

BBN	Bayes Belief Nets
CASE	Computer Aided Software Engineering
CERD	Cost Effective Reliability Design
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
COTS	Commercial Off The Shelf
GUI	Graphical User Interface
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standard Organisation
SPICE	Software Process Improvement and Capability dEtermination
STUT	Statistical Usage Testing
VDM	Vienna Development Method
V&V	Verifiointi ja validointi
Y2K	Year 2000

1. Johdanto

Tässä tekstissä termiä ‘luotettavuus’[1] (engl. dependability) käsitellään Laprien (1998) määrittelemässä muodossa. Sen mukaan luotettavuus on järjestelmän ne ominaisuudet, joiden avulla voimme luottaa järjestelmän toimivan tarvittaessa. Määrittely viittaa siten *käyttäjälähtöiseen* näkemykseen ja kattaa useita attribuutteja: toimintavarmuus, käytettävyys, turvallisuus, ylläpidettävyys ja tietoturva.

Suomen kielessä luotettavuudella useimmiten tarkoitetaan englannin kielen sanaa reliability, toimintavarmuus. Ohjelmiston toimintavarmuus määritellään usealla tavalla. Musa (1998) käyttää mielellään normaalia laitteistolle sopivaa määrittelyä: tarkasteltavan ohjelmiston kyky toimia vaaditulla tavalla vaadituissa olosuhteissa. Hän perustelee samanlaisuutta sillä, että ohjelmisto suoritetaan aina laitteistossa ja yhteinen määritelmä helpottaa järjestelmätason luotettavuuslaskentaa.

Musan määrittely on yleisin, mutta muitakin määritelmiä löytyy. ISO 9126 (1991) kohdistaa ohjelmiston toimintavarmuuden nimenomaan ohjelmistolle. Toimintavarmuus koostuu sen mukaan kolmesta tekijästä: kypsyys, virhesietoisuus ja palautuvuus.

Ohjelmiston luotettavuus on tärkeää lähes kaikilla toimialoilla. On vaikea löytää sellaista elinkeinoalaa, jossa tuotteen valmistaja tai palvelun tarjoaja ei olisi ainakin jossakin määrin huolestunut ohjelmiston luotettavuudesta. Tästä parhaana esimerkkinä oli milleniumaikaisten Y2K-vikojen etsintä, johon uhrattiin runsaasti huomiota ja resursseja. Ongelmanasettelut ja uhkakuvat saivat julkista tilaa kaikkialla mediassa. Merkittävää on sekin, että Y2K-selvityksiin ryhtyivät monet sellaiset liiketoimintatahot, yritykset ja järjestelmät, joita ongelman ei uskottu lainkaan koskevan.

Vaateita ohjelmistojen luotettavuuden parantamiseksi tulee yhä enemmän ja suuntaus jatkuu, mikä asettaa vaatimuksia sekä ohjelmiston luotettavuuden suunnittelulle että arvioinnille. Näköpiirissä ovat ainakin seuraavat syyt, jotka kuormittavat luotettavuuden suunnittelua ja arviointia:

- Ohjelmistoille annetaan vastuuta yhä kriittisemmissä toiminnoissa.
- Ohjelmistopohjaiset järjestelmät korvaavat vanhemmalla teknologialla toteutettuja järjestelmiä.
- Ohjelmisto on mukana toimintojen toteutuksessa ainoana vaihtoehtona.

[1] Joskus käytössä on myös muoto ’kokonaisluotettavuus’.

- Ohjelmiston mukaanotto osaksi järjestelmää kasvattaa integroitumista ja vuorovaikutusta, usein riippumattomasti ihmisen väliintulosta.

- Ohjelmiston tarjoamat palvelut tulevat jatkuvasti yhä laajemmaksi osaksi ihmisten jokapäiväistä elämää.

Kuten edellä todettiin, yleisin toimintavarmuuden määrittely perustuu laitteiston vastaavaan määrittelyyn, missä on selkeä ristiriita. Kaikki ohjelmistovirheet ovat suunnitteluvirheitä, koska ohjelmisto itsessään on pelkkää suunnittelua. Laitteistoviat koostuvat enimmäkseen satunnaisvioista, systemaattisiin virheisiin sovelletaan suhteellisen onnistuneesti laatujärjetelmiä. Laatujärjestelmiä suositaan erityisesti ohjelmistoprosesseissa, mutta prosesseista puuttuu tukeva luotettavuustekninen perusta samassa mielessä kuin se on olemassa laitteistoille. Mitä olisikaan sillan suunnittelu ja rakentaminen pelkästään laatujärjestelmään tukeutuen, ilman lujuus- ja luotettavuuslaskelmia.

Erityisesti ohjelmistotekniikan ja luotettavuustekniikan tutkimuksessa ja kehitystyössä tuotetaan paljon menetelmiä ja työkaluja, jotka kehittäjiensä hämmästykseksi eivät yllä käyttäjien suosioon. Automaattisen testaamisen työvälineet ovat yhtenä hyvänä esimerkkinä. Sama ongelma koskee kaikkia niitä ohjelmistokehityksen tukituotteita, joilla ei suoranaisesti vaikuteta ohjelmiston saattamiseen toimintakuntoiseksi. On jopa esitetty aina silloin tällöin mielipiteitä, että tulisi keskittyä olemassa olevien menetelmien ja työvälineiden jatkokehitykseen uusien sijasta.

Tulevaisuudessa ohjelmiston laatu on merkittävässä asemassa ohjelmistoteollisuudessa, joidenkin mielipiteiden mukaan jopa vallitsevana. Jos tämä on tulevaisuudenkuva, ohjelmiston laadunvarmistusta tukevien prosessien merkitys edelleen korostuu. Yksi tällainen prosessi on testaaminen, jolla usein tarkoitetaan pelkästään dynaamista analyysia, prosessia, jossa ohjelmisto suoritetaan syötetiedoilla eli testitapauksilla ja havainnoidaan suorituksen tulokset.

Tutkimusten mukaan testiosuus saattaa olla jo puolet ohjelmiston kokonaiskehityskustannuksista riippuen tietysti monista tekijöistä kuten tuotteesta ja kehitysprosessista. Kriittisillä ohjelmistoilla ja perättäiskehitettävillä ohjelmistoilla osuus voi muodostua tätäkin suuremmaksi. Testauksen osuus tulee vielä kasvamaan ohjelmistojen kehittämisen ja tuotannon nopeutuessa, ellei kyetä kehittämään tehokkaita suoritustapoja, sillä testauskäytännöt eivät ole pysyneet ohjelmiston kehityksessä mukana. Suuntauksena on ollut testata mahdollisimman paljon, ei suhteuttaen juuri oikean laatu- ja luotettavuustavoitteen saavuttamiseksi.

Automatisoimalla testaamista ylletään korkeaan kattavuuteen. Testaamisen automatisoimisessa tavoitteena on ikävän työn vähentäminen ja testaamisen nopeuttaminen siten,

että kattavampaan tulokseen päästään vähimmin kustannuksin. Kattavuuden parantuessa syntyy mielikuva luotettavuuden parantumisesta helpolla ja osoitettavalla tavalla.

Ohjelmiston luotettavuuden keskeisenä aiheena ovat ohjelmistovirheet. Virheitä ovat bugit koodissa tai puutteet vaatimusmäärittelyssä ja sen toteutuksessa. Virhe etenee ohjelmistokomponentin sisällä ja siirtyy tiedon siirtyessä toiseen komponenttiin ja mahdollisesti ulospäin näkyväksi virhetoiminnaksi ja ohjelmiston sisältävän järjestelmän toimintahäiriöksi. Erilaisia virhemuotoja ja -luokitteluja on lukuisia.

Kuten edellä mainittiin, ISO/ICE-standardin mukaan luotettavuus viittaa kypsyyteen, virhesietoisuuteen ja palautuvuuteen. Kypsyyteen vaikuttavia tärkeimpiä tekijöitä ovat mm. monimutkaisuus, kehittäjän ja organisaation taitavuus, testikattavuus jne. Kypsä ohjelmistokehitys välttää virheiden tekemistä. Virhesietoisuus on periaatteeltaan ohjelmistoilla samaa kuin laitteistoillakin: sillä estetään virhetilanteen kehittyminen järjestelmän virhetoiminnoksi. Palautuvuus astuu kuvaan, kun virhetoiminto on tapahtunut ja halutaan estää sen eteneminen vaaralliseksi tilanteeksi tai onnettomuudeksi.

Ohjelmiston virhemekanismien tutkiminen irrallaan virheitä ja virhetilanteita analysoivista menetelmistä ei ole ollut merkittävässä asemassa tutkijoiden keskuudessa. Virheluokitteluja on paljon, mutta luokittelumahdollisuuksia vieläkin enemmän. Geneeriseen virhetopologiaan tuskin kannattaa edes pyrkiä, mutta sovellusalakohtainen topologia saattaa olla saavutettavissa.

Mittaaminen on keskeinen toimenpide ajateltaessa minkä tahansa prosessin ohjausta ja hallintaa. Ohjelmistotuotantoon liittyviä ohjelmistoa ja ohjelmistoprosessia kuvaavia ominaisuuksia on pyritty mittaamaan yhtä kauan kuin on ollut varsinaista järjestelmällistä ohjelmistotuotantoakin. Mittaamisella hankittua tietoutta on pyritty käyttämään hyväksi arvioitaessa ohjelmiston ulkoisia ominaisuuksia, kuten ohjelmiston laatua tai ohjelmistoprojektin kustannuksia.

Luotettavuus on laadun keskeinen osa-alue. Ohjelmiston laatua ja luotettavuutta tulisi pystyä arvioimaan myös ennen suorituskelpoisen ohjelmakoodin olemassaoloa, elinkaaren varhaisissa vaiheissa. Ohjelmistomittojen laajamittaisempaa käyttöä on pidetty eräänä merkittävänä mahdollisuutena saada tietoa ohjelmiston laadullisesta tasosta jo elinkaaren varhaisten vaiheiden aikana. Ohjelmiston luotettavuutta selittäviä ominaisuuksia ja niitä kuvaavia mittoja on pyritty kartoittamaan, mutta yksiselitteisiä luotettavuutta indikoivia mittoja ei ole pystytty tunnistamaan. Nykykäsityksen mukaisesti käytettäessä ohjelmistomittoja luotettavuuden arvioinnin apuna tulee käyttää useita erityyppisiä mittoja, joiden tarjoama informaatio yhdistetään erillisen luotettavuusmallin avulla luotettavuutta indikoiviksi suureiksi (Smidts & Li 2000, Fenton & Neil 1999).

Ohjelmistomittoja käytetään ensisijaisesti ohjelmistoprosessin ja -projektin hallinnallisiin toimintoihin. Mittaustulosten yhdistäminen ohjelmistotuotannon resurssien ja kustannusten kartoittamiseen on suhteellisen suoraviivainen ja runsaasti tutkittu alue. Ohjelmistomittojen ja laadun/luotettavuuden suhde on kuitenkin mutkikkaampi, eikä laatua pystytä mittaamaan suoraan. Mittaamista voidaan käyttää toisaalta ohjelmistoprosessin kehittämisen ja kontrolloinnin kautta prosessin lopputuotteiden tasalaatuisuuden varmistamiseen, toisaalta yksittäisen ohjelmistotuotteen kelpoistustoimenpiteiden vaatiman informaation tuottamiseen pitkin ohjelmiston elinkaarta.

Ohjelmistomittojen tarjoaman informaation käyttötapa on riippuvainen siitä, kuinka arvokkaaksi tarkastelija kokee tarkasteltavan ohjelmiston luotettavuuden. Jos luotettavuus on ohjelmistolle kriittinen ominaisuus, luotettavuuden kattava mallintaminen on välttämätöntä. Ohjelmistomitat toimivat tällaisessa tilanteessa osana analyysimateriaalin muodostavaa kokonaisuutta. Jos ohjelmiston luotettavuudesta haluttava varmuuden aste on matalampi, ei kattavaa mallinnusta tarvita, vaan taloudellisempi näkökulma on käyttää ohjelmistomittoja ohjelmiston ongelmakohtien tunnistamiseen ja korjaavien toimenpiteiden kohdentamiseen.

2. Mikä neuvoksi, kun menetelmiä ei käytetä?

Ohjelmistoalalla ideoidaan ja kehitetään menetelmiä ja työvälineitä, jotka kehitysosapuolten mielestä ovat edistyksellisiä, soveltuvia ja kustannustehokkaita. Kuitenkaan hyvältä kuulostavat menetelmät eivät mene kaupaksi, mitä menetelmäkehittäjät ovat hämmästelleet jo vuosikausia sellaistenkin menetelmien kuin CASE[2]-työkalujen (Chau 1996, Iivari 1996), olio-ohjelmointitekniikoiden (Fayad et al. 1996) ja erityisesti formaalien menetelmien (Holloway & Butler 1996) kohdalla. Ohjelmistotuotannon uusia menetelmiä ovat eXtreme Programming, feature-driven development, adaptive software development, personal software process, team software process, Rational Unified Process, rapid developement jne. Ne tarjoaisivat vaihtoehtoja työtapojen tai koko yrityskulttuurin muuttamiseksi, mutta ovat jääneet ohjelmistoalan julkaisujen ja artikkeleiden muotisanoiksi.

Ohjelmiston laadun vartioimiseksi on kehitetty laatujärjestelmiä, laadun kypsyysjärjestelmiä ja laadun kyvykkyyden järjestelmiä. Niitä onkin kohtuullisesti ohjelmistovalmistajien käytössä. Luotettavuustekniset arviointimenetelmät ovat jääneet vain kaikkein kriittisimmillä aloilla työskentelevien käyttöön, vaikka luotettavuus tunnustetaan yleisesti tärkeimmäksi laatuattribuutiksi. Ongelman ratkaisu on yleinen ohjelmiston kehitysalalla, ei siis yksin ohjelmiston luotettavuuden suunnittelun tai arvioinnin.

Syitä voi olla useita: hinta, kouluttautumistarve ja käyttöönoton kynnys voivat tuntua liian suurelta. Tuotetaan menetelmiä, vaikka niiden hyödyllisyys on projektin kuluessa menettänyt arvonsa tai ne ovat käyneet saavuttamattomiksi. Projekti jatkuu, koska se on perustettu. Chau (1996) väittää, että suurin syy CASE-työkalujen käyttämättömyyteen on organisaation tavassa innovoida. Organisaatiotasolla innovaationäkymiä tarkastellaan erillään ohjelmiston kehittäjistä, jotka kuitenkin toteuttavat päätökset. Innovointi ja mielipiteiden ottaminen huomioon on erityisesti ohjelmistokehittäjien toiveissa.

Ohjelmiston luotettavuustekniikan menetelmien vähäiseen käyttöönottoon on kaksi merkittävää syytä:

1. Menetelmä on hyvä, mutta sitä ei oteta käyttöön, koska ohjelmistoprojekteissa on täysi työ saada ohjelmat toimimaan. Ohjelmistokehittäjät ovat syvällisesti kiinni itse koodaamisessa. Tukimenetelmät tunnustetaan käyttökelpoisiksi, mutta aika kuluu ohjelman tekemiseen toimivaksi eikä täsmentäviin toimenpiteisiin tartuta. Yhtenä esimerkkinä on automaattinen testaaminen. Myös luotettavuuskysymykset koetaan ylimääräiseksi hienosäädöksi.

[2]Computer Aided Software Engineering

2. Kehityksen alainen menetelmä ei sovellu jatkuvaan käyttöön. Sillä parannetaan kehitysprosessia vain kertaluontoisesti, vaikka alkuperäinen tavoite olisi ollut hyödyntää menetelmää kaikissa projekteissa.

Rifkinin (2001) mielestä ohjelmistotuotannon menetelmien käyttöönottamattomuudessa ei ole niinkään kyse muutosvastarinnasta, vaan yksinkertaisesti siitä, että yritykset katsovat etteivät uudet menetelmät sovi yrityksen strategiaan. Uusia menetelmiä käyttöönottoprosesseineen ei koeta riittävän kustannustehokkaiksi. Vaikka tarvetta olisi, riskit arvioidaan liian suuriksi.

Tiedetään käytännön kokemuksista, että uusien menetelmien omaksuminen ja hyödyntäminen jokapäiväisesti epäonnistuu todella usein. Silloin olisikin kiinnostavaa tietää mitä tekijöitä liittyy sekä onnistumiseen että epäonnistumiseen. Tekijöiksi luetellaan johtotason osallistumis- ja rahoittamisinnokkuutta, muutosten aloittajaosapuolten kyvykkyyttä ja suostuttelevuustaitoa, innovaation häiritsevyyttä sekä muutosten suunnitelmallisuutta ja hallittavuutta. Potentiaalisin menestystekijä on kuitenkin uusien menetelmien selkeä sovittaminen yrityksen strategiaan: menetelmien käyttöönopastaminen, vastuuhenkilön nimeäminen, täsmämenetelmien käyttäminen ja tyytyväisyyden mittaaminen.

Menetelmien konkreettinen käyttöönopastus on yksi ratkaisu. Käyttäjälle asennetaan ohjelma ja neuvotaan kädestä pitäen ohjelman käyttöä. Asiantuntija voi myös nopeasti päätellä, ettei menetelmä sovellu käsiteltävään aiheeseen ja säästää käyttäjää turhilta kokeiluilta, joiden lopputuloksena saattaa olla menetelmästä luopuminen. Esimerkiksi automaattiset testaustyövälineet soveltuvat yleensä vain tietyille ohjelmistoille ja käyttöympäristöille.

Ohjelmistokehityksen valmiudesta voisi vastata henkilö, jonka tehtävänä on saada asiat sujumaan sekä ideoiden kehittelyssä, jatkuvassa työstämisessä ja valitsemisessa että ristiriidattomassa päätöksenteossa. Projektit olisivat tavoitteellisia, ilmapiiri rakentava keskusteltaessa menetelmien heikoista ja hyvistä puolista. Vastuuhenkilön rooli auttaisi suunnitelmien parantamisessa, prosessien määrittämisessä ja muuttamisessa sekä organisatoristen ja prosessitekijöiden parantamisessa.

Tutkimuksilla on osoitettu ohjelmiston tarkistukset tehokkaiksi osiksi ohjelmistotuotantoa. Kuitenkin ainakin joidenkin tarkistusta suorittavien tahoilta on valitettu tarkistusten vaikeudesta, kalleudesta ja tehottomuudesta sekä ennen kaikkea siitä, että ne vievät liian paljon muutenkin vähäistä projektiaikaa. Mikä on väärin? Varmasti väärin on olla varaamatta tarpeeksi resursseja ja tilaa projektiaikataulussa, sekä lisäksi myös se, että yrityksen tarpeisiin ei ole osattu valita oikeanlaista menetelmää. Tarkistusmenetelmistä löytyy runsaasti kirjallisuutta ja aiheesta järjestetään koulutusta. Myös uusia kus-

tannustehokkaita ja jo käytössä kustannustehokkaiksi koettuja menetelmiä on kehitetty. Yksi niistä on lukemistekniikka, jolla tarkastajat löytävät entistä tehokkaammin ohjelmistovirheitä ja puutteellisuuksia artefakteista. Menetelmän systemaattisuus ja täsmällinen ohjeistus tekee lukemistekniikasta yrityskohtaisesti räätälöidyn toimintamallin. Toimintamallissa projektihenkilö, mm. suunnittelija, testaaja tai käyttäjä, voi myös vaihtaa tarkastusnäkökulmaa toisen projektihenkilön näkökulmaksi, mikä edesauttaa täsmällistä tarkistamista esimerkiksi analysoitaessa vaatimusmäärittelyitä.

Green & Hevner (2000) ovat rakentaneet mallin sille, miten onnistutaan levittämään uusia menetelmiä ohjelmistoyrityksiin. Heidän mallinsa kohdentuu informaatioteknologiaan, mutta on yleistettävissä ohjelmistovalmistukseen muillakin aloilla. Mallissa varsinaisina mittareina ovat menetelmän käyttö ja tyytyväisyys. Käyttäjämäärä ei riitä onnistumisen mitaksi, sillä menetelmästä on saatettu luopua toimittavan yrityksen siitä tietämättä. Tyytyväisyys olisi Greenin ja Hevnerin mukaan onnistumisen tärkein mitta. Tyytyväisyyden edellytyksenä on, että ohjelmiston kehittäjä on pystynyt vaikuttamaan teknologian valintaan ja ottanut sen vapaaehtoisesti vastaan. Hän tuntee valintaprosessin, pystyy hyödyntämään vahvuuksiaan uudessa teknologiassa, tietää saavansa riittävää koulutusta, ei pelkää teknologian uutuutta ja hänellä on vahva kuva teknologian tuomista eduista.

Selvimmin vaikeudet menetelmien käytännöllistämisessä ovat formaaleilla menetelmillä. Niillä kyettäisiin parantamaan virheiden löytymistä, automatisoimaan tiettyjen ominaisuuksien tarkistamista ja vähentämään tarvetta modifiointiin. Huomattavista eduista huolimatta formaaleilla menetelmillä on varsin vähäinen käyttäjäkunta ohjelmistoteollisuudessa. Vähäisen käytön syyksi esitetään henkilökunnan suurta koulutustarvetta vaativan matematiikan johdosta, yhteensopimattomuutta muiden ohjelmistopakettien kanssa sekä ohjelmiston kehityselinkaaren laajentumista.

Formaalien menetelmien käyttöönottoa voitaisiin edistää Parnasin (1998) mielestä kahdella tavalla: 1) integroimalla formaalien menetelmien koulutus yliopistojen perusaineisiin ja 2) parantamalla menetelmiä niin kauan kunnes ne soveltuvat käytännön tehtäviin. Perinteisesti matematiikka on kehitetty tiettyyn valmiusasteeseen ja otettu sen jälkeen käyttöön perusopetuksessa, mikä johtaa käyttöön jokapäiväisessä insinöörityössä. Tästä hyvänä esimerkkinä on signaalinkäsittely, jossa ei muutama vuosikymmen sitten ollut nykyisen kaltaisia matemaattisia apuneuvoja käytössä.

Ohjelmistotekniikalle matemaattinen lähestymistapa on vierasta. Tässä tekniikanalassa on selvä kuilu teorian ja käytännön välillä. On ohjelmoijia, jotka tuntevat jossakin määrin teoriaa ja ohjelmoivat, mutta hekään eivät perusta ohjelmointia teoreettiselle pohjalle. Se ei heitä opasta. Parnas on sitä mieltä, että matemaattiset menetelmät mielletään ylimääräisiksi apuneuvoiksi, joihin ei olla valmiita kouluttautumaan todellisen ohjel-

mointitaidon sijasta. Nekään oppilaat, jotka ovat saaneet perustiedot logiikasta ja oppineet siten helposti formaaleita menetelmiä, kuten Z ja VDM, eivät miellä formaaleita menetelmiä hyödyllisiksi ohjelmistonkehityksen todellisessa ympäristössä.

Kumpaa opetusta kuuluisi antaa ensiksi, perusteoriaa vai ohjelmointia? Joidenkin kannanottojen mukaan oppilaita ei saisi päästää ohjelmoimaan ennen kuin he ovat saaneet opetusta perustiedoista. Eräiden muiden kannanotot ovat yhtä selvät: ensin ohjelmointi, jolloin oppilaat kiinnostuvat aiheesta, ja sitten syventävät tiedot. Parnas on artikkelissaan (1998) sitä mieltä, että kumpikaan näistä tavoista ei ole hyvä. On koulutettava yhtä aikaa: jokaisen ohjelmointikurssin tulisi sisältää täsmällistä spesifiointia, jokainen luento esittelisi matemaattisen lähestymistavan integroituna ohjelmistokehitykseen. Kaikkiin kursseihin tulisi sisällyttää matemaattisten taitojen soveltamista.

Myöskään ohjelmiston kehitysstandardit eivät ole laajassa käytössä. Eri standardeita on runsaasti, yli 300 kappaletta. Lukumäärä ei vastaa käyttömäärää, vaikkakaan täsmällisiä tutkimuksia käyttömääristä käytännön ohjelmistokehityksen projektityöskentelyssä ei ole tehty (El Emam & Garro 2000). On oletettavaa, että standardeita hankitaan monesta muusta syystä kuin jokapäiväiseen kehitystyöhön, mm. niitä käytetään oppimateriaalina tai niistä kehitetään varsinaiset yrityksen omat ohjelmiston kehitysohjeet. Standardien myyntimäärät eivät siten ole verrannollisia käyttömäärien kanssa, etenkään, kun useat ostajat ovat oppilaitoksista tai ostettu standardi ei ole soveltunut tarkoitettuun kohteeseen (Land 1997, 1999).

Standardin koulutuksellista merkitystä ei pidä väheksyä. Automaatiojärjestelmille soveltuvaa turvallisuusstandardia EN IEC 61508 on pidetty koulutuksellisena, ehkä enemmänkin kuin teknisenä standardina, vaikka siitäkin on jätetty sen alkuajoista (1980-luvun loppupuolelta) johdannolliset ja perustelevat jaksot "liian yleisinä" pois. Varsinaisena tavoitteena ei koulutuksellisuutta voida kuitenkaan pitää. Em. standardi on todelliseen tarpeeseen ohjelmiston turvallisuuden suunnittelun ja arvioinnin tukijana. Standardia hyödynnetään erityisesti sertifioitaessa järjestelmiä tietylle sovelluskohteelle. Sen esiversiot ovat jo olleet käytössä vuosia ennen varsinaisten standardien hyväksyntää ja ovat hyödyntäneet eri toimialoja sekä teknillisenä että käytännöllisenä lähteenä. Versiot ovat tulleet ajoissa, lopputuote kuitenkin jo niin myöhässä, että monet muut turvallisuusalan standardit ovat monin kohdin ehtineet edelle (mm. ilmailuala ja lääkintälaiteala).

Merkittävimmät tietyn standardin käyttämättömyyden syyt ovat saatavuudessa ja jonkun muun vastaavan standardin suosiminen (Land 1997), ehkä juuri sen takia, että tämä toinen on ollut tarvittaessa valittavissa. Valintaan vaikuttavat standardin liian aikainen valmistuminen (standardityössä ei ole vielä ollut kaikkea tarpeellista tietoa) tai liian myöhäinen valmistuminen. Spesifit ohjelmiston kehitystyön standardit eivät ole saa-

vuttaneet suosiota liiallisen kapea-alaisuuden takia. Niihin lukeutuvat mm. IEEE 1061 (1998) ja IEC 61713 (2000), jonka uudistamista IEC:ssä parhaillaan mietitään. Laaja-alainen standardi EN 61508 ei aivan aluksi ollut suosiossa, mutta hyvin pian vielä työversiona se oli jo laajassa käytössä. Sen suosiota aluksi kavensi sen liiallinen akateemisuus sekä monien ongelmien käyttäjille vieras ratkaisutapa. Koulutuksessa aina korkeakouluja myöten standardia käytetään hyvin yleisesti niin Suomessa kuin ulkomailla.

Monet ohjelmiston kehitysprosessien parantamista ja arviointia koskevat standardit ja ohjeet kuten CMM ja SPICE ovat saavuttaneet suhteellisen laajan käyttäjäkunnan, Bootstrap, Profes, TickIT ja Trillium hieman pienemmän. Sekä CMM että SPICE kehittyvät koko ajan. Kehittyminen kummankin kohdalla merkitsee laajentumista, esimerkiksi uudessa CMMI:ssä on n. 1 500 sivua tekstiä, mikä saattaa olla liian iso kynnys joillekin yrityksille tiellä ryhtyä hyödyntämään kyseistä kypsyysmallia. Kuitenkin jatkuva kehittäminen ja laajentuva käyttö ovat osoituksia kehitysprosessistandardien ja -ohjeiden merkittävyydestä.

Ohjelmiston mittareita on kehitetty jo niin paljon, että varmasti löytyy mittausmalli prosessille, tuotteelle ja projektille. On ehdotettu kojelautaa, josta käsin organisaatio tai projekti kykenisi jäljittämään ja ohjaamaan kulloistakin laadun tilaa. Mittareita on, ja koelautakin on rakennettavissa, mutta hankaluutena on kuitenkin päättää mitä mitata. Prosessista, tuotteesta tai projektista voidaan mitata satoja ominaisuuksia. Varmasti jostakin löytyy malli niiden kaikkien mittaamiseksi.

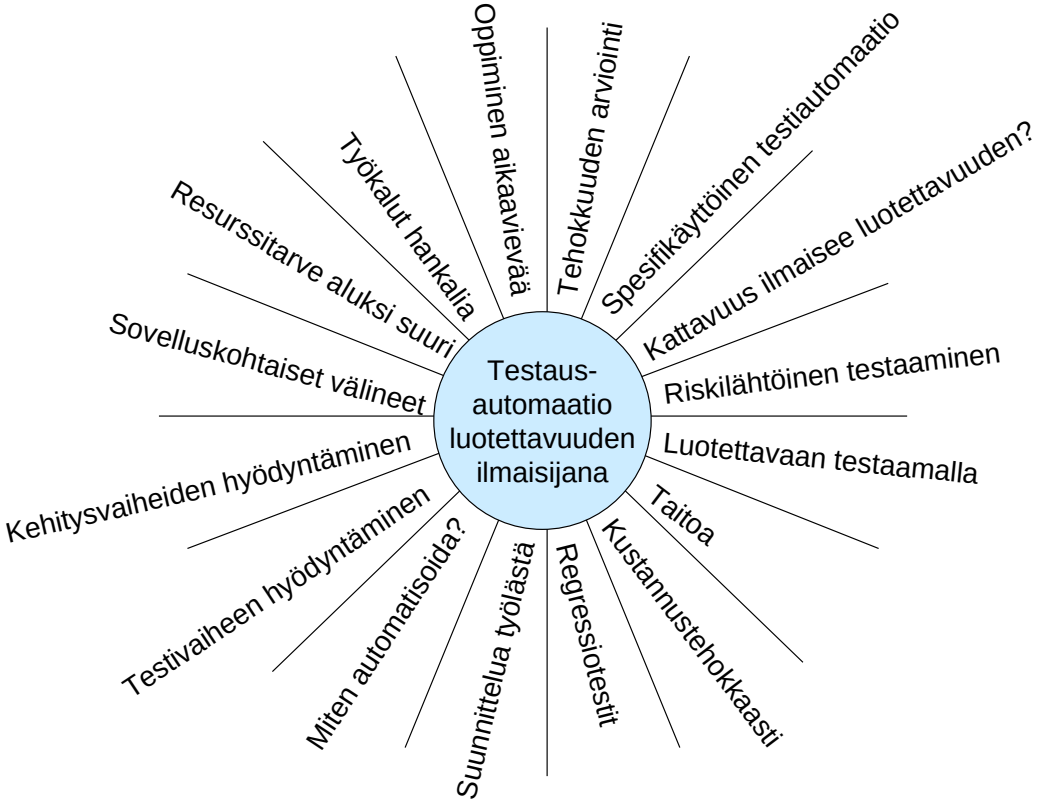
Mittaamisesta ja kehittyneistä mittareista ei kuitenkaan ole apua, jos yrityksen projektitiimeillä ei ole kunnollista mittauskulttuuria. Kulttuuri täytyy ensin perustaa, mikä vie aikansa. Tarvitaan peruskoulutusta ohjelmiston mittaamisesta, yksinkertaisia työkaluja alkuun pääsemiseksi sekä selkeät yhteydet mittareiden ja liiketoiminnan tavoitteiden välille. Mittaustietojen keruun pitäisi kuulua oleellisena osana ohjelmiston kehitysprosessiin, ei ylimääräisenä häiritsevänä tekijänä, mikä on omiaan vähentämään kiinnostusta mittareiden käyttöönottoon.

Ohjelmiston testaamiseen on laadittu useita malleja, jotka yksityiskohtaisesti ohjaavat kehitysprosessia tai arvioivat testaamisen kypsyyttä. Ne on tarkoitettu hyvin laajoille ohjelmistoille. Pienissä ja keskikokoisissa ohjelmistoyrityksissä ei yleensä ole kovin merkittävää testauskulttuuria, testauskoulutus puuttuu, testitapauksia ei laadita, testaamista ei dokumentoida eivätkä testit ole toistettavia. Useat näistä yrityksistä kuitenkin kykenevät määrittelemään testausten jälkeisen ohjelmiston laadun tai luotettavuuden. Kehittyneemmillä testaustavoilla ei niille ole merkitystä. Avuksi on kouluttautuminen testikäytäntöihin, testitapausten laadintaan, dokumentointiin jne., ei niinkään kokonaan uusien mallien hankkiminen.

3. Testausautomaatio luotettavuuden ilmaisijana

Ohjelmiston automaattinen testaaminen herättää korkeita odotuksia. Automatisoinnilla oletetaan vältettävän hankalat testisuunnittelut ja -ajot sekä tulosten raportoinnit ilman ihmisen puuttumista prosessiin. Testityökalut olisivat edullisia ja luotettavia sekä testaaminen toistettavaa. Kuvitellaan testaustyökalun parantavan ohjelmistotuotteen luotettavuutta kasvavana testikattavuutena ja vähentävän tuntuvasti testausaikaa jo heti ensimmäisestä käyttökerrasta.

Automaattinen testaaminen vaatii kuitenkin tekijöiltään harjaantumista ja ennen kaikkea valmistelutyötä. Ensimmäisessä projektissa kaikki työn osaset eivät vielä ole nivoutuneet yhteen, eikä testityökalujen investointikustannuksia saada yleensä takaisin. Syitä voi olla useita, keskeisimpinä ovat projektikiireet, jotka hankaloittavat kouluttautumista ja työvälineisiin tutustumista. Muita syitä voivat olla muutosvastarinta tai välinpitämättömyys – kaikki eivät käytä testausjärjestelmää tai viitsi opetella sen käyttämistä. Menettelyiden on kuuluttava yrityksen strategisiin päämääriin, kuten edellisessä luvussa todettiin.



Kuva 1. Testaamisen kattavuutta parannetaan automatisoimalla. Tehokkuuden tasapainoon vaikuttavat kuitenkin monet tässä luvussa käsiteltävät tekijät.

Projektidokumentaation puutteellisuus viittaa toistettavan ohjelmiston kehitysprosessin riittämättömyyteen, mikä on selkeä merkki automaattisen testaamisen perusedellytyksen puuttumisesta. Ilman ohjelmiston vaatimusmäärittelyitä testaaminen ei ylipäätään voi olla tehokasta, eikä automatisoinnilla kyetä silloin parantamaan tilannetta.

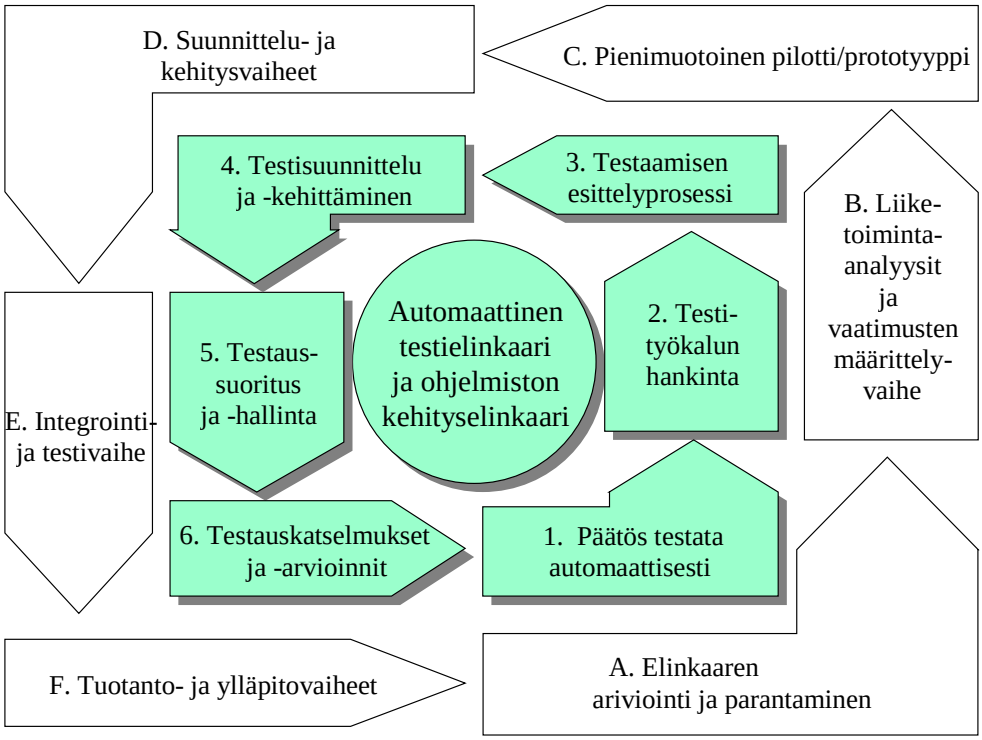
Automaattiseen testaamiseen siirtyminen ei siis ole itsestään selvää (kuva 1). Toisaalta yksikin merkittävä tunnistamattomaksi jäänyt ohjelmistovirhe voi tulla hyvin kalliiksi, kuten historian kokemukset osoittavat. Sellaisen virheen löytäminen saattaa maksaa tunnistamisprosessiin upotetut kustannukset nopeasti takaisin. Kustannusmenoja on kuitenkin muitakin kuin ikävät seuraukset epäonnisista virhetoiminnoista. Yksi merkittävimmistä on virheettömyyden osoittamisen kalleus etenkin tapauksissa, joissa joudutaan jatkuvasti uusimaan puutteellisia luotettavuuden ja turvallisuuden arviointeja.

Vaikka testaaminen onkin tärkein laadun ja myös luotettavuuden tarkastustapa, sekään ei yksin riitä. Monia muita suunnittelu- ja analyysimenetelmiä tarvitaan. Kriittisen järjestelmän suunnittelusääntöihin kuuluvat mm. eristäminen ja riippumaton toteutustapa. Järjestelmät suunnitellaan aina eristämällä kriittiset kohdat ja pitämällä ne riippumattomina. Testaaminen ja testausvälineiden soveltuvuuden arviointi kehittyvät ja testaamisen tärkeys tulee aina olemaan merkittävintä. Tässä luvussa ei niinkään tarkastella testaamista verrattuna muihin menetelmiin, vaan verrataan automaattista testaamista yleisesti muihin perinteisiin testauksiin nimeämättä erityisesti mitään manuaalista tai tietokonepohjaista menetelmää. Tarkastellaan seuraavan kahden hypoteesin pätevyyttä ohjelmistojen luotettavuuden ilmaisemisessa:

1) Automaattinen testaaminen on kattavampaa kuin muu perinteinen testaaminen ja siten mahdollisuudet löytää tehokkaasti kriittisiä virheitä ja virhetilanteita paranevat.

2) Automaattinen testaaminen tuo investoinnit takaisin ja on kannattavaa, jos sillä saadaan perinteisiä testausmenetelmiä edullisemmin selville kriittisen ohjelmiston virheettömyys.

Dustin et al. (1999) ovat kehittäneet ohjelmiston testauselinkaareen perustuvan automaattisen testausmenetelmän. Siinä kuvan 2 esittämällä tavalla testaamisen elinkaarien kohdat 1–6 vastaavat ohjelmistokehityksen elinkaaren kohtia A–F.



Kuva 2. Järjestelmän kehityselinkaaren ja automaattisen testaamisen elinkaaren välinen riippuvuus (Dustin et al. 1999).

Testauksen elinkaari vaihtelee ohjelmiston kehityksen elinkaaren tavoin yritys- ja tapauskohtaisesti. Dustinin et al. (1999) esittämä malli on ideaalinen, jota yritys joutuu sovittamaan tarpeisiinsa. Kahden elinkaarimallin yhteensovittamisesta riippuu ohjelmiston automaattisen testaamisen kustannustehokkuus ja merkityksellisten virheiden löytyminen. Täysin ei manuaalitesteistä voida luopua etenkään kriittisissä ohjelmistoympäristöissä.

Taulukko 1. Automaattisen testaamisen ydinkysymyksiä.

1.	Miten paljon enemmän yhden testitapauksen automaattinen testaaminen maksaa kuin sen manuaalinen testaaminen?
2.	Miten paljon enemmän yhden testitapauksen automaattinen testaaminen maksaa kuin sen manuaalinen testaaminen? Mikä pitäisi olla automaattisen testaamisen elinaika, jotta se tulisi kannattavammaksi kuin manuaalinen testaaminen?
3.	Kuinka todennäköisesti automaattisella testaamisella löydetään uusia merkittäviä virheitä ensimmäisen käyttökerran jälkeen?
4.	Mitä manuaalitestejä menetetään automaattisella testaamisella?
5.	Mikä on automaattisen testaamisen elinikä?

Testitapausten suunnittelua, suoritusta ja arviointeja automatisoidaan, mutta niissä toimenpiteiden on oltava hyvin määriteltyjä, ymmärrettäviä ja koettuja. Testaajan on oltava kokenut henkilö, mutta missä määrin hänen on tunnettava testaamisen erityisteoriaa tai soveltamista? Jos nämä eivät ole tiedossa, mikä vaikutus tällä on kriittisen järjestelmän kriittisten ohjelmistovirheiden löytymisessä? Automaattiseen testaamiseen siirryttäessä on ratkaistava useita ydinkysymyksiä (taulukko 1).

Teoreettisesti kaikki testausvaiheet voidaan automatisoida, mutta toisaalta edelleen teoreettisesti ajatellen ei tarvitse testata yhtään mitään. Ohjelmat tulisi koodata oikein jo heti ensimmäisellä kerralla. Pelkästään teorian avulla – siis teoriaa ehdottomasti tarvitaan – ei selvitetä testaamisen problematiikkaa, tarvitaan myös ja ensisijaisestikin käytännön päätöksentekotaitoa.

Siirryttäessä automaattiseen testaamiseen ei kannata automatisoida kuin tarkoin valitut kohteet. Taloudelliselta kannalta tarkasteltuna ensin etsitään ne manuaalisen testaamisen kohdat, joihin kuluu eniten aikaa ja jotka ovat ikävää ja virhealtista rutiinityötä. Jos taloudellisuus on merkittävää, automatisoidaan juuri ne osuudet sekä niistä vain toistettavaksi kelpaavat osuudet. Testityön ikävyyskään ei aina ole hyvä perustelu siirtyä automaattiseen testaamiseen, sillä automaattinen testaaminen saattaa olla virhealtista juuri tuon ikävän työn osalla.

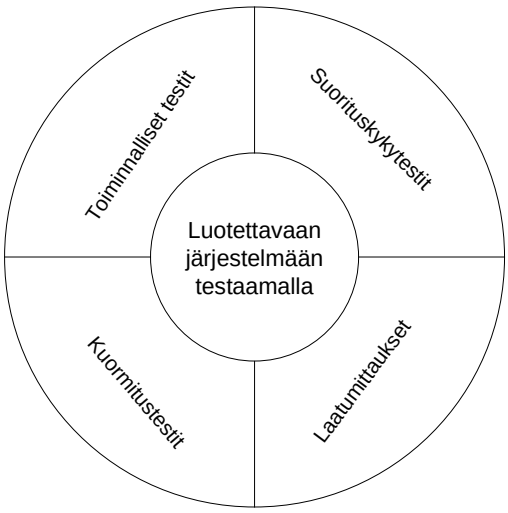
Testaaminen on tärkeämpi laadun tae kuin staattiset analyysit helppoutensa, käyttöympäristösoveltuvuutensa ja automatisoituvuutensa takia. Testauksilla ei kuitenkaan kyetä osoittamaan virheiden puuttumista.

Luotettavuuden kannalta, siis silloin kun jokin luotettavuusattribuutti on merkittävä, aikatekijä saattaa olla päinvastainen. Ei vähennetä testausaikaa, vaan panostetaan siihen. Projektissa täytyy järjestää aikaa erityiskohtien testaamiseen, ja jos aika on tiukalla, täytyy osata päättää mitkä osuudet eivät kaipaa niin täydellistä huomiota. Testaaminen on kuitenkin taitolaji, sillä useinkaan ei ole helppoa karsia testitapauksia, vaikka ne vaikuttaisivat miten turhilta tahansa. Virhe voi tulla kalliiksi.

Luotettavuusattribuuteista turvallisuuden ja toimintavarmuuden lisäksi erityisesti ylläpidettävyyteen panostaminen saattaa säästää aikaa ja vaivaa vähentyneinä muutostarpeina. Testitapaukset joko pystytään kätevästi muuttamaan uusiin tarpeisiin tai muuttaminen ei ole lainkaan tarkoituksenmukaista. Kun projekti vaihtuu, vanhoista automaattisen testaamisen proseduureista pitäisi jäädä paljon hyödynnettävää uuteen. Panostaminen vain taloudellisuuteen (mm. time-to-market) tai suorituskykyyn voi tulla kalliiksi.

3.1 Luotettavaan järjestelmään testaamalla

Testaamalla haetaan sovelluksesta poikkeamia. Mitä vähemmän virheitä, sitä lyhyempi käyttökeskeytysaika. Myös järjestelmän suorituskyvyn on oltava vaatimusten tai käyttäjän odotusten mukainen. Oikein suoritettuna automaattisella testaamisella kyetään parantamaan kehityselinkaaren ja testaamisen osa-alueita.



Kuva 3. Testaaminen on ohjelmiston luotettavuuden keskeisin osoittamistapa.

Toiminnallinen testaaminen on aina ollut testaamisen kulmakiviä (kuva 3). Luotettavan ja kustannustehokkaan järjestelmän rakentaminen alkaa vaatimusmäärittelystä, johon toteutusta toiminnallisin testauksin verrataan. Aikaisemmin piti testata yhä uudelleen ja uudelleen usean tietokoneen ja henkilön voimin suuria testimääriä tilastollisesti kelvollisten suorituskykylukujen aikaansaamiseksi. Automaattiset testilaitteet lukevat dataa tiedostoista tai taulukoista tai hyödyntävät työkalun generoimaa dataa. Testimakrot valmistetaan etukäteen siten, että testaaminen tapahtuu ilman manuaalisia toimenpiteitä.

Kuormitustesteissä järjestelmä altistetaan äärimmäisiin ja maksimaalisiin kuormiin ja testataan toimintoja yhtäaikaisesti. Tavoitteena on tutkia missä ääripisteessä järjestelmä pettää ja mikä pettää ensin. Manuaalisin testein kuormitustestit ovat hyvin kalliita, vaikeita, epätarkkoja ja aikaa vieviä. Tyypillisiä kuormitustesteillä havaittuja virheitä ovat muistin ylikuormittuminen, suorituskykyongelmat, jumiutumiset ja rinnakkaissuorittamisen ongelmat. Automaattinen testaus tuottaa laatumittoja ja sallii testausten optimoinnin. Automaattinen testausprosessi voidaan uusia samasta testimakrosta.

3.2 Testaaminen – taitolaji

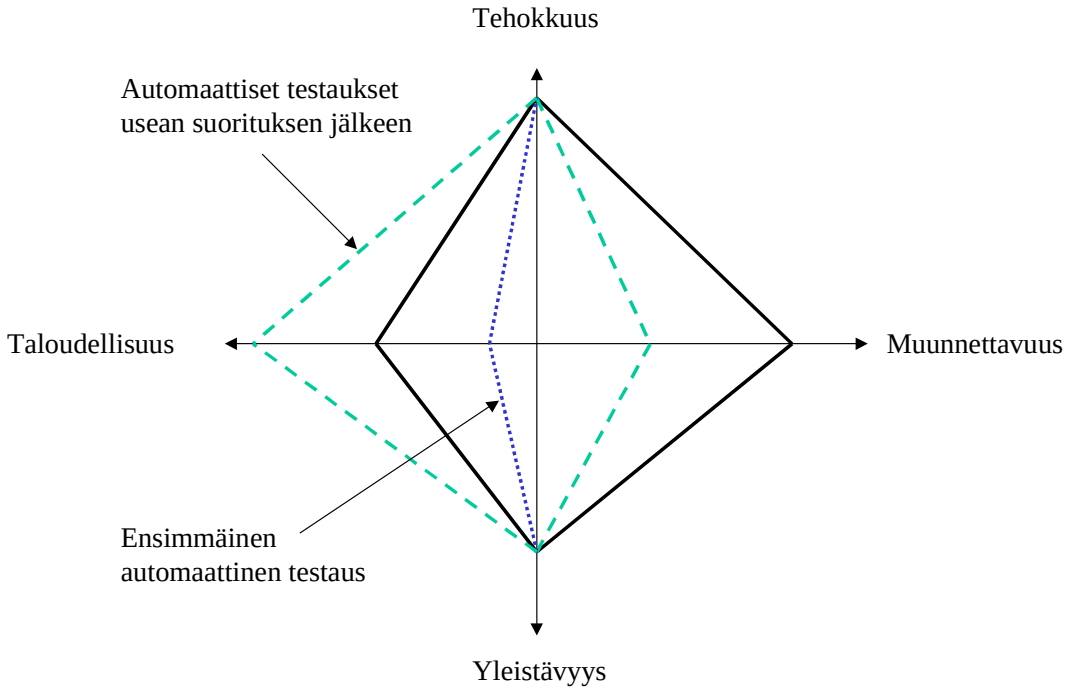
Testaaminen on taitolaji, jossa eniten taitoa tarvitaan jouduttaessa valikoimaan lähes äärettömästä joukosta testitapauksia vain rajoitettuun aikahaarukkaan tai suppeisiin resursseihin mahtuvat testimäärät. Koska supistetuilla testimäärillä tulisi kuitenkin löytää kaikki merkittävät ohjelmistovirheet, testisuunnitteluun on panostettava tavanomaista enemmän. Usein yhdenkin merkittävän virheen löytäminen tekee testauksesta kannattavan.

Kannattavuus ilmaistaan muullakin tavoin kuin merkittävien virheiden löytymisenä. Etenkin kriittisissä sovelluksissa ohjelmiston virheetön toiminta on kyettävä demonstroimaan. Näissä kohteissa ei ole todennäköistä, että ohjelmistopohjaisen järjestelmän hyväksymistesteissä löydetään ainoatakaan merkittävää kriittiseen toimintaan vaikuttavaa virhettä. Järjestelmän virheettömyyttä tämä ei kuitenkaan merkitse ohjelmiston osalta, sillä vaikka normaalitoiminta olisikin testattu riittävän kattavasti, virhetilanteiden testaaminen saattaa olla riittämätöntä. Ohjelmistolle on saatettu antaa vastuuta ohjelmiston virhetilanteen lisäksi laitteistovikojen ja käyttövirheiden kontrolloimisessa erilaisin vikasietoisin ratkaisuin, mutta toteutuksen verifiointi testaamalla ei ole yksinkertaista[3].

Mikä on hyvä testitapaus? Fewster (2000) vastaa tähän kysymykseen esittämällä seuraavat neljä hyvän testitapauksen ominaisuutta: 1) tehokkuus, 2) yleistävyys, 3) taloudellisuus ja 4) muunnettavuus. Niistä tehokkuus on tärkein testaamisen ominaisuus nimenomaan siinä mielessä, mitkä mahdollisuudet testitapauksella on löytää virheitä. Yleistävyydellä hän viittaa siihen, että testien tulisi suoriutua useammasta kuin vain yhdestä asiasta. Siten testitapausten määrää saataisiin vähennettyä. Taloudellisuus liittyy virheiden etsimiskustannuksiin ja muunnettavuus siihen, miten paljon lisäkustannuksia tarvitaan testitapauksen täsmentämiseen ohjelmistomuutoksen jälkeen.

Näiden neljän testausominaisuuden on oltava keskinäisessä tasapainossa (ks. kuva 4), sillä esimerkiksi jos yleistävyys on korkea, eli yhdellä testitapauksella suoritetaan lukuisia testejä, myös suoritus- ja analyysikustannukset ovat korkeita ja toisaalta muunnettavuuskustannukset kasvavat. Korkea yleistävyys puolestaan johtaa huonoon taloudellisuuteen ja muunnettavuuteen.

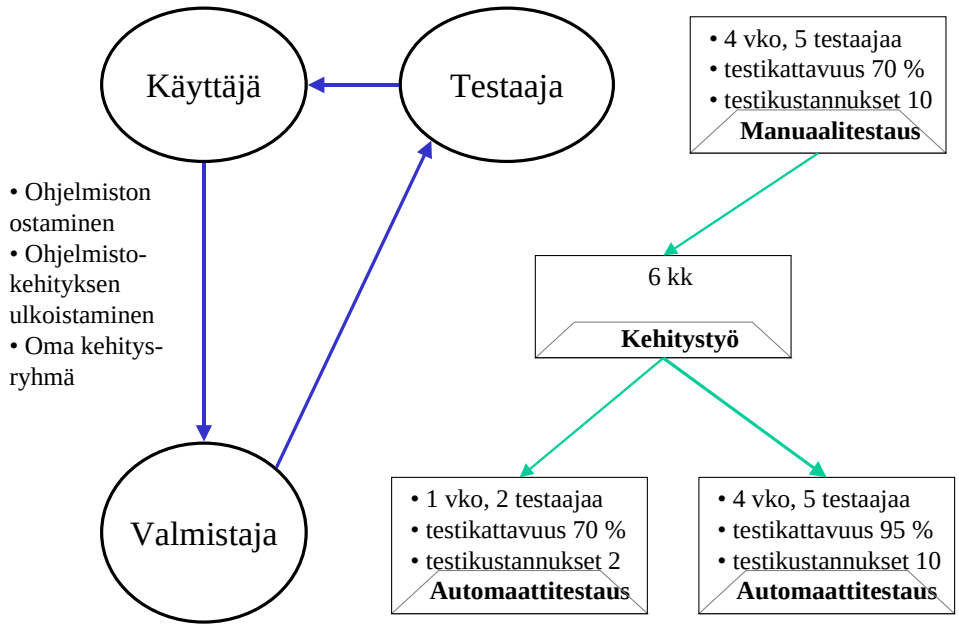
[3] Yksi tapa testata vikasietoisuutta on virhetilanteiden syöttömenetelmä.



Kuva 4. Testitapausten laatu neljän ominaisuuden avulla esitettynä. Mitä suuremman alan kaavio peittää, sitä parempi testitapaus (Fewster 2000).

Kuva 4 vakioi tehokkuuden ja yleistävyyden olettamalla, ettei testimenetelmän valinnalla automaattisen tai manuaalisen välillä olisi niihin vaikutusta. Testien automatisointi vaikuttaa vain taloudellisuuteen ja muunnettavuuteen. Kuva esittää vain ominaisuuksia testitapausta kohden, joten siinä ei verrata testikattavuutta automaattisen ja manuaalisen testaamisen välillä.

Testaaminen vaatii siis taitoa, sillä tavoitteena ei saa olla pelkästään virheiden löytäminen, vaan myös suurten kustannusten välttäminen. Fewster (2000) väittää automaattisen testaamisen eroavan tässäkin taitolajissa kertasuoritteisesta manuaalisesta testaamisesta. Useimmille yrityksille on tehokkaampaa laatia kertasuoritteisia tietokonepohjaisia testejä kuin automaattisia toistettavia testejä. Jos halutaan testata automaattisesti, täytyy olla selkeät näkemykset siitä, miten usein samoja testitapauksia tullaan toistamaan useamman kerran. Automaattisen testaamisen suunnittelu on eräiden karkeiden arvioiden mukaan 4–6 kertaa kalliimpaa kuin manuaalitestien vastaava suunnittelu (kuva 5).



Kuva 5. Automaattisten testausten kehitysaika on huomattavan suuri.

3.3 Testaaminen käyttäjän kannalta osoitettaessa ohjelmiston luotettavuutta

Käyttäjän kannalta testausten kehittäminen tai testausautomaatioon siirtyminen hyödyttäisi ainakin seuraavilta osin:

- mahdollistamalla testikattavuuden osoittamisen kustannustehokkaasti
- helpottamalla testauksen tuloksena syntyvien testiartefaktien hyödyntämistä
- painottamalla testausvaihetta edeltävien artefaktien sisältöä testaustarpeen suuntaan
- helpottamalla luotettavuuden osoittamista komponenteista koostuville järjestelmille ja
- helpottamalla perättäiskehittämisellä rakennettujen ohjelmistojen testaamista.

3.3.1 Testikattavuuden osoittaminen kustannustehokkaasti

Testaamisen tehokkuutta ilmaistaan testikattavuudella, mikä on testitapausten osuus kaikesta tarvittavasta testisyötemäärästä. Tarvittavan määrän määrittelevät laatu- ja luotettavuusvaatimukset. Kattavuudella täydellisyyden lisäksi tarkoitetaan myös merkittävien asioiden testaamista, siten ei yksin pyritä testaamaan mahdollisimman paljon vaan myös järkevästi. Dynaamisten testausten kattavuus on mahdollisimman laajan syötekombinaation testaamista. Lisäksi syötesekvensseillä on merkitystä erityisesti hajautetuissa reaaliaikaisissa järjestelmissä, joissa pienetkin syötesekvenssien muutokset tulee alistaa testauksille. Etenkin näissä testimäärät kasvavat suunnattomasti.

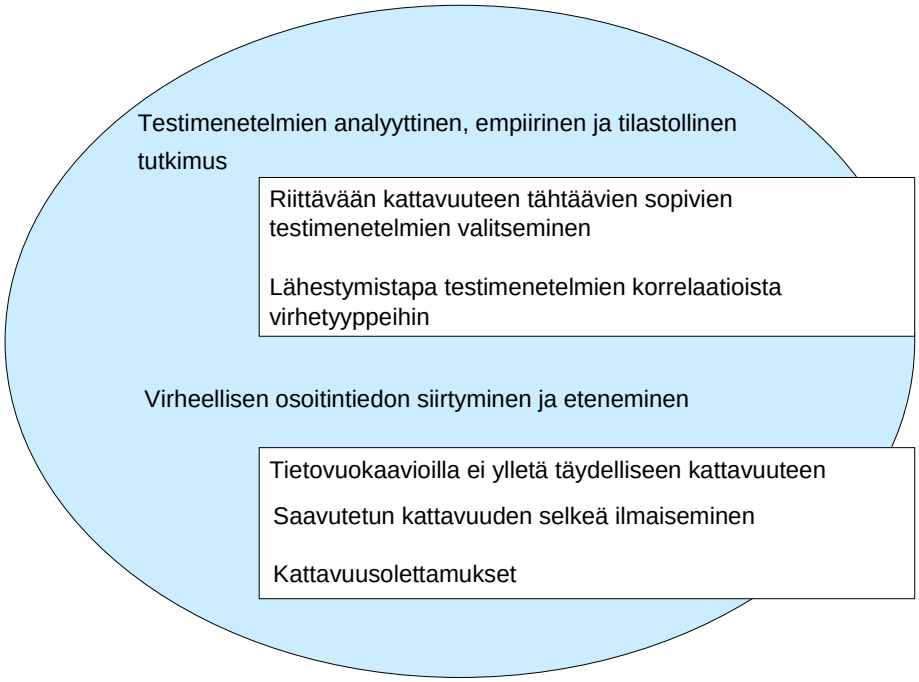
Testitapausten valitsemiseksi on kehitetty useita menetelmiä aina 70-luvun puolivälistä lähtien. Niitä valitaan mm. ohjelmistolle asetettujen tavoitteiden mukaan ilman, että toteutusta tunnetaan ja edelleen kaikissa vaiheissa vaatimusten määrittelystä suunnitteluvaiheiden kautta koodaukseen. Testitapauksia saadaan myös staattisista, erityisesti riskipohjaisista analyyseista, jotka korostavat merkittäviä testaustarpeita (taulukko 2).

Taulukko 2. Testaamisen tehokkuutta ilmaistaan testikattavuudella. Testitapauksen tehokkuus: miten hyvin virheet löytyvät.

Vaiheet	testitapausten valintakohteet
Tavoitteet	tarpeet, ohjattava kohde, ympäristö, systeemi, ohjelmisto
Kehitysvaiheet	vaatimusten määrittely, suunnittelu, koodaus, uudelleen
Analysoinnit	staattiset, formaalit ja riskipohjaiset, käyttökokemukset

Mistä kokonaisuudesta riittävä testikattavuus valitaan? Tiettyyn ympäristöön ja kohteeseen soveltamisessa tunnistetaan mahdolliset virhetyypit ja valitaan niihin sopiva menetelmä. Kaner (1996) esittää taulukon erilaisista testitapaustyypeistä. Toisaalta Beizer (1990) kuvaa ohjelmiston virhetopologian, jossa esitetään virhetyypit (mm. looginen virhe, toiminnallinen virhe) sen mukaan missä ohjelmistokehityksen elinkaarivaiheessa ne syntyvät. Kummastakin taulukosta saa hyvän käsityksen siitä miten laajasta ja monipuolisesta asiasta sopivien testitapausten valinnassa on kyse. Kaner luettelee 101 testikattavuustyyppiä. Taulukot sisältävät mm. tietovuoanalyyseilla[4] löydettävät virhetyypit.

[4] Tietovuoanalyysiin keskittyviä tutkimuksia on tehty runsaasti (mm. Laski & Korel 1983, Rapps & Weyuker 1985).



Kuva 6. Luotettavuuteen kohdistuva tutkimustarve testikattavuuden ilmaisemisessa.

Virheitten paljastaminen testeillä on ohjelmiston ja ohjelmistopohjaisen järjestelmän tärkein laadun osoittamistapa. Testien riittävään kattavuuteen tähtäävien sopivien testimenetelmien valitseminen on kuitenkin tutkimuskohteena jäänyt pahasti taka-alalle (kuva 6). Esitetään taulukoita virhetyypeistä, mutta kokonaisvaltaista lähestymistapaa testimenetelmien korrelaatioista virhetyyppeihin on vähän saatavilla. Tulisikin kiinnittää huomiota testimenetelmien sekä analyyttiseen, empiiriseen että tilastolliseen tutkimukseen.

Väärän tai korruptoituneen osoitintiedon[5] siirtyminen ja eteneminen saattavat olla luotettavuudelle merkittäviä. Tähän tietovuokaaviot ovat tärkeitä lähestymistapoja, ja niihin onkin kehitetty työkaluja, mutta läheskään täydelliseen analyysi- tai testikattavuuteen ei laajoissa ohjelmistoissa ylletä. Suorittaminen tulisi kalliiksi, mutta siitä huolimatta usein olisikin riittävää, että ilmoitettaisiin saavutettu kattavuus.

3.3.2 Testaustulosten hyödyntäminen

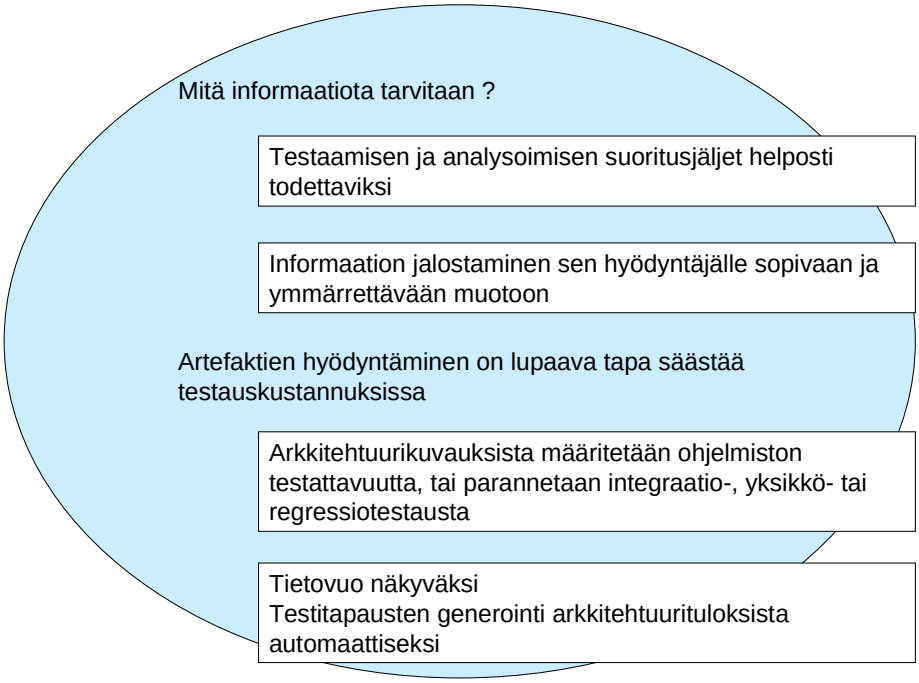
Testaaminen tuottaa useita artefakteja, joista on havaittavissa testissä suoritetun ohjelmiston suoritusjäljet. Jäljistä näkyvät mitkä ohjelmapolut on suoritettu, missä ohjelma-

[5] Pointer eli osoitin on muuttuja, jonka arvona on osoite.

käskyissä on käyty sekä mitä arvoja muuttujilla on ollut suorituksen aikana. Artefakteihin voidaan myös tallentaa tiedot testien läpäisystä.

Erityisen merkittävää testiartefaktien uudelleenkäyttäminen on integraatiovaiheen regressiotestauksissa. Niissä varmistutaan, etteivät moduuleihin tehdyt muutokset tai moduulivaihdot ole vaikuttaneet muihin ohjelmistonosiin. Virheellisen, ylimääräisen tai väärän datan eteneminen saattaa aiheuttaa piilevän virhetilanteen, jonka vaikutukset eivät ole ennalta tiedossa.

Testikattavuuden informaatioarvoa regressiotestien valitsemiseksi on tarkasteltu monella taholla. Rosenblum & Weyuker (1997) kehittivät menetelmän muutostöiden jälkeisen uusintatestaustarpeen arvioimiseksi. Sillä supistetaan ja priorisoidaan tarvittavaa testimäärää. Regressiotestattavuutta ja yleensä ylläpidettävyyttä on pyritty parantamaan tukemalla visualisoinnein testiartefakteja.



Kuva 7. Keskittyminen artefaktien hyödyntämiseen laadun ja luotettavuuden arvioimisessa.

Tutkimustulokset ovat vähäiset, mikä merkinnee ettei testiartefaktien hyödyntämiseen laadun tai luotettavuuden arvioinnissa ole kiinnitetty riittävää huomiota. Tarvittaisiin tutkimustuloksia, jotka selvästi osoittavat, että olemassa olevilla tekniikoilla kyetään antamaan hyödyllistä informaatiota sekä ohjelmistokehittäjille että luotettavuuden ja laadun arvioijille (kuva 7).

Toisaalta tarvittaisiin myös tutkimuksia siitä, mitä informaatiota ohjelmistokehittäjät ja arvioijat tarvitsevat testiartefakteista. Informaatio olisi varmasti erilaista eri ohjelmistokehityksen elinkaaren vaiheissa. Oli informaatio sitten mitä hyvänsä, varmasti tarvitaan sen jalostamista käyttäjälle sopivaan ja ymmärrettävään muotoon.

3.3.3 Ohjelmointia edeltävien tulosten hyödyntäminen testaamisessa

Integrointia, koodausta ja testausta edeltävät yleensä määrittely-, arkkitehtuurisuunnittelu- ja detaljisuunnitteluvaiheet. Ne kaikki tuottavat tietoa, jonka pohjalta voidaan suunnitella testaamista. Testisuunnitelmia voidaan myös kehittää automaattisesti, mikä edellyttää artefakteilta tiettyä formaalisuutta.

Testisuunnitelmat laaditaan yleensä joko koodin tai spesifikaation pohjalta. Koodipohjainen testaaminen ei verifiointi- ja validointimenetelmien joukossa ole tärkeimpiä tapoja osoittaa ohjelmiston luotettavuutta, sillä siinä testataan koodia verrattuna siihen itseensä. Koodipohjaisia testausmenetelmiä on runsaasti ja niillä on asemansa ja merkityksensä laadunvarmistusten joukossa. Yleisimpiä menetelmiä ovat white-box- eli glass-box-testit, joita käytetään kehitysprojektissa. Koodin kattavuuden mittaaminen eroaa koodipohjaisesta testaamista, ja vaikka näiden kahden menetelmän eroavuudella ei ole merkitystä manuaalitesteissä, automaattisissa testeissä ne on erikseen määriteltävä[6].

Spesifikaatiopohjainen testaaminen tarkoittaa testitapausten kehittämistä ohjelmiston vaatimusmäärittelystä. Spesifikaationa voi olla mikä hyvänsä kuvaus, jossa ohjelmistotuotteelta odotettavat toiminnot ja ominaisuudet kuvataan. Riippumattomassa testaamisessa testaajat saavat tottua erilaisiin dokumentteihin ja vaihteleviin formaatteihin. Mitä kriittisempi on kohde, sitä formaalimmat ovat dokumentit ja sitä helpommin kohde soveltuu automatisoitavaksi. Automaattiseen testitapausten generoimiseen suhtaudutaan kahtalaisesti. Generoinnit ovat toisaalta olleet suosittuja kohta kahden vuosikymmen ajan, toisaalta isotkin yritykset varovat niitä lähinnä vaadittavan manuaalityön suuren osuuden takia.

Vaatimusten määrittelyvaiheen keskeisyys testaamisen lähtökohtana korostuu lähinnä siksi, että vaatimusspesifikaatiossa olevat virheet siirtyvät suunnitteluun ja koodiin aiheuttaen mahdollisia tuotteen toimintahäiriöitä. Testaaminen itsessään ei tee spesifikaatiosta sen virheettömämpää, mutta niin manuaalisissa kuin automaattisissa testisuunnit-

[6] Koodin kattavuuden mittaaminen on evaluointia, mikä sopii sekä koodipohjaisen että spesifikaatiopohjaisen testaamisen evaluointiin.

teluissa asiaan voidaan puuttua. Kun testisuunnittelu suoritetaan rinnan määrittelyvaiheen kanssa, tarkistaminenkin tapahtuu mahdollisimman aikaisin ja riittävän kattavasti. Kaikki toimenpiteet, joilla kyetään estämään vaatimusvirheiden siirtyminen kehitysprosessissa muihin elinkaarivaiheisiin, kannattaa mahdollisuuksien mukaan tehdä. Testisuunnitelmien merkittävyys tässä vaiheessa korostuu erityisesti luotettavuusvaatimusten virheettömänä ja täydellisenä sekä kuvaamisena että toteuttamisena. Testisuunnitelmat toimivat siten myös demonstroitaessa turvallisuusvaatimusten täyttymistä. Jos välttämättä on tyydyttävä ohjelmistoprojektin vain yhden artefaktin tarkastamiseen – onneksi sellainen tilanne on vain esimerkkitapauksissa – dokumentin tulisi olla vaatimusmäärittely.

Riskipohjainen analyysi on yksi viime aikoina suosiota saavuttanut tapa osoittaa kriittisten kohtien riittävä luotettavuus tai laadukkuus. Siinä fokusoidaan yleisesityksistä tai vaatimusmäärittelyistä kriittiset toiminnalliset vaatimukset ja katsotaan myöhempien vaiheiden artefakteista, miten riskiä on onnistuttu vähentämään näissä kohdissa.

Arkkitehtuurivaihe on luotettavuuden kannalta monessakin mielessä merkittävä. Siinä tehdään luotettavuudelle tärkeitä ratkaisuja. Tiedon siirtyminen ja eteneminen voidaan visualisoida arkkitehtuurivaiheessa, mikä mahdollistaa virhekäyttäytymisen analysoimisen ja testitapausten suosittamisen. Käytön ja ylläpidon aikaiset muutostarpeet johtavat usein juuri muutoksiin arkkitehtuuridokumenteissa. Korjaukset ja modernisoinnit ovat virhealttiita koskiessaan kokonaisia ohjelmistoyksiköitä ja -komponentteja.

Tutkimuksissa on kiinnitetty huomiota mm. siihen, miten arkkitehtuurikuvauksista määritetään ohjelmiston testattavuutta tai parannetaan integraatio-, yksikkö- tai regressiotestausta. Näillä tavoilla pystytään aikaistamaan virheiden löytämistä dynaamisin analyysein.

Artefaktien hyödyntäminen on lupaava tapa säästää ohjelmistotestaamisessa. Testitapausten valitsemisessa tulisi erityisesti hyödyntää arkkitehtuurivaiheen tuloksia. Tietovuo tulisi saada näkyväksi. Testitapausten generointia arkkitehtuurituloksista tulisi automatisoida.

3.3.4 Komponenteista koostuvien sovellusjärjestelmien luotettavuuden osoittaminen

Käyttäjän ja sovellussuunnittelijan näkökulmat komponenteista koostuvien järjestelmien laatuun ja luotettavuuteen eroavat jonkin verran valmistajan näkökulmasta. Edellisten näkökulma on testaamalla saadut tiedot. Lähestymistyyli on sovellusriippuva, kun ohjelmistokomponenttien valmistajan lähestymistyyli on riippumaton sovelluskohteesta.

Jälkimmäisille on tärkeää saada tuote myytyä mahdollisimman moneen kohteeseen. Käyttäjälle on ensisijaisen tärkeää soveltaa niitä konfiguraatioita ja asioita, jotka ovat hänen käyttökohteelleen ominaisia.

Ohjelmakoodin saatavuus on yksi merkittävä ongelma sovelluskohtaisessa analysoinnissa ja testaamisessa. Valmistaja voi hyödyntää ohjelmakoodia, mitä etenkään COTS-ohjelmistojen osalta käyttäjä ei voi tehdä. Vaikka COTSien valmistusta ei tueta standardeilla ja ohjeilla, niitä tullaan käyttämään enenevässä määrin kriittisissä sovelluksissa.

Normaalisti testaukseen käytettyjä menetelmiä on yritetty laajentaa ohjelmistokomponenteille sopiviksi. Valmistajille tarkoitettuja testausmenetelmien parannusehdotuksia ovat mm.

- koodiin perustuvien testausten laajennukset
- algebrallisten määrittelyiden pohjalta tapahtuvat testilaajennukset
- olioiden tilan pohjalta tapahtuvat testaukset.

Komponenttien käyttäjän näkökulmasta on tehty joitakin tutkimuksia, mutta ne ovat selvästi vähemmistössä verrattuna valmistajan näkökulmaan. Tutkimuksissa on kehitetty teoriaa testien riittävyyden arvioimiseksi ja testikattavuuden määrittämiseksi puuttumatta komponentin sisäiseen informaatioon.

Käyttäjiä varten tarvitaan tutkimustietoa, jonka pohjalta voi kehittää ja valita uusia tehokkaita testimenetelmiä ja -työkaluja komponenteista koostuvien järjestelmien laadun ja luotettavuuden arvioimiseen. Luotettavuuden arviointi koostuisi kaikista luotettavuusattribuuteista; toimintavarmuus, käytettävyys, ylläpidettävyys, turvallisuus ja tietoturva ovat enimmäkseen käyttäjälle tärkeitä ominaisuuksia. Web-käyttöiset järjestelmät eivät ainakaan vähennä luotettavuusattribuuttien merkitystä. Kehitettävien tehokkaiden menetelmien tulisi antaa käyttäjille informaatiota, jonka perusteella he voivat luottaa sovelluksensa onnistumiseen liiketoiminta-alueillaan.

Tulisi myös selvittää mitä informaatiota komponentista käyttäjä tarvitsee selviytyäkseen komponenttiin liittyvistä testauksista. Käyttäjä voi esimerkiksi tarvita komponentin testikattavuustiedot vain tietylle sovelluskohtaiselle virhejoukolle. Käyttäjän tulisi kyetä informaation perusteella itse testaamaan komponentin virhejoukon määrittelemille testitapauksille. Toisaalta käyttäjä voi tarvita informaatiota, jonka pohjalta arvioida komponentin integroitumista sovellusjärjestelmässä.

3.3.5 Perättäiskehittämisen ohjelmistojen testaaminen

Regressiotestaus on tärkeä mutta kallis toimenpide, jolla varmistutaan siitä, ettei muutettu ohjelmisto sisällä uusia ohjelmistovirheitä. Regressiotestausta sovelletaan silloin, kun markkinatarpeet tai teknologia muuttuvat tai muutoin tarvitaan jatkuvaa ohjelmiston kehittämistä ja ylläpitoa. Erityisesti perättäiskehittämisessä, jossa tuote valmistetaan ja julkaistaan vähitellen lisäämällä toimintoja, regressiotestaukset ovat merkittäviä. Regressiotestaamisen kustannustehokkuuteen onkin kiinnitetty runsaasti huomiota. Rothermel & Harrold (1996) luettelevat seuraavia parannuskohteita:

- viimeisten testisarjojen ja suoritettujen testien informaation pohjalta laaditaan uudet testausvaatimukset muutetulle ohjelmistolle
- regressiotestattavuuden arviointi tekniikoiden valitsemiseksi
- testisarjakokojen kasvun hallinta
- ohjelmointivaihetta edeltävät regressiotestisuunnittelut.

Regressiotestaamista pitäisi nykyisestään tehostaa mm. ennallistamalla regressiotestien suunnittelua. Vaatimusmäärittelyvaihe tai arkkitehtuurisuunnitteluvaihe olisivat sopia aloitustasoja. Jo silloin laadittaisiin suuntaviivat tulevien ohjelmistoversioiden testaustarpeelle sekä myös testitapausten kehittämiselle. Ennallistamista olisi myös uudelleentestausta tarvitsevien ohjelmistokohtien aikainen tunnistaminen.

Priorisointi testitapausten valinnassa on vaihtoehto kiireisiin projekteihin, joissa ei ehditä testata uudelleen koko testisarjaa. Priorisointi voi tapahtua kattavuuden, kustannusten tai suoritusajan suhteen.

3.3.6 Mitä automatisoida?

Automatisointi on kannattavaa toistuville testeille. Investointikustannukset palautuvat uudelleen käytettäessä testiskriptejä. Regressiotestit ovat tärkeä automatisoinnin kohde edellyttäen, etteivät testiskripteihin vaadittavat muutostyöt ole suuria. Kertaluonteisia testejä ei kannata automatisoida lainkaan.

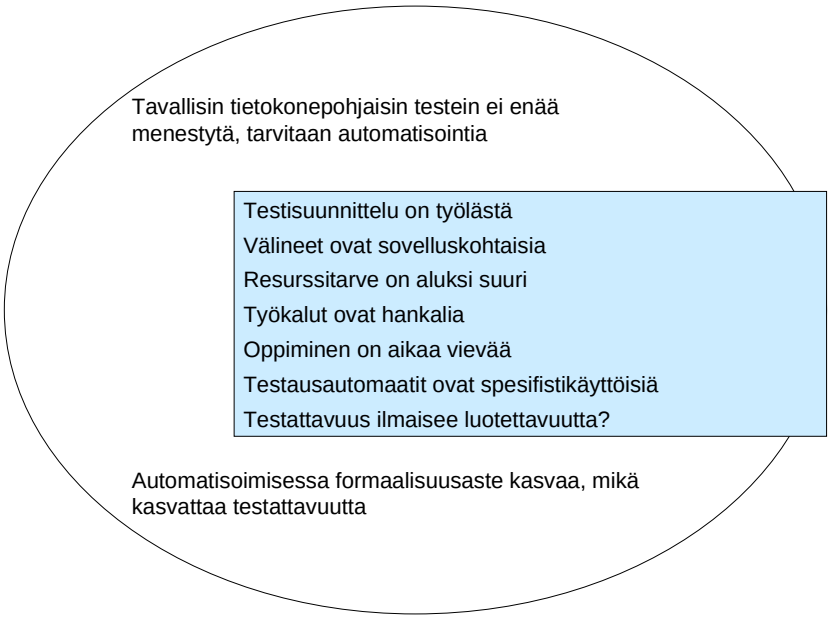
Automatisointi on kannattavaa, jos testaamiseen kyetään integroimaan analyyseja ja formaaleita menetelmiä testaamisen selvien etujen vielä häviämättä. Selvimpiä etuja ovat itse testaussuorituksen helppous ja yksinkertaisuus verrattuna manuaaliseen testaamiseen. Mutta jos esimerkiksi vaatimusmäärittelyä joudutaan formalisoimaan tai analysoimaan tietyillä välineillä, testaamisen selkeimmät edut saattavat hiipua.

Automatisointi kannattaa vasta kun ohjelmisto on stabiilissa tilassa, ts. ohjelmistokehityksessä ei ole nähtävissä lähiaikoina suuria muutostöitä. Muutokset merkitsevät testiautomaationkin muuttamista. Automatisointia ei kannata tehdä myöskään monimutkaista aikariippuvuutta sisältävälle testaukselle, etenkään jos se hoituu manuaalitesteillä nopeasti.

3.4 Automaattisen testaamisen problematiikkaa

Useiden automaattisten testausmenetelmien kehittäjien mielestä testityökalut on otettava ajoissa käyttöön, jo projektin määrittelyvaiheessa. Silloin ne hyödyntävät ohjelmistokehityksen elinkaarivaiheita, erityisesti arkkitehtuuria ja toteutusta, lisäksi kyetään entistä paremmin välttämään projektin loppuosuuksien kalliiksi tulevat korjaustarpeet.

Automaattinen testaaminen parantaa testikattavuutta, jota tarvitaan järjestelmien jatkuvasti kasvaessa ja monimutkaistuessa. Tavallisin tietokonepohjaisin testein ei enää menestytä. Lisäksi tietty formaalisuusaste, jota automatisointi aina edellyttää, myös osaltaan vaikuttaa testattavuuden parantumiseen. Siirtymisessä automaattiseen testaamiseen on kuitenkin havaittavissa esteitä kuten kuva (kuva 8) osoittaa.



Kuva 8. Testiautomaation hankintaprosessin päätöskohteita.

3.4.1 Työläs suunnitelmien generointi

Nykyisillä testaustyökaluilla ei kyetä perusteellisesti laatimaan testisuunnitelmia tai edes tukemaan testien suunnittelua. Testiproseduurien suunnittelemisen, luomisen ja suorittamisen työpaljous on yllättänyt monet projektipäälliköt. Testisuunnitelmien automatisointi edellyttää ohjelmiston vaatimusten määrittelyvaiheelle tehtäviä muutoksia, mitkä lähes poikkeuksetta ovat formaaliin suuntaan. Myös testitulosten tarkastelu ei usein ole niin automaattista kuin on kuviteltu. Automaattinen testaamisen on todettu vaativan laadunvarmistuksen asiantuntijoita tuekseen (Dustin et al. 1999).

3.4.2 Sovelluskohtaiset välineet

Useimmissa kirjallisuudessa referoiduissa testimenetelmissä ja -menettelytavoissa on kaksi merkittävää puutetta: 1) ei määritellä riittävän selvästi millä ehdoilla ja minkälaisiin järjestelmiin menetelmiä on tarkoitus soveltaa, 2) menetelmät ovat puhtaasti akateemisia ilman mahdollisuutta käytännöllistämiseen työkalutuen avulla.

Testityökalut soveltuvatkin vain tiettyihin käyttöympäristöihin (mm. Sun-työasemat, käyttöjärjestelmät UNIX, Windows 95/97/NT ja ohjelmointikielet: COBOL, C, C++, MS Access ja Visual Basic; client-server-teknologiat ja web-teknologiat). Myös eri ohjelmakielten vaikutukset testaamiseen ovat osoittautuneet yllättävän suuriksi (Daiqui 1996)[7]. Prechelt (2000) väittää lisäksi, että ohjelmakielen valinnalla on merkitystä ohjelmiston luotettavuuden kannalta, mutta merkittävyys tulisikin testaussuunnittelun helppoudesta, skriptistille kielille[8] kun on helpompi tehdä testitapauksia kuin ei-skriptisille, ”tavallisille” kielille[9].

Myös sovelluskohteen erityisominaisuudet haastavat testaajan. Valmari & Helovuo (2001) pureutuivat rinnakkaisten ja reaktiivisten järjestelmien[10] testaamisen vaikeuksiin kehittämällä automaattista testaamista:

---

[7] Daiqui (1996) kuvaa integroidun, modulaarisen, metriikkapohjaisen testausjärjestelmän, joka kohdentuu sekä isoihin että pieniin projekteihin. Testausten kohdentuminen tapahtuu myös luotettavuustason mukaisilla painotuksilla. Integroinnissa testausjärjestelmä kytketään sekä elinkaarimalliin että ohjelmiston sisältämään kokonaisjärjestelmään.

[8] Skriptisiä kieliä ovat mm. Perl, Python, Rexx ja Tcl.

[9] Ei-skriptisiä kieliä ovat mm. C/C++ ja Java.

[10] Reaktiivinen järjestelmä: yhtäaikainen toiminta usean liittymän kautta.

- Rinnakkaisuudelle pätevät satunnaisilmiöt, mikä merkitsee myös virheiden satunnaisuutta ja siten toistettavuuden vaikeutumista.

- Reaktiivisuuden takia käyttäytymisen määrittäminen on vaikeaa, sillä normaali toiminnallisuus input–output ei pidä paikkaansa ja usein samoista tuloista tulee erilaiset lähdöt (toiminnalliset testit hankalia).

3.4.3 Suuri resurssitarve alussa

Testiponnistelujen vähentäminen on yksi niistä päätekijöistä, miksi automaattisia testityökaluja yleensä halutaan hankkia. Toistettavuus paranee perinteisestä menettelystä, eikä väärinymmärryksille ole samassa määrin sijaa. Kuitenkin työkalujen käyttöönotto ei välittömästi johda testausresurssien pienenemiseen. Alussa työmäärä saattaa jopa kasvaa.

Uuden testausmenetelmän käyttöönotto merkitsee yrityksen testausprosessien komplisoitumista. Vanhasta siirrytään vain osittain uuteen, tarvitaan yhä manuaalisia testejäkin, sillä automaattisin testein voidaan vain osa työstä korvata. Kohteen tarkastelu on tärkeää juuri automatisoitaviksi sopivien testausten valitsemiseksi. Testaamisen painottaminen riskianalyysipohjaisilla menettelyillä kohdentaa testaamista, mutta sillä on samat ongelmat kuin automaattisilla testauksillakin. Kompleksisuus lisääntyy ja näkyy uusina opeteltavina työtapoina. Varhain tehdyt painotukset kuitenkin vähentävät virhealttiutta ja testaustarvetta sekä helpottavat vaatimusten toteuttamista määrittelyssä, suunnittelussa ja koodauksessa. Virheiden vähentyminen merkitsee kustannusten, ajankäytön ja uudelleen tehtävän työn vähentymistä.

Testien ja analyysien kohdistaminen ja automatisointi, ”kerralla valmista”, ja juuri oikeaan luotettavuuteen tähtääminen ovat keskeiset elementit pohdittaessa, miten luotettavuustekniikka voisi tukea testausponnisteluiden vähentämistä. Nämäkin toimenpiteet vaativat huolellista suunnittelua, hyvin määritellyn ja strukturoidun prosessin sekä pätevät ohjelmistokehittäjät.

3.4.4 Hankalat työkalut

Testiautomaatti ei heti generoi valmista komentokielistä testimakroa eli -skriptiä, vaan skripti vaatii aina manuaalisen käsittelyn. Mitä paremmat valmiudet yrityskohtaisin menetelmin on testausprosessiin, sitä robustisemmaksi, uudelleenkäytettävämmäksi ja ylläpidettävämmäksi testiskriptit saadaan. Testiskriptien muuttaminen vaatii perehtymistä komentokieliseen ohjelmaan.

3.4.5 Aikaa vievä oppiminen

Uuden testausmenettelyn luomisessa lähdetään vanhan kehittämisestä liikkeelle tai kehitetään kokonaan uusi testausprosessi. Testiryhmien ja mahdollisesti kehitysryhmienkin täytyy oppia uudet menettelytavat, mikä lisää kynnystä aloittaa uudistumisprosessi. Gormley et al. (1995) kehittivät automaattisia testimenetelmiä finanssi- ja vakuutusalan yrityksille. Tavoitteena oli laatia malli, joka tukisi testausten yrityskohtaista räätälöintiä. Heidän kokemustensa perusteella automaattinen testaaminen on kytkettävä integroiduksi osaksi ohjelmiston elinkaarimallia, mikä johtaa elinkaarimallin uudelleenkehittämisen lisäksi väistämättä merkittävään koulutustarpeeseen. Ilman menetelmäkoulutusta parhaita tai riittävän sopivia käytäntöjä ei onnistuta omaksumaan yritystason laajuudessa.

Testaaminen voidaan kuitenkin keskittää osaaviin käsiin: lokakuussa vuonna 2000 perustettiin Suomen testauskeskus Savonlinnaan. Myös ohjelmistosertifiointi on ollut jatkuvasti esillä, mutta tiettävästi laihoin tuloksin. Myös testauksen etäkäyttöä on tarkasteltu.

Niemi (2001) oli kehittämässä menetelmiä, jotka nopeuttavat tuotekehitystestausta ja pienentävät tuotekehityksen kokonaiskustannuksia samalla, kun tuotteen laatu ja testausvaiheen seuranta paranee. Menetelmät kohdistuvat seuraaviin asioihin:

- Testausjärjestelmien käyttöä tehostetaan etäkäytön avulla, sillä järjestelmät ovat kalliita ja keskittävät käyttö- ja asiantuntijahenkilöstön testauspisteeseen koko testaustapahtuman ajaksi.

- Testaussuunnitelman tekstuaalista esitystapaa parannettiin siten että se kuvaa testausympäristön, testitapaukset ja niiden suoritusjärjestyksen. Lähtökohtana projektille oli yhtenäisen määrämuotoisen testaussuunnitelman esitystavan puuttuminen, testitapausten uudelleenkäytön hankaluus ja testiohjelmien laadinta vain manuaalisesti.

3.4.6 Spesifikäyttöinen testausautomaatio

Kaikkia sovelluskohteita ei kyetä automaattisesti testaamaan. Esimerkiksi sovellettaessa ensimmäistä kertaa GUI-testereitä (Graphical User Interface) kannattaa suorittaa sovelluskohteen yhteensopivuustestit niin että kaikki kohteen oliot tai moduulit ja kolmannen osapuolen kontrollit kyetään tunnistamaan.

Kolmas osapuoli laatii mm. ActiveX-kontrollit Windowsin käyttöliittymissä. Useimmat testausjärjestelmät eivät pysty pitämään yllä satoihin nousevaa erilaisten kontrollien

määrää. Seurauksena saattaakin olla, ettei testityökalulla havaita kaikkia kontrollikohteita työkalun näytöllä.

Ohjelmiston vaatimusten tiheä muuttaminen on merkittävä luotettavuutta vaarantava tekijä. Erityisesti virhealtista on perättäiskehittäminen, missä uuden tuotteen konstruointi tapahtuu lisäämällä ominaisuuksia ja toimintoja vanhaan tuotteeseen. Lisäominaisuudet ensisijaisesti muuttavat arkkitehtuuria ja onnistuakseen myös vaatimusmäärittelyä on muutettava. Virhealttius ilmenee jo testaus- ja ylläpitovaateiden kasvamisena.

3.4.7 Ilmaiseeko testattavuus testaamisen luotettavuuden?

Nykyisten järjestelmien testaaminen on päättymätön prosessi. On mahdotonta testata kaikkia syötteitä, syötteiden kombinaatioita tai muutoksia. Kaikki arvot, sekä kelvot että epäkelvot, tulisi testata ennen kuin saadaan selville mahdolliset merkittävät ongelmat ohjelmiston toiminnoissa tai sen tarjoamissa palveluissa. Edes kohtuullisen monimutkaisen järjestelmän kaikkia polkuja ei kyetä käymään läpi perusteellisesti. Testaaminen alkaa vaatimuksista ja ilman hyviä lopetussääntöjä se ei pääty koskaan.

Ohjelmistot ovat sitä laadukkaampia ja robustisempia mitä intensiivisemmin ne on testattu. Ohjelmistokehitys helpottuu ja ohjelmiston suorittaminen halpenee. Automaattiset testiproseduurit voivat olla perusteellisempia ja kattavampia kuin perinteiset menetelmät, mutta ne vaativat enemmän valmistelua. Ylimääräinen panostaminen valmisteluun hyödyttää vasta, kun muutostarpeita tulee paljon käyttövaiheen aikana (Gaburro 1996).

Testausten määrää rajoittavat sekä aika että kustannukset. Jotkut testit on edullisempaa tehdä perinteisesti kuin automaattisesti. Siksi mm. vain kerran suoritettavat testit eivät ole kannattavia automatisoida. Testattavuus tulisi selvittää jo vaatimusmäärittelystä, mikä edellyttää kaikilta vaatimuksilta yksiselitteistä ja yhtäpitävää esitystapaa. Testaajan tulee saada kaikki tarvitsemansa tieto tavoiteltaessa korkeaa testikattavuutta.

Ohjelmiston testattavuus kuvaa sitä, miten helposti tietokoneohjelma kyetään testaamaan. Tai miten vaikeasti, sillä testaamista ei yleensä ole suunniteltu helpoksi, vaikka erilaisia keinoja[11] onkin kehitetty ohjelmistotuotannon historian alusta saakka. Testattavuudellekin löytyy mittoja, joilla pystytään kuvaamaan testaamisen ominaisuuksia, ja joskus testattavuus määritellään testikattavuutena eli mittana siitä, miten riittävästi jotkut testiosuudet kattavat tuotteen. Testikattavuus käsittää silloin toiminnallisuuden ja

[11] Tarkistuslistat, ylimääräiset käskyt tärkeisiin ohjelmakohtiin jne.

suorituskyvyn sekä muut kokonaislaatuun kuuluvat ominaisuudet kuten havaittavuuden, hallittavuuden, yksinkertaisuuden, stabiilisuuden ja ymmärrettävyyden.

Ohjelmistokehittämisen ja -tuottamisen nopeutuessa vaaditaan testeiltä yhä enemmän. Jotta taloudellisuus, suorituskyky ja luotettavuus pysyisivät tasapainossa automaattisin testein, tulisi hallitusti korottaa testikattavuutta tiukasti määriteltyihin ja toistettaviin kohteisiin unohtamatta manuaalisen työn tärkeyttä.

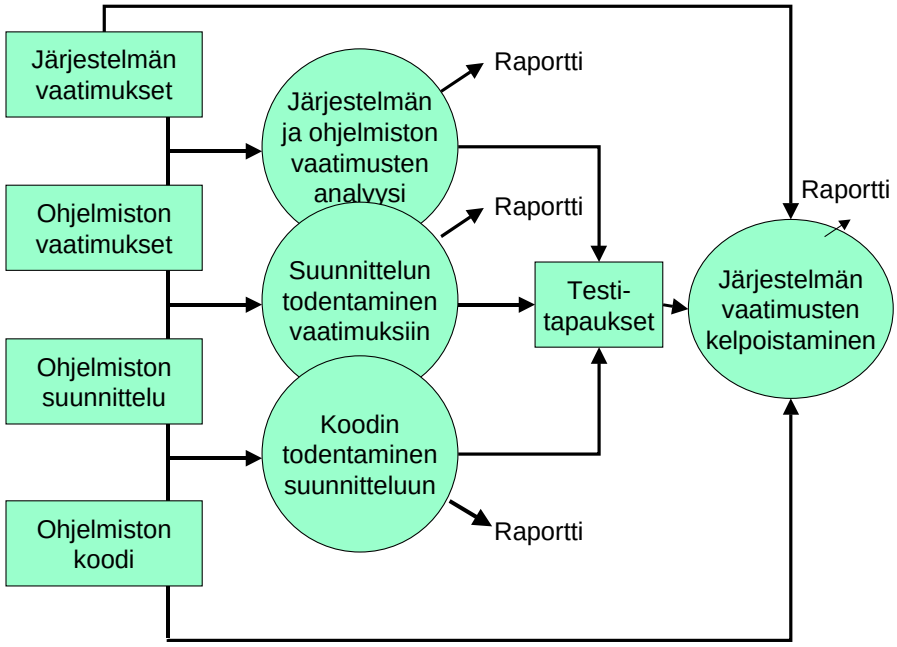
3.5 Automaattisella testaamisella yksinkertaistetaan ohjelmiston verifiointia ja validointia

Verifioinnilla ja validoinnilla (V&V) selvitetään laadun asemaa ohjelmiston kehitysprojektissa. V&V-menettely on kuitenkin kohtalaisen raskas, eikä siksi sovellu kuin ohjelmistoille, joiden laatutavoitteet ovat suhteellisen korkeat. Huono laatu voi aiheuttaa niin ongelmia henkilö- ja ympäristöturvallisuudelle kuin suoraan rahassa mitattavia omaisuus- tai liiketoimintamenetyksiä, puhumattakaan imagoa kolhivista julkisuusarvoista. V&V-menettelyä kyetään kuitenkin keventämään automatisoimalla menettelyyn kuuluvia analyyseja ja testauksia. Automatisoinnin etuihin tulisi myös oleellisena osana kuulua luotettavuuden osoittaminen siitä kiinnostuneille osapuolille.

3.5.1 Vaatimuslähtöinen verifiointi- ja validointimenettely

Jotta V&V-menettely saadaan kustannustehokkaaksi, sitä on rajoitettava ja kohdistettava vain kriittisiin asioihin. Kohdistamiseen on käytettävissä useita menettelyitä, kuten tilastolliset, riskipohjaiset ja käyttöprofiilit, mutta yleensä ne kaikki pohjautuvat vaatimusmäärittelyyn. Tosin vaatimusmäärittelyn puuttuessa kelpaavat projektiosapuolten haastattelut ja saatavissa olevat projektiasiakirjat, joista selviää mitä ohjelmiston kuuluu tehdä ja kuinka hyvin. Kuva 9 esittää erästä lukuisista V&V-malleista. Kuvauksessakin korostuu ohjelmistoprojektin määrittelyn tärkeys. Siitä myös selviää, mikä on testaamisen merkitys ohjelmiston kelpoisuudesta päätettäessä.

V&V-menettelyn kulmakiviä on vaatimusmäärittely. Vaatimusmäärittelyn kelpoisuudesta (mm. virheettömyydestä ja oikeellisuudesta) riippuu V&V-menettelynkin kelpoisuus. Jos määrittelyssä on puutteellisuuksia, niiden havaitseminen on hankalaa, aina ei edes mahdollistakaan. Kuten edellä mainittiin, V&V-menettelyn kohdistaminen pohjautuu lähes kaikissa tapauksissa vaatimusmäärittelyyn tai sitä korvaaviin tietoihin. Kohdistamismenettelyillä tuetaan kehitysprosessin heikkoja kohtia, mutta vaatimusmäärittelyn keskeisyydestä voi tulla menettelyn kompastuskivi. Erityisesti luotettavuusattribuuttien osalta kohdistamismenettelyn ulkopuolelle voi jäädä merkittäviä virhelähteitä.



Kuva 9. Eräs malli todentamis- ja kelpoistamisprosessille.

Yleensä V&V-menettelytapaa tehdään riippumattomasti kehitysryhmän työstä. Riippumattomuudella saavutetaan selkeitä hyötyjä, mm. erilaisen näkökulman tuomisessa ongelma- tai vikatarkasteluun sekä ennen kaikkea riippumaton V&V-ryhmä tuntee testaamisen ja analysoinnin erityispiirteet, mitkä eivät välttämättä ole kehitysryhmän hallussa[12]. V&V-menettely tulisi sovittaa sekä työn tarkkuuden että laajuuden osalta ohjelmistoprojektiin, jolloin ohjelmiston kriittisyydestä, monimutkaisuudesta ja kypsyysasteesta tulee keskeiset tekijät. Täydellisesti suoritettuna riippumattoman osapuolen V&V-menettely voisi sisältää seuraavia tehtäviä, joista selvästi näkee vaateet kehitysryhmän suuntaan:

1. Ohjelmiston kehitysmenetelmien arviointi

   – Tunnistetaan esimerkiksi koekäyttämällä tai analysoimalla, miten menetelmät soveltuvat vaihteleviin kohteisiin.

[12] Joissakin tapauksissa sallitaan ryhmien välinen vuorovaikutus, etenkin jos kummankin osapuolen prosessit täydentävät toisiaan, mutta näkemysmenetysten pelosta vuorovaikutteisuutta ei yleensä hyväksytä. Vuorovaikutteisuus voi aiheuttaa ongelmia ja mahdollisesti projektin viivästymistä, mutta yleensä niistä selviää pitkäaikaisen yhteistyön tuomalla kokemuksella.

2. Vaatimusten tunnistaminen ja jäljittäminen

   - Analysoidaan kaikki järjestelmän spesifikaatiot ja vaatimusmääritellyt ohjelmistoon vaikuttavien vaatimusten tunnistamiseksi.

   - Jäljitetään vaatimukset eteenpäin suunnitteludokumenttiin ja ohjelmakoodiin tarkkailemalla samalla vaatimusten oikeaa, johdonmukaista ja täydellistä allokoitumista.

   - Jäljitetään vaatimukset taaksepäin koodista vaatimuksiin siten, että kutakin koodikohtaa vastaa ainakin yksi vaatimus.

3. Testitapausten kehittäminen

   - Kehitetään testivaatimukset jokaiselle tunnistetuille prioriteetiltaan merkittäville ohjelmiston vaatimuksille.

4. Todennustestit

   - Todennetaan, että ohjelmiston vaatimukset ovat täydellisiä, oikeita ja yhtäpitäviä järjestelmätason vaatimusten kanssa.

   - Todennetaan, että ohjelmiston ulkoiset ja sisäiset liittymät vastaavat tunnistettuja vaatimuksia ja järjestelmän laitteistoa.

   - Todennetaan, että suunnittelu vastaa vaatimuksia.

   - Todennetaan, että ohjelmakoodi vastaa suunnittelua sekä logiikan, algoritmien että datan suhteen. Kunkin ohjelmamoduulin osalta tarkistetaan myös haarautumat, yksikkötesti ja rasitustestit.

5. Kelpoistustestit

   - Kelpoistetaan integroitu koodi järjestelmämäärittelyjä ja tunnistettuja vaatimuksia vasten. Niihin kuuluvat kuormitustestit ja vioittumistestit.

6. Laadunohjaus

   - Tarkistetaan dokumentoitu laatujärjestelmä, erityisesti ongelmien kirjaamisen ja jäljittämisen osalta.

7. Tuotehallinta

   - Tarkistetaan, että tuotehallinta sisältää kaiken dokumentti-, tietokanta-, ja ohjelmakoodiversioinnin ja versioinnin testidatasta, -tuloksista ja koodimuutoksista.

8. Datan hallinta

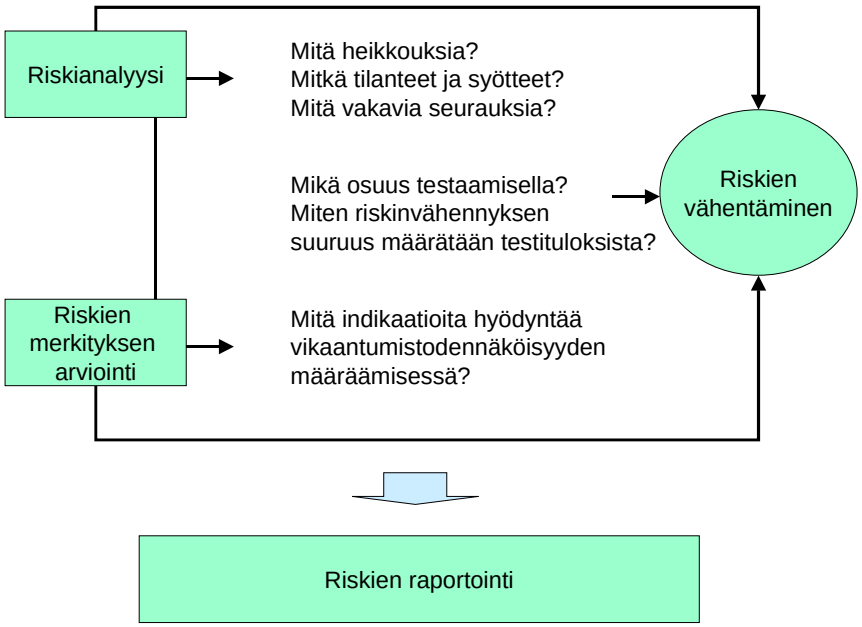
   - Tarkistetaan, että menettelyillä saadaan itsestään selville, jos jokin vaatimus tai testi ei täyty.

Kehitysryhmästä riippumattomalla vaatimusvetoisella V&V-menettelyllä tunnistetaan, jäljitetään ja testataan kaikki tai ainakin merkittäviksi priorisoidut vaatimukset. Menettely saattaa olla aika laaja ja monimutkainen, josta syystä sitä on syytä yksinkertaistaa automatisoimalla joitakin tehtäviä. Automaattisen työkalun hoidettavaksi tulisi ainakin seuraavat tehtävät, jotka ovat etupäässä työläitä ja siten myös virhealttiita rutiineja:

- jokaisen vaatimuksen tunnistaminen
- kunkin vaatimuksen jäljittäminen aina koodiin asti ja koodikohtien jäljittäminen vaatimuksiin
- todennetaan, että vaatimus on toteutettu
- vaatimusten testimenetelmän tunnistaminen
- testitulosten itsenäisen läpäisyn kirjaantuminen
- merkintä vaatimusten täydellisestä toteuttamisesta ja testaamisesta.

3.5.2 Riskilähtöinen testaaminen

Tässä kohdassa tarkastellaan, miten riskienhallintaprosessilla kohdennetaan ohjelmiston testaamista. Kohdentamisen päämääränä on projektin testivaiheen kustannusten vähentäminen optimoimalla testiprosessia ohjelmistotuotteen koko elinkaaren aikana.



Kuva 10. Riskien vähentämisessä oleellinen osuus kuuluu testaamiselle. Vähentämisen suuruus on kuitenkin vaikeasti pääteltävissä.

Riskienhallinnalla tarkoitetaan päätöksentekoprosessia, jossa päätetään hyväksyttävän riskin tasosta ja menetelmistä, joilla asetetut tasot sovelluskohtaisesti saavutetaan. Lisäksi riskienhallintaan kuuluu asetettujen tasojen saavuttamisen valvonta. Riskienhallintaprosessi kattaa koko tuotteen elinkaaren. Siihen sisältyvät seuraavat kohdat (kuva 10):

1. *Riskianalyysi* on jäsennelty prosessi, joka tunnistaa tuotteen käyttötarkoituksen ja vaarat sekä arvioi vaaroista johtuvien riskien suuruuden.

2. *Riskien merkityksen arvioinnissa* päätetään jokaiselle tunnistetulle vaaralle arvioidun riskin suuruuden perusteella mahdollisista riskien vähentämistoimenpiteistä.

3. *Riskien vähentämisessä* poistetaan tai vähimmäistetään riskiin johtaneita syitä tai vältetään riskien vaikutuksia.

4. *Riskien raportointi* on jäsennelty prosessi tarvittavan riskin vähentämistoimenpiteiden ohjaukseen ja valvontaan siten, että jokaiselle vaaralle jäännösriski on hyväksyttävä.

Riskianalyysissa aluksi tarkastellaan, mikä voi tuotteessa epäonnistua. Täsmällisemmin tulisi tarkastella tuotetta osa osalta:

- Mitä heikkouksia osassa on?
- Mitä sellaisia tilanteita tai syötteitä on olemassa, että osan heikkoudet pääsevät vallitseviksi?
- Mitä vakavia seurauksia heikkouksilla voi olla ihmisille, ympäristölle, omaisuudelle tai tuotannolle ja miten pahoja seuraukset ovat?

Riskien tunnistamiseen kuuluu tiedon kerääminen projekteista (esim. järjestelmä- ja ohjelmistoprojektit), mikä voidaan systematisoida eri projektivaiheille. Voidaan selvittää osan toiminnot esimerkiksi toiminnallisesta vaatimusmäärittelystä ja haastattelemalla määrittelijää tai suunnittelijaa. Toiminnosta selvitetään mm. voiko se käynnistyä väärällä hetkellä, onko vääräaikaiseen käynnistykseen itsetarkistuksia, mitä seurauksia siitä on jne. Selvitetään, voiko määrätty tieto korruptoitua ja mitä seurauksia siitä on paikallisesti ja muualla järjestelmässä.

Dokumenttipohjainen selvittäminen ei yleensä riitä ellei tarkastaja ole aihealueen erityisasiantuntija. Haastattelun merkitystä korostaa esim. kaavioiden täydellisyyden arvioiminen. Kyselemällä suoraan tekijältä saadaan selville, mitä jokin kaavio tai kuva esittää ja erityisesti mitä siitä on jätetty pois. Tuotteeseen kohdistetaan laatukriteereitä, jotka johtavat tiettyjen haavoittuvuuskohtien selvittämiseen sekä tuotteen uusien vaatimusten määrittelyyn. Laatukriteerit voidaan luokitella usealla sovelluskohteesta riippuvalla tavalla. Tyypillisiä kriteereitä ovat mm. seuraavat:

- Luotettavuus, mikä ilmaisee sen, miten hyvin kohde toimii myös ohjelmistovirheiden läsnä ollessa.
- Kyvykkyys, mikä ilmaisee sen, pystyykö tuote suoriutumaan vaatimuksista.
- Käytettävyys, mikä ilmaisee sen, miten helposti käyttäjä pystyy käyttämään ja hallitsemaan tuotetta.
- Suorituskyky, mikä ilmaisee tuotteen nopeuden ja herkkyyden.
- Testattavuus, mikä ilmaisee sen, miten tehokkaasti tuote kyetään testaamaan.
- Ylläpidettävyys, mikä ilmaisee sen, miten taloudellisesti tuote kyetään korjaamaan ja parantamaan.
- Siirrettävyys, mikä ilmaisee sen, miten hyvin tuote sopii eri teknologioihin.

On olemassa riskejä, jotka vaikuttavat kaikkiin kohteisiin. Yleisiä riskitekijöitä ovat mm. seuraavat:

- Kompleksisuus, mikä on riski suhteettoman suurelle, hankalatöiselle tai sekavalle.
- Uutuus, mikä tarkoittaa sitä, että ei ole kehityshistoriaa.
- Muutettu, mikä tarkoittaa kaikkea korjattua tai parannettua.
- Induktiivinen riippuvuus, jossa virhetoiminta johtaa perättäisiin järjestelmätason virhetoimintoihin.
- Deduktiivinen riippuvuus, joka on erityisen herkkä muiden järjestelmäosien virhetoiminnoille.
- Kriittisyys, jossa virhetoiminto voi aiheuttaa huomattavaa vahinkoa.
- Tarkkuus, jossa vaatimuksia on noudatettava täsmällisesti.
- Strategisuus, joka on yrityksen liiketoiminnalle merkittävä.
- Virhealttius, joka sisältää kaikki minkä tiedetään merkitsevän ongelmia.

Riskien merkityksen arviointiin kuuluu mahdollisten riskien vaikutusten vakavuuden määrääminen. Arvioinnissa kysytään: Onko tämä riski vai ei? Miten vakavasta riskistä on kyse? Mitkä ovat seuraukset? Millä todennäköisyydellä riski tapahtuu? Riskin arviointiprosessin pohjalta päätetään jatkotoimista, jotka voivat olla joko päätös riskin vähentämisestä, hallitsemisesta tai sietämisestä (ei jatkotoimia). Arvioinnissa merkittäviä tunnistamisasioita ovat seuraavat:

- Mitä indikaatioita voidaan hyödyntää määrättäessä vikautumistodennäköisyyttä?
- Mitkä ovat tietyn toiminnan epäonnistumisen seuraukset?

Vikautumistodennäköisyyden määrittelyssä tunnistetaan toiminnon laadulle merkittävät seikat. Niitä ovat suunnittelun aikaisten muutosvaateiden määrä, ohjelman koko ja monimutkaisuus, ohjelmoijan taito, testiponnistelut jne. Seurausten merkittävyyttä on joskus mahdotonta määrittää täsmällisesti, mutta usein riittää kvalitatiivisuus: pieni, kohtalainen tai suuri. Yhdistämällä vikautumistodennäköisyys ja seurausten suuruus saadaan yksittäisten toimintojen riskin suuruus määrättyä.

Riskin vähentämiskeinot pohjautuvat riskien tunnistamisessa ja riskien merkityksen arvioinnissa kerättyyn informaatioon. Monista vaaran esiintymistodennäköisyyteen tai seurausten suuruuteen vaikuttavista vähentämistekijöistä tässä julkaisussa kiinnitetään

huomiota testausten merkitykseen. Tarkoituksena on riskianalyyseilla kohdentaa tunnistettujen kriittisten kohtien (esim. toiminnot tai komponentit) testaamista siten, että kohdan vaikutus tuotteessa vähimmäistyy.

3.6 Miten arvioida automaattisen testaamisen tehokkuutta?

Käsitellään lyhyesti mittoja riskien esiintymisen ja vähentämisen ilmaisijoina. Mittojen hyödyntämiseen on lukuisia syitä, tässä tapauksessa mm. seuraavat:

- automaattisen testauksen investointikustannusten kannattavuus
- mahdollisuuksien arviointi, vaihtoehtojen vertaaminen, parannusten valvonta
- ongelmista varoittaminen mahdollisimman aikaisin, ennustaminen.

Mitoista on kirjoitettu hyvin paljon (mm. Fenton & Pfleeger 1997). Jakotapojakin on useita, tässä ne jaetaan kahteen ryhmään, edistymistä ilmaiseviin ja virheen esiintymistodennäköisyydestä ilmaiseviin.

Edistymistä ilmaisevia mittoja ovat mm. seuraavat:

- suunniteltujen, suoritettujen ja valmiiden testitapausten lukumäärä
- virheiden toimintokohtainen lukumäärä
- käytettyjen testituntien määrä löydettyä virhettä kohti
- käytettyjen korjaustuntien määrä virhettä kohti (korjaaminen ja uudelleentestaaminen).

Edistymitoilla voidaan myös verrata automaattisen testaamisen edullisuutta manuaaliseen testaamiseen. Siinä kiinnostavaa on tietää, löydetäänkö automaattisilla testauksilla uusia merkittäviä virheitä ja mitkä ovat niistä aiheutuneiden seurausten mahdolliset kustannukset. Virheen esiintymistodennäköisyyttä ilmaisevat mitat poikkeavat huomattavasti edellisistä. Ilmaisimia ovat mm.

- toimintojen muutostiheys edellisen julkistamisen jälkeen
- funktion koko

- toiminnallinen tai rakenteellinen monimutkaisuus
- suunnitteludokumentaation laatu.

Jokaiselle käytetylle ilmaisimelle voidaan antaa luku 1, 2 tai 3 (pieni, kohtalainen tai suuri) toiminnoittain ja lisäksi ilmaisimia voidaan painottaa keskenään. Virheen esiintymistodennäköisyys voidaan nyt laskea jokaiselle käsitellylle järjestelmän toiminnolle sekä edelleen asettaa toiminnot todennäköisyystason mukaiseen tärkeysjärjestykseen.

$$Riski(f) = p(f)\frac{C(a) + C(t)}{2} \tag{1}$$

missä *p(f)* kuvaa testattavan toiminnon laatua, ts. ohjelman tai moduulin laatu, mikä johtaa matalaan tai korkeaan virheen esiintymistodennäköisyyteen,
C(a) kuvaa seurausten kustannuksia asiakkaan kannalta, ja
C(t) kuvaa seurausten kustannuksia toimittajan kannalta.

Toiminnon laatua ilmaisevia indikaattoreita ovat monimutkaisuus, ohjelmoijien kokeneisuus, suunnittelun taso jne. Asiakkaan kannalta kustannukset muodostuvat markkinaosuuden menetyksestä, oikeuskäsittelystä, viranomaismääräysten vastaisuudesta jne. Toimittajan kannalta kustannukset muodostuvat kielteisestä julkisuudesta, korkeista ylläpitokustannuksista jne.

4. Ohjelmiston virhemekanismit

Virhemekanismi on keskeinen tekijä ohjelmiston luotettavuustekniikassa. Ohjelmiston ja sen laitteiston virhemekanismin täsmällinen ymmärtäminen auttaa kehitettäessä oikean suuruisia luotettavuuden parannuskeinoja ja tukee täsmällistä luotettavuuden arviointia.

Luotettavuus on myös suhteellista – näkökulmasta riippuvaa. Luotettavuusattribuutteja voidaan painottaa hyödyntäjän näkökulmilla siten, että vain tietyt tai tietynlaiset poikkeamat halutuista arvoista katsotaan merkittäviksi. Onnettomuuksiksi johtavat tapahtumat ovat eittämättä tällaisia poikkeamia, kun taas kosmeettiset virheet eivät välttämättä ole. Luotettavuuden suhteellisuus johtaa siten myös virhemekanismin suhteellisuuteen. Virheet eivät ole samanarvoisia kaikkien hyödyntäjien silmissä.

Kohdassa 4.1 määritellään ohjelmiston luotettavuustekniikka, kohdassa 4.2 esitetään lyhyesti kaksi merkittävää ohjelmistovirheestä aiheutunutta onnettomuutta, joilla on ollut vaikutus luotettavuus- ja laatuprosessien kehittymiselle ohjelmistotuotannossa. Kohdassa 4.3 esitetään virheketjun osatekijöitä ohjelmistovirheitä, -virhetilanteita ja -virhetoimintoja. Tiedetään, että jotkut ohjelmistomoduulit tai -komponentit ovat paljon virhealttiimpia kuin toiset. Syitä selvitetään kohdassa 4.4.

Ohjelmistokomponentin virhemekanismien mallintamisessa tulee ottaa huomioon ohjelmistovirheen synty- ja etenemismekanismit, ihmisen tekemästä virheestä ohjelman tai osajärjestelmän sisäiseen virhetilanteeseen ja edelleen järjestelmätason virhetoimintaan tai vikautumiseen. Kohdassa 4.5 tarkastellaan erästä esimerkkiä järjestelmän luotettavuuden arvioimiseksi komponenttien virhetilannetietämyksen pohjalta. Esimerkki liittyy hajautettujen järjestelmien virhesietoisuuden luotettavuuteen. Virhesietoisuus rakennetaan lähtien tietyistä virhetilanneolettamuksista, joiden ollessa vääriä käyttötilanteessa järjestelmä vikautuu. Merkittävää myös on, että varmistusten lisääminen ei välttämättä kasvata luotettavuutta, vaan kuten kohdan 4.5 esimerkissä havaitaan, se voi myös vähentää luotettavuutta.

Virheen tai virhetilanteen syöttömenetelmät ovat olleet laajan tutkimuksen kohteena, mutta vähän käytettyjä. Kohdassa 4.6 esitetään eräs virheensyöttömenetelmä, jolla ohjelmiston luotettavuutta tietyin edellytyksin voidaan arvioida.

4.1 Ohjelmiston luotettavuustekniikka

Luotettavuus on järjestelmän keskeinen laatuun liittyvä ominaisuus, jolla perustellaan luottamusta järjestelmän tuottamiin palveluihin eli turvaudutaan järjestelmään. Palvelu

on se osa järjestelmän tuottamista tuloksista, jonka käyttäjä havaitsee. Usein luotettavuutta pidetään yleisterminä, jolla tarkoitetaan järjestelmän kaikkia luotettavuusattribuutteja: toimintavarmuutta, käyttövarmuutta, ylläpidettävyyttä ja turvallisuutta. Turvallisuudella tarkoitetaan myös eri asioita. Joskus sen merkitystä laajennetaan käsittämään lähes kaikki, mikä on tavalla tai toisella kriittistä, kuten henkilö-, ympäristö-, omaisuus-, tehtävä- ja käyttökeskeytysturvallisuus.

Musa (1998) määrittelee ohjelmiston toimintavarmuuden virheettömyytenä tietyissä olosuhteissa ja ympäristössä. Hänen mukaansa ohjelmisto on luotettava spesifikaatioihinsa verrattuna. Palveluun ei olla tyytyväisiä, jos toteutus ei vastaa määrittelyä, ja silloin jossakin kehitysvaiheessa on tehty virhe. Ohjelmiston virheettömyys yhtyy Musan määritelmässä ohjelmiston toimintavarmuuteen. Siinä ei ole kyse toimintojen oikeellisuudesta, ts. ei oteta kantaa siihen, onko ohjelmisto määritelty ja suunniteltu tarkoitusta vastaavasti.

Ohjelmiston luotettavuustekniikka koostuu luotettavuuden parantamisesta, attribuuteista ja virhemekanismeista. Parannukset luokitellaan virheitä välttäviin ja eliminoiviin, virhesietoisiin ja virheistä ennustaviin menetelmiin. Ne ovat tekniikoita, ratkaisuja ja työkaluja, joita tarvitaan luotettavan tuotteen suunnittelussa, tuottamisessa ja arvioimisessa. Luotettavuusattribuutit ovat mitattavia ominaisuuksia, joilla tuotteen tai palvelun laadun merkittävyyttä arvioidaan. Luotettavuusattribuutteja ovat erityisesti toimintavarmuus, käytettävyys, ylläpidettävyys ja turvallisuus. Aina kun puhutaan luotettavuudesta, tarkoitetaan tietyllä tavalla kriittistä toimintaa. Virhemekanismi muodostaa jatkuvan ketjun, joka alkaa ihmisen tekemästä virheestä tai ohjelmiston tai laitteiston komponenttitason vikautumisesta ja jatkuu osajärjestelmän ja järjestelmän vikautumisiin sekä edelleen järjestelmän ulkoisiin tapahtumiin.

4.2 Onnettomuus on monen yhteensattuman summa

Virheketjun alkaminen ihmisestä on ilmennyt jo lukemattomissa ohjelmoitavan tekniikan aiheuttamissa onnettomuuksissa. Lähes poikkeuksetta ohjelmistovirheet ovat suunnitteluvirheitä, jotka saavat alkunsa jossakin vaiheessa ohjelmiston kehitystyötä. Tyypillisempiä virhealttiita vaiheita ovat vaatimusmäärittelyt. Onnettomuustutkinnat ovat kuitenkin osoittaneet, ettei virheen eteneminen onnettomuudeksi johdu yksin ohjelmistovirheestä.

4.2.1 Therac 25 -onnettomuudet

Eräs onnettomuussarja, joka vaikutti henkilöturvallisuuden kehittymiseen, tunnetaan nimellä Therac 25 (Leveson 1995). 1980-luvun puolivälissä tapahtuneet lääkintälaitten ohjelmistovirheestä aiheutuneet onnettomuudet olivat ratkaiseva alkusysäys monille lääkintälaitevalmistusta koskeville sääntömuutoksille ja vaikutukset muidenkin alojen ohjelmistokehitykseen ovat olleet ilmeiset. Food and Drug Administrationin (FDA) mukaan (Wallace & Kuhn 2000) yksikään sen tietoon tullut ohjelmistovirhe ei ole ollut kriittinen sen toimialalla kuuluvissa lääkintälaitteessa vuoden 1987 jälkeen.

Therac 25 -onnettomuudet johtuivat ohjelmistovirheestä, mutta myös monen ulkoisen suojautumistekniikan epäonnistumisesta ja yhteistoimintahäiriöistä eri asianosaisten välillä. Lisäksi Leveson (1995) mainitsi, että kaikissa tapauksissa todellinen onnettomuuteen johtanut virhemekanismi ei ollut enää analysoitavissa, vaikka kaiken alkujuuri – ohjelmistovirhe – olikin selvästi osoitettavissa.

Therac 25 oli erään lääkintälaitevalmistajan sädehoitolaitteen uusin versio. Onnettomuuksiin johtavia tapahtumia tällä laitteella sattui kaikkiaan kuusi, jotka vaativat ainakin kuusi kuolonuhria. Onnettomuudet tapahtuivat vuosina 1985–1987, siis suhteellisen pitkällä aikajaksolla. Sädehoitolaitteen aikaisemmat versiot eivät olleet aiheuttaneet vastaavia vaaratilanteita, vaikka eräässä niistä (Therac 20) olikin ko. virheen sisältämä ohjelma. Laitteessa Therac 20 oli mekaaninen suoja, mikä oli estänyt tapahtuman kehittymisen. Therac 25:n suoja oli ohjelmallinen eikä se ollut estänyt tapahtuman kehittymistä. Ohjelmistovirhe oli tästä aikaisemmasta versiosta poistettu, mutta ei Therac 25:sta siitä syystä, että laatupäällikköllä oli tieto, ettei Therac 25:ssa ollut käytetty Therac 20:n ohjelmia.

Ohjelmistovirheenä voidaan Therac 25 -tapauksessa pitää sitä, että ohjelma ei osannut ottaa huomioon käyttäjän odottamattomia ohjaussyöttöjä. Ohjelma odotti, että käyttäjä syöttää manuaalisesti uudelleen kaikki syöteluvut pelkän palautuspainikkeen (return) sijasta niin kuin kaikissa onnettomuustapauksissa oli tehty.

Useissa tapauksissa sädehoitotilan valvontalaitteet, mm. kameravalvonta, olivat olleet epäkunnossa. Niillä laitteen hoitaja olisi kyennyt näkemään, että jokin ei ollut kunnossa hoitohuoneessa ja pysäyttänyt toiminnan. Muita valvontakeinoja kuin potilaiden reaktiot ylisäteilytykselle ei ollut käytössä. Riippumattomat tarkistukset laitteen ja sen ohjelmiston oikeasta toimimisesta puuttuivat.

Hoitaja oli tehnyt syöttövirheen, joita sattui useita päivän aikana. Niistä tuli virheilmoituksia, mutta ohjeet ilmoituksista olivat olleet puutteelliset. Tietty virhekoodi ohjekirjassa sisälsi kaksi virhemahdollisuutta: joko liian pieni säteilymäärä tai liian suuri sä-

teilymäärä. Useimmiten kyseessä oli turha virheilmoitus tai ilmoitus liian pienestä säteilymäärästä. Hoitajat olivat yleensä käynnistäneet laitteet uudelleen muutettuaan syöttöarvoja, ja hoitotoimet olivat sujuneet ongelmitta.

Laitteelle oli tehty myös riskianalyysit. Ensimmäisissä analyyseissa ei ollut mukana ohjelmisto-osuutta ja seuraavassa oletettiin kaikille ohjelmistovirheille yhtä suuri todennäköisyys. Analyysin tulokset viittasivat suurimpaan riskiin kytkimen kohdalla, ja valmistaja paransi kytkimen turvallista toimintaa. Valmistaja olikin sitä mieltä, että myöhemmät onnettomuudet eivät voineet johtua heidän laitteestaan, sillä todennäköisyyspohjaiset analyysit osoittivat turvallisuuden parantuneen kuusi dekadia kytkimen uudelleensuunnittelun johdosta. Kuuden dekadin kasvu turvallisuuden eheydessä on kuitenkin aika vaikea perustella. Lukuun on otettu mukaan useita suojauskeinoja, ja laskelmissa on erehdytty pitämään niitä riippumattomina.

Therac 25 -onnettomuuden opetukset vaikuttavat ohjelmistoturvallisuuden alalla työskentelyyn yhä edelleen. Virheen eteneminen on estettävissä usealla tavalla (Leveson 1995):

1. Älä luota liikaa ohjelmistoon. Yhä edelleenkin kuulee sanottavan, että ”ohjelmisto ei voi vikautua”, mikä johtaa liialliseen tyytyväisyyden ja turvallisuuden tunteeseen ohjelmiston toimintoja kohtaan. Ohjelmistoversion moitteeton toiminta yhdessä suoritusympäristössä ei välttämättä merkitse moitteetonta toimintaa jossakin muussa ympäristössä. Vaikka laitteistoviat ovatkin yleisin ohjelmiston sisältävän järjestelmän vioista, tulisi myös epäillä ohjelmistovirhettä.

2. Tarkista turvallisuus, toimintavarma ohjelmisto ei riitä. Ohjelmisto voi toimia toimintavarmasti pitkän ajan, mutta turvallisesti sen tulisi toimia myös silloin, kun toimintavarmuus pettää.

3. Suojaa ohjelmisto. Suunnittelussa tulisi ottaa huomioon itsetarkastukset tai muut virhetilanteiden paljastus- ja käsittelykeinot.

4. Eliminoi perussyyt. Turvallisuuskriittisissä asioissa helposti kohdistetaan tarkastelu pelkästään tiettyihin ohjelmistovirheisiin. Riskitekijöiden poistaminen ei kuitenkaan ole riittävää, sillä aina on muita ohjelmistovirheitä. Usein myös yhden virheen poistaminen saattaa synnyttää uuden virheen.

5. Älä tuudittaudu tyytyväisyyden tunteeseen. Usein tarvitaan onnettomuus ennen kuin huomataan teknologian synnyttämät vaarat.

6. Arvioi riskit realistisesti. Todennäköisyyspohjaisissa riskitarkasteluissa tulisi realistisesti nähdä, miten riippumattomia ohjelmistovirheet ovat suojaavissa tai muissa redundanttisissa osuuksissa.

7. Seuraa onnettomuusraportteja. Jokaisella turvallisuuskriittisiä järjestelmiä valmistavalla yrityksellä tulisi toimintaohjeet, joiden pohjalta seurata onnettomuusraportointeja ja ylipäätään pieniäkin vihjeitä virhetilanteisiin, jotka voivat johtaa onnettomuuksiin.

8. Pidä ohjelmistotekniikka asianmukaisena. Seuraavat ohjelmistotekniikan periaatteet kannattaa pitää mielessä turvallisuuskriittisiä ohjelmistoja tuotettaessa:

   – Ohjelmistomäärittelyä ja -dokumentointia ei saisi jättää myöhempiin projektivaiheisiin.

   – Täsmälliset ohjelmiston laadunvarmistusmenettelyt ja -standardit tulisi laatia.

   – Suunnittelut olisi pidettävä yksinkertaisina ja arvelluttavia koodaustapoja vältettävä.

   – Virhetilanteiden paljastamiskeinot tulisi alusta lähtien suunnitella ohjelmistoon.

   – Ohjelmisto tulisi testata perusteellisesti ja analysoida formaalein menetelmin sekä moduuli- että ohjelmistojärjestelmätasolla. Regressiotestit tulisi suorittaa aina muuttamisen jälkeen.

   – Tietokonenäytöt ja käyttäjille kohdistettu informaatio (mm. virheilmoitukset ja käyttömanuaalit) tulisi huolellisesti suunnitella.

9. Ole varovainen ohjelmiston uudelleenkäytössä. Kaupalliset valmisohjelmistot eivät välttämättä ole turvallisia, vaikka niistä olisikin laajat käyttökokemustiedot. Toimintavarmuus ei vastaa turvallisuutta, eikä komponentin luotettavuustiedot sellaisenaan riitä. Järjestelmäksi kootut yksittäiset komponentit eivät ole riippumattomia, mitä yleensä luotettavuusmallinnuksissa edellytetään.

10. Varo suunnittelemasta käyttöliittymiä käyttäjäystävällisiksi turvallisuuden kustannuksella. Tulisi erityisesti ottaa huomioon, että käyttäjä ei toimi juuri suunnittelijan haluamalla tavalla. Hän ei esimerkiksi aina huolellisesti tarkista syöttämiään tietoja.

4.2.2 Ariane 501 -nousun epäonnistuminen

Toinen tapahtunut onnettomuus kuvaa myös, miten pienistä tekijöistä virhemekanismin etenemisessä ohjelmistojen osalta saattaa olla kyse. Seuraukset eivät useimmiten aiheudu yhdestä tekijästä vaan joukosta epäonnisia ratkaisuja. Euroopan avaruusjärjestön ESA:n kantoraketti Ariane 501 tuhoutui noin puoli minuuttia laukaisun jälkeen 4. kesäkuuta 1996. Raketti oli odottamatta muuttanut suuntaa, mikä ilmeisesti käynnisti rakettijärjestelmän automaattiset tuhoamispanokset, ja kun se havaittiin, lennonjohto räjäytti raketin kokonaan. Onnettomuuden tutkimuslautakunta asetettiin nopeasti onnettomuuden jälkeen ja sen julkaisema virallinen raportti (ESA 1996) on ollut yleisesti saatavilla.

Tutkimuslautakunta analysoi kaikki nousun aikana nauhoitetut vikautumista ilmaisevat mittaustiedot ja oli tutkinut suuren joukon kantoraketin teknisiä ja laadullisia dokumentteja.

Ariane 501:n vikautuminen aiheutui raportin mukaan täydellisestä ohjaus- ja asentoinformaation menetyksestä 37 sekuntia pääkoneiden sytyssekvenssin alkamisesta. Informaation menetys aiheutui määrittely- ja suunnitteluvirheistä inertiaalireferenssijärjestelmässä (SRI). Laajat Ariane 5 -kehitysohjelmassa toteutetut katselmukset ja testit eivät raportin mukaan sisältäneet riittävästi SRI:n tai lennonohjausjärjestelmän analysoimista ja testaamista.

Ongelman kehittyminen lähti liikkeelle täysin vaarattomalta vaikuttavassa ohjelmistokomponentissa, jossa raketin vaakasuoraa nopeutta kuvaava 64 bitin liukuluku muutettiin 16 bitin kokonaisluvuksi. Tässä tapauksessa lukuarvo oli kuitenkin yli 32 768, joka on suurin 16-bittisessä kokonaislukuaritmetiikassa esitettävissä oleva luku. Muunnos johti ylivuotoon, jota varten oli poikkeustilan käsittelymenettelyt. Prosessori antoi virhetilanteesta ilmoituksen ja tulosti virheraportin. Ohjelmisto oli koodattu Ada-kielisellä ohjelmalla, joka erityisesti tunnetaan turvallisuuskielenä.

Virhetilanteen käsittelyä ei kuitenkaan määritelty Ada-koodissa, jolloin ohjausjärjestelmä yritti tulkita tuloksen raketin ohjauskomennoiksi. Ohjauskomennot eivät pelkästään koskeneet vaaratonta komponenttia, vaan myös kriittisiä, mistä seurauksena oli holtiton käyttäytyminen ja lopulta raketin itsetuhomekanismin käynnistyminen. Kyseinen osa ohjauskoodia ei ollut tarpeen Ariane 5:ssä ja oli joka tapauksessa ohjelmoitu poistumaan käytöstä 40 sekuntia laukaisun jälkeen.

Kolmen ja puolen miljardin markan tappiot aiheuttaneen onnettomuuden syyksi paljastui seuraavan kaltainen ohjelmanpätkä:

```
short y;
float x;
...
y = convert(x);
```

Järjestelmä koostui kahdesta erilaisuusperiaatteella toimivasta kanavasta, mutta ohjelmisto oli kummassakin identtinen. Kummatkin kanavat aiheuttivat lähes samanaikaisesti järjestelmän alasajon.

Luotettavuuden parantamiseksi laitetasolla on huomattavasti varmistuksia. Kaksi SRI:tä, joilla on identtiset ohjelmistot, toimivat rinnakkain. Toinen SRI:stä on aktiivinen ja toinen kuumavarmennettu siten, että se kytkeytyy välittömästi päälle, kun kahdennettu OBC (On-Board Computer) havaitsee aktiivisen laitteen vikautuneen edellyttäen, että kytkettävä laite on kunnossa. OBS yritti kytkeä toiminnassa olleen SRI 2:n tilalle SRI 1:tä, mutta tämä oli lopettanut toimintansa viimeisen 72 ms -syklin aikana. Syy oli sama kuin SRI 2:lla. Ariane 5:n SRI on käytännöllisesti sama laite kuin vanhemmassa versiossa Ariane 4:ssä erityisesti ohjelmiston osalta.

Tutkintalautakunta antoi myös lukuisia suosituksia, joista osa voidaan yleistää ja kohdistaa useaan muuhun toimialaan:

1. Tärkeiden tehtävätoimintojen aikana tulisi välttää suorittamasta sellaisia ohjelmistotoimintoja, joita ei välttämättä tarvita.

2. Testaukset tulisi hoitaa niin reaalisilla laitteilla kuin on teknisesti mahdollista, mm. syötettävä realistista syöttötietoa sekä suoritettava täydelliset, suljetun piirin järjestelmätestaukset. Ennen tehtävään ryhtymistä on järjestelmä simuloitava täydellisesti. Testikattavuuden on oltava korkea.

3. Antureiden ei saa sallia lopettaa laadullisesti parhaan datan lähettämistä.

4. Ohjelmiston kelpoistusta varten tulisi järjestää erityiset katselmukset. Näihin ohjelmiston kelpoistuskatselmuksiin kuuluisi myös osallistua kokoonpanosta vastaavan henkilön (The Industrial Architect). Hänen tehtäviinsä kuuluisi tiedottaa järjestelmätestauksista, joita laitteelle on suoritettu. Kaikki laitetta koskevat rajoitukset tulee eksplisiittisesti tiedottaa katselmustyöryhmälle. Kaikki kriittinen ohjelmisto tulisi merkitä kokoonpanovalvontaan (Configuration Controlled Item).

5. Ohjelmiston kelpoistuskatselmoinneissa tulisi erityisesti kiinnittää huomiota seuraaviin seikkoihin:

   – Tunnistettava erityisesti koodidokumentaatiosta kaikki implisiittiset suureiden arvoja koskevat olettamukset.

   – Todennettava kaikki sisäisiä muuttujia ja tietoliikennemuuttujia koskevat arvoalueet.

– Kiinnitettävä huomiota kaikkiin potentiaalisiin ajotietokoneen (On Board Computer) ohjelmisto-ongelmiin, joita projektiryhmä on esittänyt.

6. Poikkeuskäsittelyitä tulisi mahdollisuuksien mukaan rajoittaa.

7. Kriittisiksi luokiteltavien komponenttien määrittelyssä otettava huomioon ohjelmistoperäiset, erityisesti yksittäisviasta aiheutuneet vikautumiset.

8. Määrittelyn, koodin ja perusteludokumenttien katselmuksiin tulisi osallistua projektista ulkoinen osapuoli. Näiden katselmointien tulee ehdottomasti koskea sisältöä, ei vain sen toteamista, että verifioinnit on tehty.

9. Testauskattavuudet on katselmoitava.

10. Perusteluihin tulisi kiinnittää yhtä lailla huomiota kuin koodiin. Tulisi parantaa koodin ja perusteludokumenttien yhdenmukaistavaa tekniikkaa.

11. Tulisi asettaa erityinen ryhmä, joka laatii toimintaohjeet ohjelmiston kelpoistamiselle, ehdottaa tiukkoja sääntöjä kelpoistuksen vahvistamiseksi ja hankkii varmuuden siitä, että ohjelmiston määrittely, verifiointi ja testaus ovat korkeatasoisia. Ulkoisia luotettavuusasiantuntijoita tulisi harkita tähän ryhmään.

12. Tulisi harkita mahdollisimman avointa yhteistyökumppanuutta kaikkien osapuolten kesken.

Ariane 5:n onnettomuus on merkittävyytensä (mittavat taloudelliset menetykset) vuoksi ollut jatkuvan spekulaation alaisena sekä kirjallisuudessa että alan ammattilaisten keskustelupalstoilla.

Merkittävää on, että jos Adan tilalla olisi ollut Fortran, C tai C++, muuntaminen olisi aiheuttanut rauhallisen ylivuodon, minkä seurauksena vaaraton komponentti olisi toiminut virheellisesti, mutta kriittiset oikein. Ohjelmoijat eivät olleet ymmärtäneet kaikkia poikkeuskäsittelyn vaikutuksia valitessaan kaikkiaan seitsemästä poikkeustilanteesta taloudellisista syistä vain kolme jatkokäsittelyyn. Integerimuunnoksen poikkeuskäsittely ei kuulunut valittuihin. Päätös oli normaalia kompromissien tekoa, joita on aina insinöörityössä tehtävä. Päätöksentekijöillä vain ei ollut kaikkea tietoa saatavilla. Toinen ohjelmointityötä koskeva puute on erilaisuusperiaatteen soveltaminen vain laitteistolle, mutta ei ohjelmistolle.

Monet tutkimukset ovat osoittaneet, että järjestelmän luotettavuudella ja ohjelmointikielellä ei ole selvää keskinäistä korrelaatiota. Kaksi eri sovellusohjelmistoa on voitu

kirjoittaa samalla ohjelmointikielellä, mutta sovellusten luotettavuus voi olla täysin erilainen. Esimerkiksi tekstinkäsittelyohjelmassa, jossa on automaattinen tallennus aina kymmenen minuutin välein, ei aiheuta merkittävää ajanhukkaa. Toisessa tapauksessa, lääkeaineiden annostelussa, taloudelliset ja suorituskyvylliset menetykset voivat olla myös merkityksettömät, mutta virhetoiminto vai aiheuttaa vaaran terveydelle. Kummassakin tapauksessa ohjelmointikieli voi hyvin olla sama.

Ariane 5:n tapauksessa voidaan epäillä virheellisesti toteutettuja tai puutteellisia kehitysprosesseja, verifiointeja ja validointeja, kuten tutkimuslautakunnan virallisesta raportista esitettyjen useiden suositusten perusteella ilmeneekin. Lukuisista suosituksista huolimatta raportti ei kuitenkaan ilmaise selkeitä puutteita tai ongelmia dokumentoinnissa, validoinnissa tai johtamisessa. Testatakaan ei aina voida kaikkea, vaan valintoja on pakko tehdä.

Perussyy oli määrittelyn uudelleenkäytössä. SRI:n vaakasuoraa nopeutta kuvaava moduuli oli otettu Ariane 4:stä, eikä se ollut sopinut Ariane 5:lle. Jälkeenpäin on helppo spekuloida, mutta Jézéquel & Meyerin (1997) mielestä muunnos olisi pitänyt tehdä eksplisiittisesti, esimerkiksi Eiffel-kielellä seuraavasti:

```
convert (horizontal_bias: INTEGER): INTEGER is

        require
                horizontal_bias <= Maximum_bias
        do
                ...
        ensure
                ...
        end
```

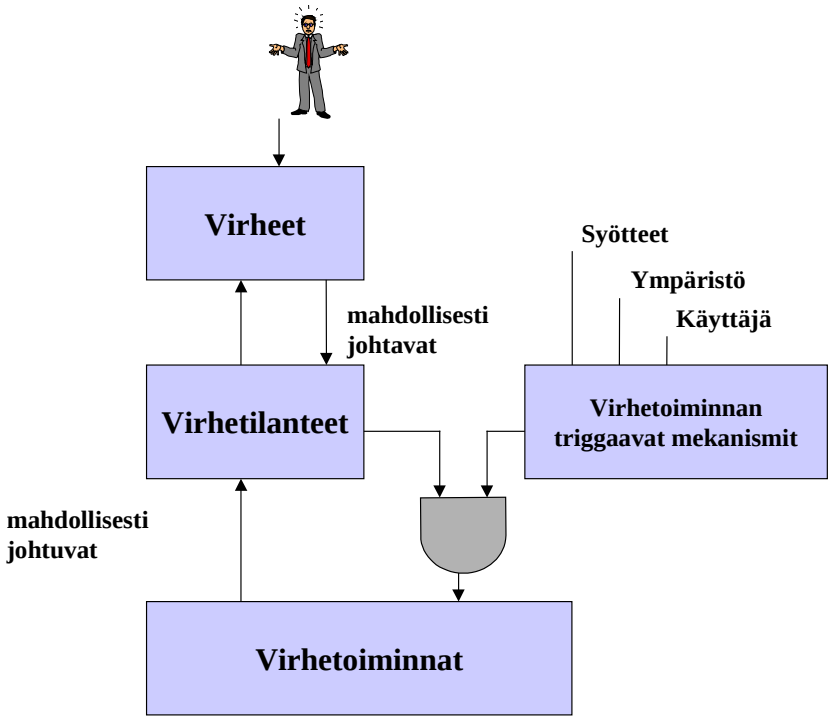
missä ennakkoehto selkeästi ja täsmällisesti määrää tarkistukset, jotka syöte on täyttävä ollakseen hyväksyttävä.

Tietokoneohjelmien virheellinen käyttäytyminen on kaikkialla jatkuvan kiinnostuksen kohteena. Pohjimmiltaan halutaan tietää, miten testaamalla tai analysoimalla kyettäisiin paljastamaan ohjelmistovirheitä tai mitkä ovat sellaiset ohjelmisto-ominaisuudet, jotka piilottavat virheet sekä testaajilta että analysoijilta. Ohjelmiston virheellisen käyttäytymisen selvittäviä menetelmiä onkin kehitetty vuosien saatossa. Niistä mainittakoon tässä yhteydessä mutaatioanalyysit (DeMillo et al. 1978), vikakeskeiset testit (Morell 1988), herkkyysanalyysit (Voas 1992) ja ohjelmiston turvallisuusanalyysit, joista viime mainittuja esitetään kattavasti kirjassa (Leveson 1995).

4.3 Virhemekanismin käsitteet

Ohjelmiston luotettavuuden heikennyksiä kuvaavat virhekäsitteet virhe (engl. fault), virhetilanne (engl. error) ja virhetoiminta (engl. failure). Virhekäsitteet eivät ole vakiintuneita niin suomenkielisesti kuin englanninkielisestikään. Vikatermiä käytetään yleensä yleismerkityksessä tarkoittamaan lähes kaikkia vikakäsitteitä, virheellä taas viitataan ihmisen tekemään erehdykseen. Viime mainitusta syystä tässä julkaisussa käytetään ohjelmistovioista johdonmukaisesti virhe-termiä.

Virhetoiminta on ohjelman ulkoisesti havaittavan palvelun poikkeama ohjelman spesifikaation mukaisesta toiminnasta eli siitä, mitä järjestelmän pitäisi tehdä. Virhetilanne on se osa järjestelmän tilaa, joka voi mahdollisesti johtaa virhetoimintaan, ja virhe on virhetilanteen todellinen, havaittu tai oletettava syy (kuva 11).



Kuva 11. Virhemekanismi. Ohjelmistovirheet ovat aina suunnitteluvirheitä, jotka jäävät piileviksi virheiksi tai virhetilanteiksi, ennen kuin jokin ulkoinen ilmiö, esimerkiksi testaus, tuo ne esiin.

Virhemekanismissa on kyse virhetapahtuman syysuhteesta eli tapahtumavirrasta. Ohjelmistoille syysuhde on kolmidimensioinen käsittäen prosessin, ohjelmiston ja laitteis-

ton. Virheet syntyvät ja etenevät missä tahansa elinkaaren vaiheessa, ohjelmisto- ja laitteistohierarkiassa.

Korkean luotettavuus- tai turvallisuustason saavuttamiseksi on ensiarvoisen tärkeätä kyetä vähimmäistämään virhetilanteisiin ja edelleen virhetoimintoihin johtavien virheiden lukumäärää. Vähimmäistämiseen on valittava suunnitteluratkaisut useista vaihtoehdoista, jotka on luokitettavissa mm. termein defense in depth, redundancy, diversity and robustness.

Virhemekanismin käsitteet ovat täysin olennaisia yritettäessä purkaa tapahtumasuma, ennen kuin se muodostuu piilevästä näkyväksi ongelmaksi. Käsitellään syysuhdetta vastavirtaan, niin kuin luotettavuustarkasteluissa tehdään, eli ensiksi tunnistetaan mahdolliset ongelmat.

4.3.1 Virhetoiminnat

Virhetoiminnan seurauksena ohjelman on mahdotonta suorittaa tehtäväänsä, mikä merkitsee myös sitä, että järjestelmällä ei ole virhetoimintaa, jos se toimii spesifikaatioidensa mukaisesti, vaikka jotakin poikkeavaa tapahtuisikin. Tällaisissa tapauksissa kyseessä on virhe spesifikaatiossa.

Eri virhetoiminnoilla on usein erilaisia vaikutuksia järjestelmän toimintaan. Vaikutukset voivat ilmaantua äkillisesti ja odottamatta tai hitaasti siten, että järjestelmän toiminta heikkenee vähitellen. Tällöin vain osa toiminnoista on ehkä menetetty, tärkeimmät tehtävät saatu ”hajautettu hallinta” -periaatteen mukaisesti.

Onnettomuuteen johtavat virhetoiminnot saavat runsaasti palstatilaa niin mediassa kuin alan tieteellisissä julkaisuissakin. Jälkeenpäin virhemekamismin tunnistaminen on kovin helppoa, ja mediakirjoitusten mukaan virheetkin olisivat olleet vältettävissä noudattamalla jo hyviksi koettuja menetelmiä. Otetaan vaikka kaksi esimerkkiä tunnetuista onnettomuuksista, Ariane 5 ja Therac 25, joiden yhtenä syynä oli liiallinen luottamus uudelleen käytettäviin ohjelmistokomponentteihin.

Komponenttien uudelleenkäyttö on keskeistä valmisohjelmistoteollisuudessa, mitä pidetään tulevaisuudenkuvana ohjelmistovalmistuksessa. Hyödyt näkyisivät mm. parantuneena luotettavuutena. Luotettavuus keskimäärin varmasti paraneekin, mutta integroituvuus aiheuttaa ongelmia, joita ei suunnitteluvaiheessa ole kyetty ottamaan huomioon. Käytäntö on tuonut selvästi esille, miten pienikin ohjelmistovirhe voi aiheuttaa mittavia taloudellisia menetyksiä tai ihmisuhreja, kuten Ariane 5 ja Therac 25 -tapauksissa.

Ariane 5 -onnettomuus tapahtui 4.7.1996, kun laukaisuraketti joutui väärälle liikeradalle ja se jouduttiin räjäyttämään sekunnin kuluttua laukaisusta.

Koska järjestelmät vioittuvat eri tavoin, on johdettu käsite vioittumistapa. Vioittumistapa on keskeinen useimmissa kvalitatiivisissa luotettavuustarkasteluissa. Se on ohjelmiston virhemallissa läheisessä suhteessa virhetoiminnan käsitteeseen. Siten voidaankin puhua virhetoimintatyypeistä, joille kehitetään luotettavuustarkasteluita varten kuvaavia avainlauseita (Harju 2000).

Virhetoiminnan luokittelun tavoitteena on antaa tukea arvioitaessa suojaavia ja muita vastaavia toimenpiteitä, joilla tietyn virheen eteneminen virhetoiminnoksi kyetään vastaisuudessa estämään. Minimiluokittelu sisältää ainakin seuraavat tiedot:

- vioittumistapa, joka on järjestelmätasolla havaittu tapahtuma
- tapahtuman ajankohtatiedot ja käynnissä ollut operaatio
- virhetoiminnon käynnistävät tekijät, kestoaika ja laajuus.

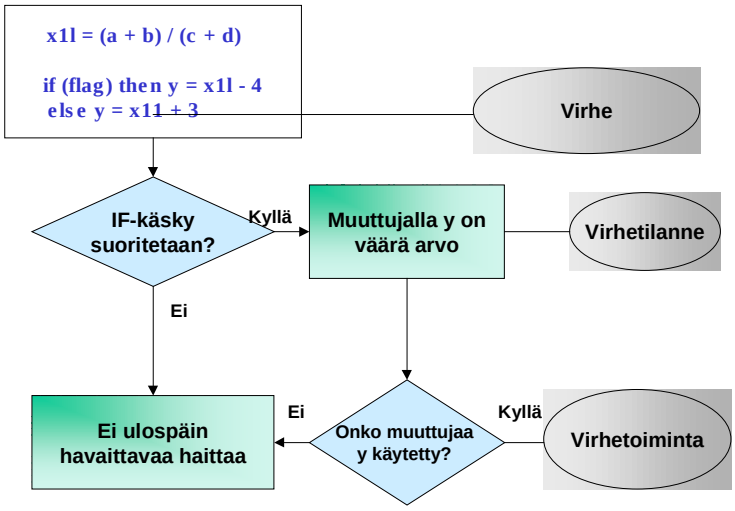
Lisäksi saatavuudesta riippuen tulisi kuvata virhetoiminnan kriittisyys, vaikuttavat komponentit, havainto-, toipumis- ja suojausmekanismit. Kriittisyys määritellään ohjelmiston vaikutuskohteen merkittävyyden mukaan, vaikuttavat komponentit ovat ne jotka ovat välittömästi olleet virhetoiminnan vaikutuksen alaisina, havaintomekanismit ovat ne tavat, joilla virhetoiminta ensimmäiseksi paljastui. Toipumismekanismeja ovat mm. virheen peittäminen, automaattinen kytkentä toiselle komponentille, automaattinen uudelleenkäynnistys, operaattorin toimenpiteet, alasajo jne.

Virheen ja virhetoiminnan sattuessa tiedonkeruun tärkeimpiä osuuksia on ympäristön kuvaaminen. Siihen kuuluvia osuuksia ovat ainakin seuraavat tiedot: paikan tunniste, toiminto, tietokonejärjestelmä, ohjelmointikieli, lähdekielisen koodin koko, kaupallisen tai uudelleenkäytetyn koodin osuus, kehitys- ja testimenetelmä, riippumaton verifiointi- ja validointi, suunnittelun kestoaika sekä järjestelmätyypin kokonaiskäyttöaika hyväksyntätesteistä eteenpäin.

4.3.2 Virhetilanteet

Virhetilanne on se osa järjestelmän tilaa, joka voi tietyillä ehdoilla johtaa virhetoimintaan. Erilaisilla virhesietoisilla varmistuksilla joko estetään tietyn virhetilanteen eteneminen tai konstruoidaan yleinen varmistusmekanismi kaikenlaisille mahdollisille virhetilanteille. Kuva 12 esittää yksinkertaista esimerkkiä syntaktisen virheen etenemisestä näkyväksi virhetoiminnaksi. Vain jos muuttujaa *y* tullaan käyttämään esimerkiksi suo-

rituksessa toiminnallisten testausten yhteydessä, voidaan havaita tietyn tuloksen poikkeavan halutusta tuloksesta ja lähteä jäljittämään virhetilannetta ja poistaa sen syy.



Kuva 12. Syntaktisen virheen siirtyminen virhetoiminnoksi.

Kuvan tapauksessa virhetoiminta löytyy, kun ohjelma osataan suorittaa tietyillä syötteiden arvoilla ja mahdollisesti vielä tiettynä ajanhetkenä. Virhetilanne on kuvan esittämää tapausta paljon monimutkaisempi silloin, kun kyse on semanttisesta syysuhteesta tai ilkivaltaisesta ohjelmoinnista. Semanttisessa tapauksessa koodinosan merkitys on saattanut muuttua esimerkiksi virheellisen korjauksen jälkeen niin, että vaikka tiedetäänkin virhetoiminnon läsnäolo, sitä ei osata jäljittää oikeaan kohtaan, vaan mahdollisesti tehdään virheellinen muutos toiseen osaan ohjelmaa.

Ilkivaltaisessa ohjelmoinnissa on tahallisesti koodattu ohjelmanosia ja niitä herättäviä syötemahdollisuuksia. Näitä ovat salaovet ja takaportit, joita ei spesifioida vaatimuksiksi näkyvästi vaan peitellysti siten, että testaaminen vaikeutuu.

Virhetilanteet luokitellaan Christmansson & Chilleregen (1996) mukaisesti neljään pääryhmään:

1. Yksinkertainen osoitevirhetilanne, kun virheestä seuraa väärä osoite. Luokkaa voidaan vielä täsmentää: control block address, storage pointer, module address, linking of data structures and register.

2. Muut yksinkertaiset virhetilanteet ovat virheellisiä arvoja, parametreja, lippuja, sanan pituuksia, lukituksia, indeksejä ja nimiä.

3. Moninkertaiset virhetilanteet ovat samasta syystä samanaikaisesti syntyneitä virhetilanteita.

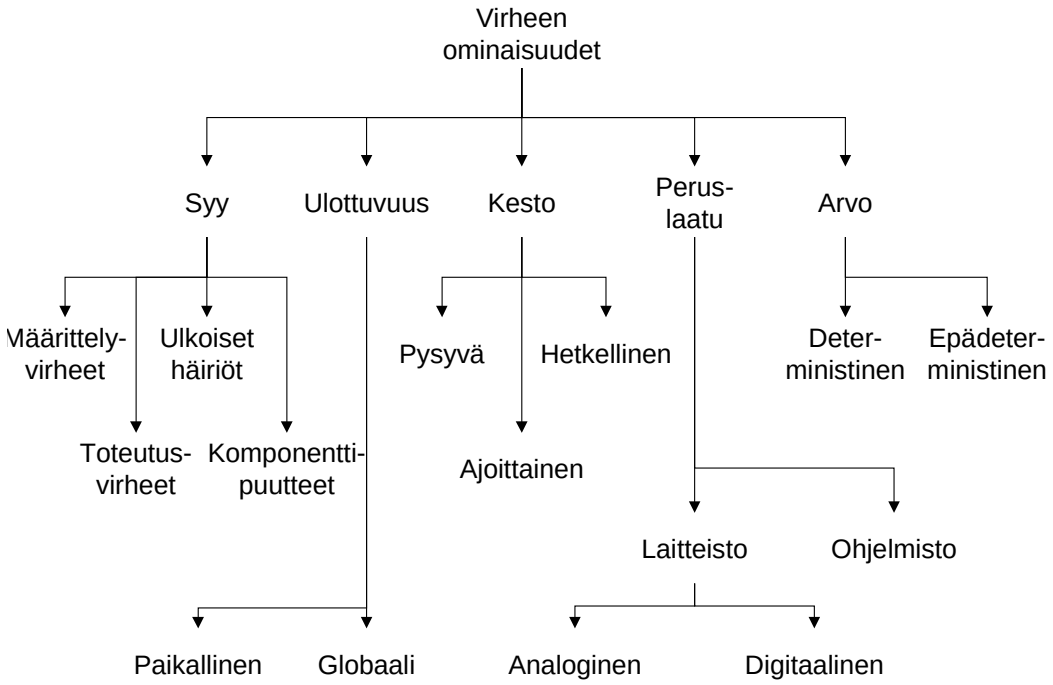
4. Ohjausvirheet esiintyvät muistiin tallennetuissa tiedoissa hyvin monivivahteisesti ja epädeterministisesti. Itse asiassa syntynyttä virhetilannetta on vaikea tunnistaa tai luokitella mihinkään edeltävään kolmeen luokkaan.

4.3.3 Virheet

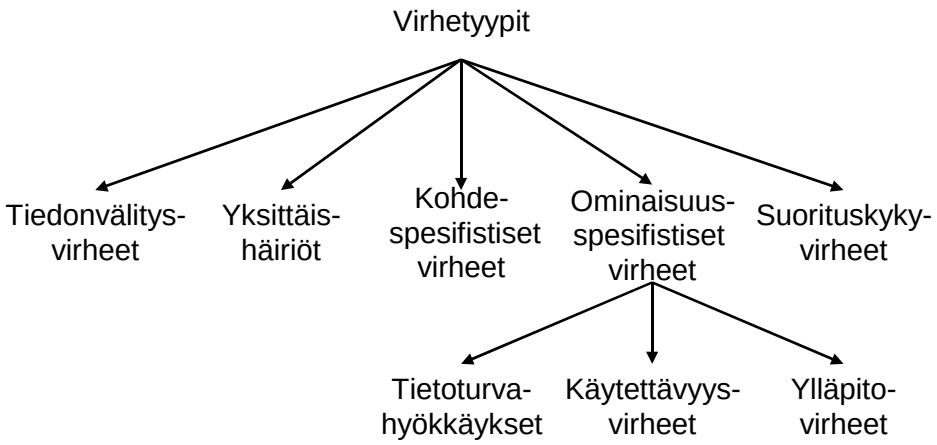
Virhe on virhetilanteen havaittu tai oletettu syy. Kaikki ohjelmiston virhetoiminnot johtuvat virheistä. Virhe voi olla fyysinen tai inhimillinen, tahallinen tai tapaturmainen. Se voi syntyä missä prosessointivaiheessa tahansa, jolloin on vastaavasti kyse määrittely-, suunnittelu- tai käyttövirheistä. Virhe voi olla ohjelmiston sisäinen tai ulkoinen laitteisto-, järjestelmä- tai ympäristövirhe. Virhe on aina jossakin muodossa syntaktisena tai semanttisena tarkasteltavassa ohjelmassa tai sen laitteistossa, mutta se on prosessin tuote.

Luotettavuustarkasteluiden tarkkuus riippuu täysin siitä, miten tarkasti virheiden luonne pystytään ymmärtämään. Nelson & Caroll (1982) kuvasi viisi kriittistä ominaisuutta, jotka virheestä olisi tunnettava: syy, kesto, ulottuvuus, arvo ja peruslaatu (kuva 13). Virheen syy jakautuu neljään ensisijaiseen tyyppiin, virheen kesto kuvaa aikaa, jona virhe on aktiivinen. Hetkelliset virhetyypit, jotka ilmestyvät lyhyeksi ajaksi ja häviävät, voidaan jakaa osatekijöihin kuvan 14 mukaisesti. Ajoittaiset virheet ilmestyvät ja häviävät toistuvasti. Virheen ulottuvuus jaetaan kahteen osaan: ulottuvuus on paikallista tai sen vaikutus on globaalista. Determinanttinen virheen arvo tarkoittaa muuttumatonta arvoa tietyssä ajassa ja epädeterminanttinen vastaavasti muuttuvaa arvoa.

Suurin osa ohjelmiston virhetoiminnoista on jäljitettävissä vaatimuksiin ja määrittelyihin, mikä on luontaista ohjelmistoille. Suunnittelija voi ymmärtää väärin käyttäjän toiveet, mikä saattaa johtaa virheellisiin teknisiin tulkintoihin määrittelyissä ja edelleen johtaa virheelliseen toteutukseen. Väärinkäsitykset asiakkaan ja suunnittelijan välillä ovat keskeisimpiä virhelähteitä, mistä syystä tulisikin paneutua erityisesti kohdealuetietämykseen ja vaatimusten määrittelytekniikkaan.



Kuva 13. Ohjelmistovirheen perusominaisuudet (Nelson & Caroll 1982).



Kuva 14. Transienttivirheiden taksonomia.

Virheet ovat peräisin myös suunnittelu- ja toteutusvaiheista, joissa vaatimukset on mahdollisesti tulkittu väärin. Tulkkina voi olla suunnittelija tai työkalu, esimerkiksi ohjelmakääntäjä. Ongelmaratkaisu voi jäädä vajaaksi, johtuen ehkä suunnittelijan keskittymisen hetkellisestä herpaantumisesta. Tyypillisiä toteutusvirheitä ovat ohjelman syntaktiset virheet.

Kaikki ohjelmistovirheet ovat ihmisen tekemiä, mikä johtuu yksinkertaisesti siitä, että ihminen joko välittömästi tai välillisesti on rakentanut ja ohjannut järjestelmiä. Virheet esiintyvät aina ohjelmassa, ohjelma on aina laitteistossa. Laitteistoviat voivat aiheuttaa ohjelmistotoiminnon epäonnistumisen. Mahdollisena virhetoiminnon syynä on lähes aina laitteistovika, jota vastaan on kuitenkin olemassa suhteellisen monipuolinen valikoima vikasietoisia menettelytapoja.

Virhetietojen keruun pääasiallisena tavoitteena on tukea ohjelmistovirheiden vähentämistä tulevissa projektissa. Luokittelun avulla kyetään kohdentamaan resursseja virhealttiisiin kohtiin ja virheen paljastusprosesseihin. Minimiluokittelu kattaa ainakin seuraavat kohdat, joista tiedot kohtiin neljännestä eteenpäin voivat olla vaikeasti saatavilla:

1. Virheiden alkuperä, joka voi ylimmällä tasolla olla laitteisto, ohjelmisto, käyttöliittymä tai ympäristö. Laitteiston ja ohjelmiston alimmat virheen alkuperän tasot ovat komponentit, käyttöliittymien kohdalla työn luonne ja vastuuhenkilöt sekä ympäristön kohdalla erilaiset energian syötöt (sähkö, lämmitys), tulipalot jne.

2. Havaitsemistavat, joita ovat tarkistukset, katselmukset, testit, käyttö ja hälytykset.

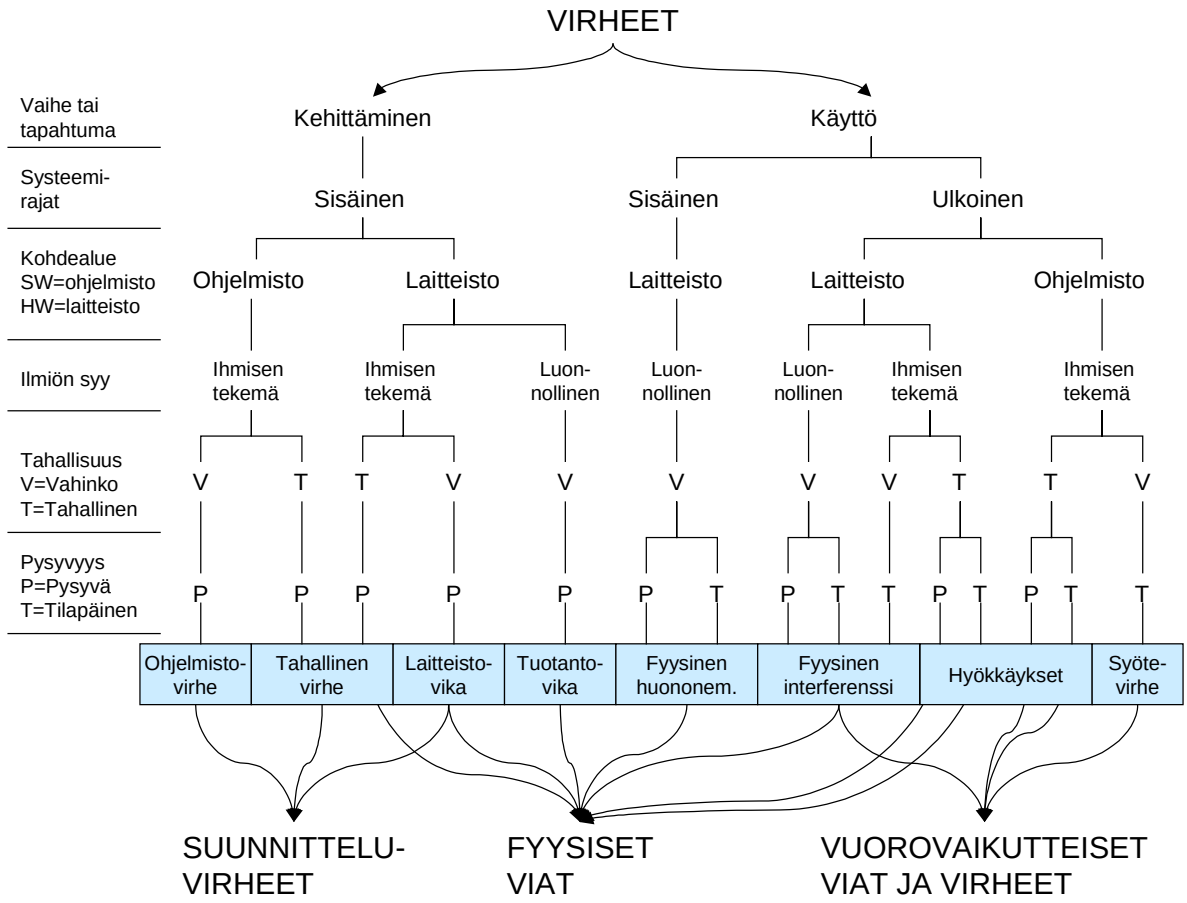
3. Tunnistamisaste, joka kuvaa ohjelman suoritusvaiheen havaitsemisajankohtana.

4. Syy, joka on monitasoinen luokittelu ja ylimpänä on virheiden alkuperätiedot. Ohjelmistolle sopivia syitä ovat vaatimukset tai määrittely, suunnittelu, koodaus ja testaus. Edelleen niillä on alakohtia, kuten suunnittelulla mm. algoritmit, rakenne, data ja liittymät.

5. Eristäminen, jota tarvitaan virheen korjaamiseksi ja huolehtimaan toiminnan häiriintymättömästä jatkumisesta. Luokkia ovat analyysit, testit, simulointi ja erityisinstrumentointi.

6. Korjaavat toimenpiteet, joiden ylin luokka koostuu virheiden alkuperäistiedoista (kohta 1). Ohjelmistolle tarkempia luokkia ovat uudelleen suunnittelu, aloittaminen alusta, toiminta alentuneella suorituskyvyllä, muutos vaihtoehtoiseen toimintaan ja vikautuneen komponentin vaihtaminen.

Virheiden luokittelutapoja on monenlaisia. Avižienisin et al. (2001) luokittelusysteemissä päädytään kolmeen virheiden perustyyppiin kuusiportaisen määrittelymekanismin pohjalta (kuva 15). Virhetyypin määrittäviä asioita ovat ensinnäkin virheen syntytapahtumaa kuvaavat vaiheet, joita voi olla muitakin kuin tekijöiden käyttämät kehitysvaihe ja käyttövaihe. Muita määritysasioita ovat virheen kohdistuminen joko sisäisesti tai ulkoisesti laitteistoon tai ohjelmistoon. Virheen tekemisen tahallisuus, luonnollisuus ja pysyvyys ovat kolme viimeisintä porrasta määrittelymekanismissa.



Kuva 15. Virheluokittelua kuusiaskeleisella määrittelyportaikolla Avižienisin et al. (2001) mukaan.

Erityisesti virheensyöttötekniikassa, johon tässä luvussa vielä palataan, on tärkeää luokitella syötettävät virhetyypit. Virheensyöttötekniikka soveltuu hyvin tarkasteltaessa hetkellisiä virheitä, joita luokiteltiin kuvassa 14.

Pressman (1997) jakaa ihmisen tekemät ohjelmistovirheet seuraavasti:

1. *Suunnitteluvirheet* ovat kehitystyön aikaisia, tahattomia tai tapaturmaisia ilman vahingoittamisen tarkoitusta. Vaatimukset on väärin tulkittu ja siten toteutettu. Ohjelmiston suunnitteluvirheet ovat aina eliminoitavissa ja korjattavissa uudelleensuunnittelulla.
2. *Vuorovaikutteiset virheet* ovat ulkoisia virheitä ilman tahallisen vahingoittamisen tarkoitusta.
3. *Ilkivaltaisia sisäisiä virheitä* ovat mm. virukset, madot, Troijan hevoset, salaovet, loogiset ja aikapommit.
4. *Tunkeutumiset* ovat ilkivaltaisia ulkoisia virheitä. Ne ovat mahdollisia vain, jos järjestelmässä on jokin määrätty suunnitteluvirhe.

Muutokset ja korjaukset ovat virhetilanteeseen johtavista syistä eräitä yleisimpiä. Ulkoisen ympäristön vaatimukset voivat muuttua, suunnittelussa on saatettu havaita muutos- tai parantamistarvetta, jota ei ole viety laadunvarmistuksen prosesseihin.

Virheet voidaan luokitella myös niiden säilyvyyden mukaan. Leveson (1986) erottaa kolmentyyppisiä virheitä: tilapäiset, pysyvät ja ajoittaiset, jotka voivat olla piileviä tai havaittavia. Yleensä kaikki virheet ovat piilevinä ainakin jonkin aikaa, kunnes ne tietyillä mekanismeilla havaitaan. *Tilapäiset virheet* ilmestyvät ja vaikuttavat hetken ja häviävät. Ne aiheuttavat tietokonejärjestelmälle virhetoiminnan, joka on poistunut järjestelmää uudelleenkäynnistettäessä. Niiden syyt ovat tietysti vaikeasti selvitettäviä, usein staattisen sähkön aiheuttamia.

Pysyvät virheet ilmestyvät tiettynä kehitysprosessin hetkenä ja jäävät määräämättömäksi ajaksi. Ne ovat piileviä virheitä, jotka tietyillä ehdoilla etenevät virhetilaan. Ohjelmiston suunnitteluvirheet ovat pysyviä niin kauan kuin ne korjataan uudelleensuunnittelulla. Yksi suunnitteluvirhe voi aiheuttaa useita virhetilanteita ja virhetoimintoja, ennen kuin se riittävän kattavan diagnoosin kautta kokonaan korjataan. *Ajoittaiset virheet* ilmestyvät ja häviävät, syynä on usein lämpöherkkyys.

Beizer (1990) on luokitellut ohjelmistovirheet niiden syntytavan mukaan. Hänen luokittelussaan on useita tasoja, joista toiminnallisten vaatimusten ja ominaisuuksien spesifioinnissa ja toteuttamisessa voi esiintyä seuraavanlaisia virhetyyppejä:

1. Vaatimus on väärä tai virheellinen, ei-toivottu. Vaatimus voi olla oikein määritelty mutta ei haluttu, tarpeeton tai ylimääräinen.

2. Vaatimus on epälooginen: ristiriitainen ja havaitaan yleensä staattisissa analyyseissa. Tai se voi olla

   – kohtuuton: looginen ja johdonmukainen, mutta rajoitteisiin sopimaton

   – saavuttamaton: mahdoton vaatimus esimerkiksi toteutettavaksi annetuilla resursseilla

   – yhteensopimaton: ei sovi muiden vaatimusten tai ympäristön yhteyteen

   – sisäinen: ilmeinen virhe tietyssä komponentissa

   – ulkoinen: ristiriitainen muiden komponenttien kanssa

- yhteensopimaton: vaatimus on yhteensopimaton laitteiston, ohjelmiston tai käyttöjärjestelmän kanssa.

3. Vaatimus on vaillinainen: Vaillinainen määrittely – variaatiot, attribuutit, ominaisuudet ym. ovat määrittelemättä. Tai vaatimus voi olla

   - puuttuva: vaatimusta ei ole määritelty
   - päällekkäinen: vaatimus on määritelty jo muualla
   - geneerinen: vaatimus on oikea ja ristiriidaton mutta liian yleinen sovellettavaksi.

4. Vaatimus ei ole todennettavissa: annetuilla resursseilla vaatimus on mahdoton näyttää toteen millään keinolla. Esimerkiksi oikeat testit voidaan suunnitella mutta ei toteuttaa, tai vaatimukseen liittyy

   - puutteellinen dokumentaatio: vaatimukset ovat oikeita, mutta esitysmuoto ei
   - virheet standardeissa: vaatimusstandardit, joiden perusteella vaatimus on määritelty, sisältävät virheellistä tietoa.

Liityntävirheet alkavat olla pääsyynä ohjelmistopohjaisten järjestelmien ongelmissa. Empiirisiin kokemuksiin vedoten Perry & Evangelist (2000) luokittavat ne seuraavasti:

1. Konstruktio. Konstruktiovirheet ovat ohjelmakielestä aiheutuneita virheitä, jotka syntyvät kun erillisellä ohjelmalla erotetaan fyysisesti liityntäspesifikaatio toteutettavasta ohjelmasta.

2. Toimintojen riittämättömyys. Nämä virheet johtuvat siitä, että järjestelmän joku osa olettaa, että toinen osa ei suorita tiettyjä toimintoja.

3. Toimintojen lisäykset. Kokonaan uuden toiminnon lisääminen voi ”näkyä” järjestelmän muutoksena.

4. Liitynnän väärä käyttö. Tällainen virhe voi syntyä, kun on ymmärretty väärin mitä eri yksiköiden väliseltä liitynnältä vaaditaan.

5. Tietorakenteen muuttaminen. Joko tietorakenteen koko on riittämätön tai sen tietokenttä on riittämätön.

6. Virhetilannekäsittelyn riittämättömyys. Virhetilanteet eivät ole riittävän täsmällisesti ilmaistuja.

7. Jälkikäsittelyn riittämättömyys. Johtaa virhetoimintoon, jossa ei ole vapautettu laskentakapasiteettia käytön jälkeen.

8. Riittämätön liityntätuki. Varustetut toiminnot eivät riittävästi tue määriteltyjä liityntäominaisuuksia.

9. Alustus- ja arvovirheet. Alustuksen virhetoiminta tai oikean arvon sijoittaminen tietorakenteeseen ovat syynä näihin virheisiin.

4.4 Virhealttiit ohjelmistokomponentit

Tietyt ohjelmistonosat ja -komponentit ovat virhealttiimpia kuin toiset. Niiden tunnistaminen on merkittävää sekä korjausten että erityisesti uudistusten yhteydessä, koska parannukset ja järjestelmärakenteen muuttaminen esimerkiksi komponenttia vaihtamalla saattaa aiheuttaa yllätyksiä järjestelmän toiminnassa. Yhtä merkittävää on tunnistaa stabiilit komponentit, jotka eivät ole virhealttiita muutoksille eivätkä johda lisäponnisteluihin.

Virhealttiiden komponenttien tunnistaminen ei ole kuitenkaan kovin helppoa. Tunnistamisessa on otettava huomioon sekä komponentin merkittävyys että sen testikattavuus. Siten testattavuus ei yksin riitä, vaan komponentin toiminnot on myös analysoitava. Kokemus on osoittanut, että virheitä löytyy huonosti testatuista ja toiminnan kannalta kuitenkin merkittävistä komponenteista.

Ohjelmistokomponentit ikääntyvät suhteessaan ympäristömuutoksiin, joita ovat myös komponentin asema järjestelmässä. Ikääntyminen pahenee kehityksen myötä julkaisu julkaisulta. Komponentin kannalta ikääntymistä aiheuttaa puutteellinen dokumentaatio, jossa ei ole riittävästi otettu huomioon järjestelmän toiminnallisuuden ja monimutkaisuuden kasvattamista. Koska aikaa myöten ikääntyminen voi tulla hyvin kalliiksi, on välttämätöntä jäljittää virhealttiit komponentit sekä tarkastella täsmällisesti ikääntymisen syitä.

4.5 Oletukset virhetilanteista

Luotettava järjestelmä koostuu korkealaatuisesta ohjelmistosta, toipumismekanismeista ja korkeatasoisesta ylläpidettävyydestä. Jos kaikki ohjelmiston virheet ja virhetilanteet

kyetään hallitsemaan mm. toipumisaktiviteeteilla, kyse on korkean luotettavuuden ohjelmistosta ja järjestelmästä. Virheitä hallitaan monella eri tavalla, yksi hallitsemistapa on sisäisten tarkistusten sijoittaminen ohjelmaan. Assertioiden (so. tarkistusten) oikeassa sijoittamisessa tulisi tuntea ohjelman virhemekanismit, mikä käytännössä tapahtuu olettamalla tiettyjä virhetilanteita. Samoja virhetilanneoletuksia voidaan erään ehdotuksen (Powell 1995) mukaan hyödyntää määrättäessä sellaisen järjestelmän luotettavuutta, jonka oikeellisuus riippuu näiden oletusten kelpoisuudesta. Tässä kohdassa käsitellään lyhyesti Powellin esittämää lähestymistapaa analysoida formaalisti virhetilanteita, mutta valitettavasti Powellin esittämät virhetilanteisiini perustuvat luotettavuusarvioinnit eivät mahdu tämän julkaisun raameihin.

4.5.1 Virhetilannekattavuus

Virhetilannekattavuus (engl. error coverage) on ehdollinen todennäköisyys sille, että järjestelmä toipuu virhetilanteen ilmaannuttua (Bouricius et al. 1969). Virhetilannekattavuuden luotettavuusvaikutuksen ovat esittäneet monet tahot (mm. Bouricius et al. 1969, Kaufman & Johnson 1999). Virhetilannekattavuus on vaikeasti estimoitavissa, ja siksi anlyyttiset menetelmätkin ovat vähissä. Kehitetyt menetelmät valtaosin vaativat hyvin spesifisiä parametreja, joiden hankkiminen on vaikeaa. Koska riittävä testaaminen on usein taloudellisista syistä perusteetonta, virhetilannekattavuutta on estimoitu tilastollisilla malleilla.

Tilastollisessa lähestymistavassa aluksi valitaan virhe(tilanne)joukosta satunnaisesti virhe, joka syötetään joko järjestelmän prototyyppiin tai malliin ja tarkastellaan sen jälkeen virheen vaikutusta. Suoritettavien syöttöjen lukumäärä on verrannollinen vaadittavaan tietyn järjestelmän virhetilannekattavuuteen. Virheen syöttömenetelmää tarkastellaan lähemmin tämän luvun seuraavassa kohdassa, mutta tässä yhteydessä mainittakoon, että tämä menetelmä vaatii yleensä hyvin suuren määrän syöttöjä, mikä osaltaan vaikuttaa menetelmän käyttökelpoisuuteen. Tulisikin suunnittelussa jo ottaa huomioon, miten merkittävää on selvittää virhekäyttäytymistä ja suunnitella virheensyöttöä annettujen resurssien suomissa mahdollisuuksissa. Mahdollisesti resurssit eivät riitä vaadittavan virheensyöttökattavuuden täyttämiseksi, mikä merkitsee järjestelmän uudelleensuunnittelua siten, että vaadittava luotettavuustaso voidaan saatavilla olevilla menetelmillä ja resursseilla saavuttaa. Tarvitaankin estimointimenetelmiä, joilla suunnittelija kykenee arvioimaan järjestelmän kattavuustekijät.

Mahdollisten ohjelmistovirheiden järjestelmätason vaikutukset on tunnettava. Muutoin ei voi olla kyse riittävästä luotettavuudesta. Tunnistaminen voi alkaa käyttökokemuksista ja virhetilanteiden kvantitatiivisesta tai kvalitatiivisesta mallinnuksesta. Tunnistamista edesauttaa geneerinen tietämys virhemekanismeista, mitä ilman päätök-

senteko ja ohjelmistotuotannon menettelyt jäävät karkealle tasolle, eikä silloin sovelluskohtainen tietämys ehkä enää ole riittävää. Uudet järjestelmät kyetään myös tehokkaasti validoimaan, kun virhetilannemallit ovat saatavilla.

Käytön aikana havaitaan lisää ohjelmiston virhetilanteita, lähinnä koska käyttötilanteet eroavat kehityksen aikaisista testitilanteista. Testauksissa on vaikeata tai joskus jopa mahdotonta ottaa huomioon kaikkia ympäristötekijöitä ja niiden muutoksia. Ohjelmiston virhetilanteiden ymmärtämiseksi tarvitaan huolellista virhetoimintojen ympäristötarkastelua ja ohjelmistoon korjausten jälkeen tehtyjen muutosten tarkastelua.

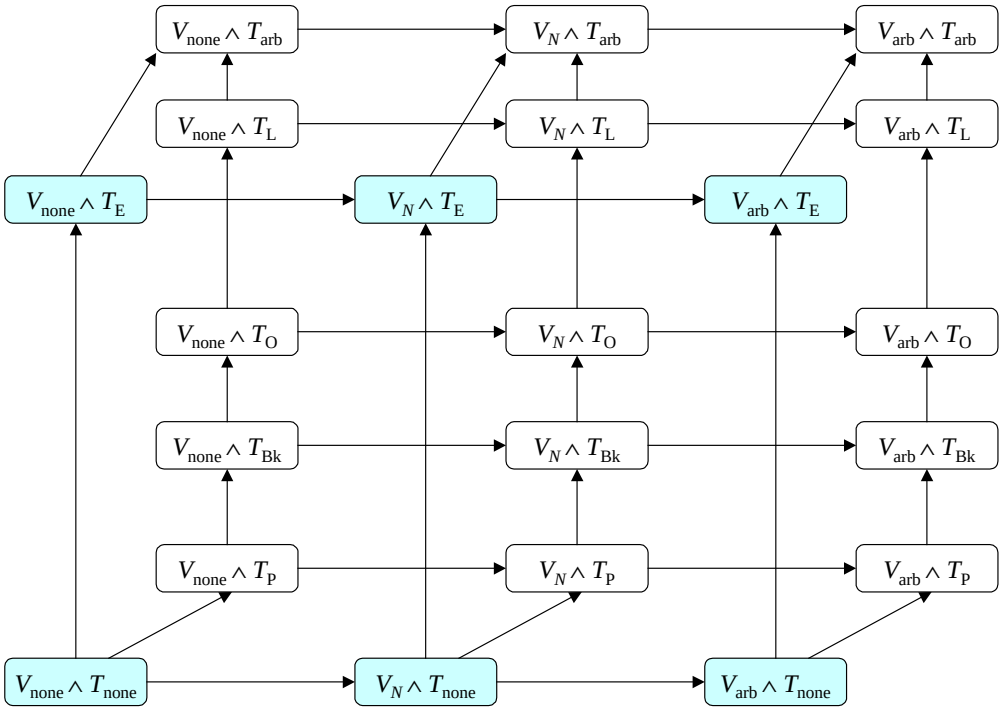
Virhemekanismin selvittämisessä oleellisinta on sekä virhetyyppien, komponenttien vikautumistiheyden että vikautumistavan tunteminen. Tietoa virhemekanismeista voidaan hyödyntää sekä suunniteltaessa että arvioitaessa järjestelmän luotettavuutta. Suunnittelussa vikasietoisuuden täsmällinen rakentaminen on merkittävää, arvioitaessa keskeiseksi tulee myös, millä kattavuudella vikoja/virheitä, virhetilanteita ja vikautumisia/virhetoimintoja on tarkasteltu. Tarkastelun kattavuudella on selkeitä viitekohtia myös valittaessa vioittumistapoja riskilähtöisessä analyysimenettelyssä, jollaisesta esimerkkinä Tiira, joka on kuvattu julkaisussa (Harju 2000). Kelpoisuuteen vaikuttavat sekä standardinomainen analysoimisen lähestymistapa, jota kuitenkin juuri ohjelmistojen tarkasteluun on vaikea määrittää, vioittumistapojen systemaattinen tarkastelutapa sekä prosessien ja artefaktien ottaminen huomioon juuri oikealla tarkkuustasolla.

4.5.2 Datavirhetilanteet

Virhesietoiset varmistukset epäonnistuvat, jos suunnitteluperusteet eli -oletukset osoittautuvat vääriksi tai ne ovat muuttuneet jostakin syystä järjestelmää käytettäessä. Jos suunnitteluolettamukset laaditaan ääritapausten mukaan, pelko olettamusten rikkomisesta on tosin vähäinen. Kuitenkin on tapauksia, joissa virhesietoisuus on rakennettava hyvin monille virhe- tai virhetilannetyypeille, joiden vikautumiset ja vioittumistavat ovat hyvin vapaasti määriteltyjä. Silloin myös virhesietoisuutta on vastaavasti kasvatettava, mikä merkitsee kustannusten kasvamista. Kustannusten kurissa pitämiseksi tulisi Powellin (1995) mukaan löytää juuri oikea virhesietoisuusaste ja täsmälliset olettamukset vioittumistavoista, sillä korkea virhesietoisuusaste merkitsee myös suurta vioittumistapojen määrää.

Kuvan 16 esittämässä kaaviossa virhetilannetyyppejä on kaksi: arvo- ja ajastusvirhetilanteet. Kutsuttakoon niitä tässä datavirhetilanteiksi. Powellin mukaan oletetukset vikautumistapojen kattavuudelle ovat merkittävässä asemassa virhetilanteen syöttömenetelmien toteuttamisessa. Tällä käsitteellä hän tarkoittaa todennäköisyyttä sille, että väitteet, joiden perusteella vikautumistapa muodostetaan, ovat paikkansapitäviä.

Yksi datavirhetilanne voi edetä moduulissa tai siirtyä muihin moduuleihin aiheuttamalla uusia datavirhetilanteita. Datavirhetilanne voi myös häipyä, tai virhe voi korjaantua kesken kaiken ennen virhetilanteen etenemistä tai korruptoitunutta dataa kyetään käyttämään tavalla, mikä ei vaikuta ohjelman käyttäytymiseen.



Kuva 16. Vioittumistapojen seurauskaavio kuvattuna kolmidimensioisella karteesin tulolla. V_x ja T_x ovat vastaavasti datan arvovirhetilanteita ja datan ajastusvirhetilanteita tietyille siirtymille x (Powell 1995).

Kuvan matemaattinen merkintä $V \wedge T$ esittää karteesista tuloa datan arvovirhetilannejoukon V ja datan ajastusvirhetilannejoukon T välillä. Kahden joukon V ja T karteesi tulo määritellään kaikiksi pistejoukoiksi (v, t), missä $v \in V$ ja $t \in T$. Tässä tapauksessa kyse on kolmidimensioisesta karteesisesta tulosta, jossa on määritelty kuvan mukaisesti kaikkiaan 21 yhdistävää siirtymää datavirhetilanteiden välillä. Mielekkäitä siirtymiä on mahdollista määrittää muitakin kuin kuvassa esitetyt. Kuvan virhetilannesiirtymät määritellään seuraavasti:

– V_{none} tarkoittaa sitä, että arvovirhetilanteet ovat oikeita ja tuotteen palvelut tuottavat arvoltaan oikeita tuloksia.

- V_N tarkoittaa tilannetta, jossa ainoat arvovirhetilanteet ovat koodaamattomat[13] arvovirhetilanteet. Tämän tilanteen seurauksena ei välttämättä ole virhetoimintaa.

- V_{arb} tarkoittaa mielivaltaista arvovirhetilannetta, missä arvovirhetilanteen määrittelyä ei ole rajoitettu kvalitatiivisesti tai kvantitatiivisesti.

- T_{none} tarkoittaa sitä, että ajastusarvovirhetilanteita ei ole. Tuotteen palvelut toteutetaan oikea-aikaisina.

- T_O tarkoittaa sitä, että ajastusarvovirhetilanteet johtuvat tekemättä jättämisestä (haluttua tilaa ei saavuteta, ohjelma kaatuu).

- T_L tarkoittaa sitä, että ajastusarvovirhetilanteet ovat myöhässä syntyviä. Palvelu toteutuu joko ajallaan tai myöhässä.

- T_E tarkoittaa sitä, että ajastusarvovirhetilanteet ovat liian aikaisin syntyviä. Palvelu toteutuu ajallaan tai liian aikaisin.

- T_{arb} tarkoittaa mielivaltaista ajastusvirhetilannetta, jossa ajastusvirhetilanteen määrittelyä ei ole rajoitettu kvalitatiivisesti tai kvantitatiivisesti.

Edellä kuvatut datan virhetilanteiden siirtymät ovat tilapäisiä, siis joko hetkittäisiä tai epäsäännöllisiä. Kaksi jäljelle jäänyttä kuvassa esitettyä siirtymää ovat T_P, pysyvä ajastusvirhetilanne ja T_{Bk}, rajattu tekemättäjättö (engl. bounded omission degree). Edellinen kuvaa tapausta, jossa komponentti palvelee tai toimii ajallisesti moitteetta johonkin tiettyyn palveluerään[14] asti, minkä jälkeen palvelu lakkaa. Jälkimmäinen siirtymä, T_{Bk}, rajoittaa pysyvää ajastusvirhetilannetta siten, että komponentti voi laiminlyödä joitakin palvelueriä, mutta *k*-palveluerän laiminlyönnin jälkeen komponentti jättää toimittamatta kaikki loput palveluerät tai yhteydenotot.

Mielivaltaiset arvovirhetilanteet, joita ei ole kvantitatiivisesti tai kvalitatiivisesti rajoitettu mihinkään virhetilannejoukkoon, ovat vaikeasti käsiteltävissä. Käsittely- ja arviointikustannukset kasvavat mm. monimutkaisten ja suorituskyvykkyydeltään alhaisten

---

[13] Mitä hyvänsä *n*-bitin koodia voidaan pitää kaikkien *n*-bitin merkkijonojen osajoukkona. Ne merkkijonot, jotka kuuluvat tähän tiettyyn osajoukkoon ovat koodattuja sanoja, kun taas tähän osajoukkoon kuulumattomat merkkijonot ovat koodaamattomia sanoja. Yksinkertainen sääntö virhetilanteiden ilmaisemissa : jos merkkijono on koodisana, merkkijono on oikein, muussa tapauksessa virheellinen.

[14] Palvelu on palveluerien jono s_1 , s_2 , s_3 , …

virhetilanteiden paljastusmenettelyiden takia. Mielivaltaisia arvovirhetilanteita tulisi välttää kohdistamalla suunnittelua virheettömämpään suuntaan.

4.6 Virheen ja virhetilanteen syöttäminen ohjelmaan

Dynaamista virheen tai virhetilanteen arvausta tai syöttämistä (Error Guessing, Fault Seeding, Infection/Injection Analysis) on sovellettu osaksi elektroniikan ja ohjelmiston validointia (Clark & Pradhan 1995, Benso et al. 1999). Tavoitteena on ollut selvittää millä todennäköisyydellä syötetty virhe eli virhe koodilauseessa johtaa komponentin virhetoiminnoksi. Virheen syöttömenetelmät ovat olleet tutkimuksen avainalueena jo vuosia. Niiden heikkoutena on ollut geneerisyyden puute, sillä menetelmät ovat tietokoneista ja sovelluksista riippuvia. Etuna on selvitys ohjelmistopohjaisen järjestelmän virhemekanismista. Virhetilanteen syöttömenetelmässä prosessia kiihdytetään syöttämällä virheen tilalle suoraan virhetilanne. Silloin virhetilanne valitaan siten, että se edustaa hyvin todellista vaikutustilaa, johon virhe voi johtaa sekä syötejoukko vastaa todellista virheen kautta kulkevaa tilannetta.

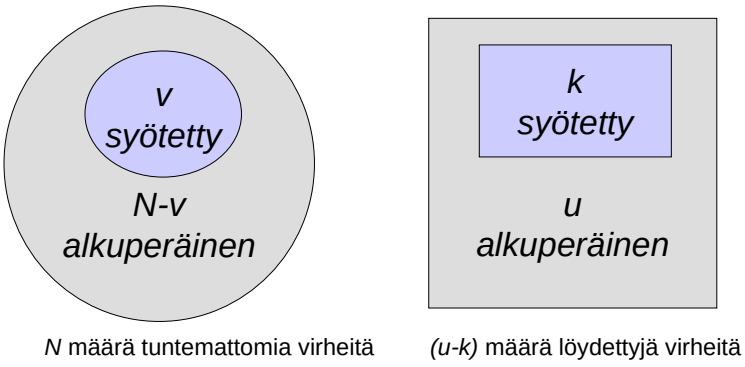
Ohjelmiston virheen syöttötekniikka on peräisin laitteistoille ensin kehitetyistä menetelmistä, kuten niin monet muutkin ohjelmiston verifiointi- ja validointimenetelmät. Ohjelmistossakin menetelmää ensin sovellettiin ohjelmiston satunnaisille laitteistovioille ja ympäristöhäiriöille, ja tälle alueelle tutkimus edelleenkin painottuu, vasta aivan viime aikoina varsinaisille systemaattisille ohjelmistovirheille. Laitteistovikojen sekä ohjelmistovirheiden (tästä eteenpäin vain ’virhe’) syöttämisellä selvitetään vian ja virheen vaikutusta ohjelmiston toimintoihin ja virhesietoisuuden riittävyyteen. Virhetilannetta tosin jäljitellään myös laitteistovioilla virheiden lisäksi (virhetilanteen syöttö).

Virheen syöttämisessä lähdekoodiin ohjelmoidaan keinotekoinen muunnos eli mutantti. Mutantit ovat olleet tutkimuksen kohteena jo pitkään, mutta ei täysin tunneta minkä tyyppisillä mutanteilla kyetään jäljittämään ohjelmointivirheitä. Ohjelmistovirheitä ei kyetä hyvin tunnitamaan järjestelmätasolla sekä laitteiston että ympäristön yhteydessä mitä mutanttien käyttäminen edellyttäisi. Tietämystä lisäämällä pystytään parantamaan virheen syöttötekniikaa ja lisäksi monta muutakin dynaamista ja analyyttista tekniikkaa. Toisaalta virheen syöttötekniikan käyttämisestä saadaan lisätietoa ohjelmiston virhemekanismista. Tiedosta on hyötyä ennustettaessa järjestelmän toimintavarmuutta tai yleensä suorituskykyä mm. arvioitaessa kattavuustekijöitä, jotka ovat parametreina monissa analyyttisissa luotettavuusmalleissa.

Virheen syöttötekniikan käyttäminen todellisessa käyttöympäristössä vastaa kokemusperäistä tietoa. Siten käyttökokemuksia saadaan nopeammin kuin odottelemalla ohjelmiston virhetoimintatietoja käyttäjiltä. Lisäksi virhetilanteiden syöttäminen virheiden

sijasta nopeuttaa edelleen vikautumisprosessia. Kvantitatiivisessa todennäköisyyslaskennassa tulee kuitenkin muistaa, että lopputuloksen täytyy olla sama, syötettiin sitten joko virhe tai virhetilanne järjestelmään.

Perusmallit ja samalla suosituimmat virheen syöttömallit pohjautuvat Shick & Wolvertonin (1973) esittämiin teorioihin.



Kuva 17. Virheen syöttömallin käsitteet.

Kuva 17 selvittää yksinkertaisesti virheen syöttömallin toimintaa. Vasemmanpuoleisessa ympyrässä ovat kaikki ohjelmiston *N* virheet, joihin sisältyy *v* syötettyä virhettä. Oikeanpuolisessa nelikulmiossa on *(u+k)* testeissä löydettyä virhettä. Viime mainituista virheistä *u* on alkuperäisten ja *k* syötettyjen virheiden lukumäärä. Jos *N* ja *(u+k)* ovat suuria, syötettyjen ja alkuperäisten virheiden suhde pysyy suunnilleen samana. Siten,

$$\frac{v}{N-v} \cong \frac{k}{u} \tag{2}$$

Ratkaisemalla saadaan

$$\hat{N} = \left\lceil \frac{v(u+k)}{k} \right\rceil \tag{3}$$

tai

$$\hat{N} - v = \left\lceil \frac{v \cdot u}{k} \right\rceil \tag{4}$$

Jos jatketaan testaamista siihen asti kunnes kaikki syötetyt virheet on löydetty, ts. *k=v*, jäljelle jäävien virheiden määrä voidaan estimoida nollaksi. Vaikka tämä onkin validi estimaatti, sillä ei ole tilastollista merkitystä. Sen sijaan voidaan estimoida todennäköi-

syys sille, että jäljelle jäävien virheiden lukumäärä on vähemmän tai yhtä paljon kuin jokin luku *w*. Tehdään prioriolettamus siitä, että

$$P(N = n) = \begin{cases} 0, & n \prec u + v \\ C, & n \geq u + v \end{cases} \tag{5}$$

missä *N* on todellinen virheiden lukumäärä (alkuperäiset ja syötetyt) ja *C* on jokin vakio. Laskennan helpottamiseksi oletetaan virheiden lukumäärä *n* äärettömäksi ja saadaan yhtälön (6) esittämä approksimaatio.

$$P(n \leq (u + v + w) | (u + v) \textit{löytyneet}) \approx 1 - \frac{\binom{u+v-1}{v-1}}{\binom{u+v+w}{v-1}} \tag{6}$$

Menetelmä soveltuu erityisen hyvin sellaisiin kriittisiin ohjelmistoihin, joista ei ole löydetty yhtään virhettä testauksissa. Tuloksena on tällöin luottamus virheettömyyteen, ei luotettavuusarvioon. Arvio ei siten ole käyttökelpoinen estimoitaessa ohjelmistosta ja laitteistosta koostuvan kokonaisuuden luotettavuutta. Menetelmässä vaaditaan erittäin paljon syötettyjä virheitä. Pienellä määrällä ei saavuteta kriittisille sovelluksille riittävän korkeaa luottamustasoa.

Niissä erityistapauksissa, joissa alkuperäisiä virheitä ei löydetä ja kaikki syötetyt virheet löydetään, saadaan todennäköisyydeksi sille, että ohjelmisto ei sisällä yhtään virhettä:

$$P((n \leq v) \cap (w \leq 0) | (v \textit{ löytyneet}) \cap (u = 0) = \frac{v - 1}{v} \tag{7}$$

Menetelmän kelpoisuutta arvioitaessa täytyy kyeetä vastamaan moniin kysymyksiin:

- Mitä lukemia tulisi kerätä ja mitä arvoja laskea? Entä miten lasketut arvot saadaan liittymään luotettavuusmalleihin?
- Miten hyödyntää käyttöprofiileja?
- Miten virhetilannemalli kuvaa riittävästi virheitä?
- Mihin ja milloin virhetilanne tulisi syöttää?

Ensimmäiseen kysymykseen ovat antaneet teoreettisen vastauksensa sekä Powell et al. (1995) että Arlat et al. (1993), toiseen osaltaan Musa (1993) aiheellaan STUTista (Sta-

tistical usage testing). Christmansson & Chillerege (1996) ovat osaltaan ratkaisseet kolme viimeisintä ongelmaa kehittämällään menetelmällä, joka käyttökokemustiedoista generoi virhetilanteen syöttöpaikan ja muut ehdot syöttötekniikkaa varten. Heidän mukaansa virhetilannejoukko koostuu virhetilannetavoista, -paikasta ja -syöttöehdoista. Virhetilannejoukon tulisi kuvata hyvin sekä kokemusperäisesti havaittuja virheitä että nimenomaan ohjelmistovirheitä eikä niinkään laitteistovikoja.

Virhe/virhetilanne-syöttömenetelmää varten Chillarege et al. (1992) määrittelevät koodivirheet viiteen tyyppiin:

- Sijoitukset, joissa arvo määrätään virheellisesti tai ei määrätä lainkaan.
- Tarkistukset, joissa ehtolauseiden parametrien tai datan validointi on puutteellista tai virheellistä.
- Algoritmit, joiden suorituskyky- tai oikeellisuusongelmat on korvattavissa suorittamalla algoritmi uudelleen tarvitsematta muuttaa suunnitelua.
- Ajastukset ja jonotukset, joissa jaettujen resurssien jonotustekniikka joko kokonaan puuttuu, kohdistuu väärään resurssiin tai on väärä teknikka.
- Liityntä- ja tietoliikenneongelmat käyttäjien, moduulien, komponenttien tai laiteajurien ja ohjelmiston välillä.

Ohjelmistotriggerit ovat ympäristöehtoja tai -tapahtumia, jotka katalysoivat virheen siirtymisen virhetoiminnaksi. Tavallisempia triggereitä ovat mm. seuraavat:

- Järjestelmän (uudelleen)käynnistäminen, jossa järjestelmä alustetaan sen sammuttamisen tai siihen sattuneen virhetoiminnon jälkeen.
- Työkuormitus, jossa järjestelmä toimii lähellä resurssirajojaan, joko ylä- tai alarajalla.
- Toipuminen ja poikkeuskäsittely estävät normaalisti virheen etenemisen. On kuitenkin tapauksia, joissa toinen virhe on päässyt etenemään nimenomaan poikkeuskäsittelyn aktivoimana.
- Epätavalliset laitteisto/ohjelmisto-kokoonpanot voivat trigata virhetilanteen syntymisen.

– Normaalimoodi voi trigata virhetilanteen syntymisen silloin, kun järjestelmä toimii hyvin ala- tai yläresurssirajoilla. Muissa ’normaalitiloissa’ järjestelmään tulevat eritysehdot triggaavat virheen virhetilanteeksi.

Virhetoiminnan vakavuus ilmaisee käyttäjän ongelman suuruusluokan. Vakavuutta on luokiteltu usealla tavalla (esimerkki taulukossa 3).

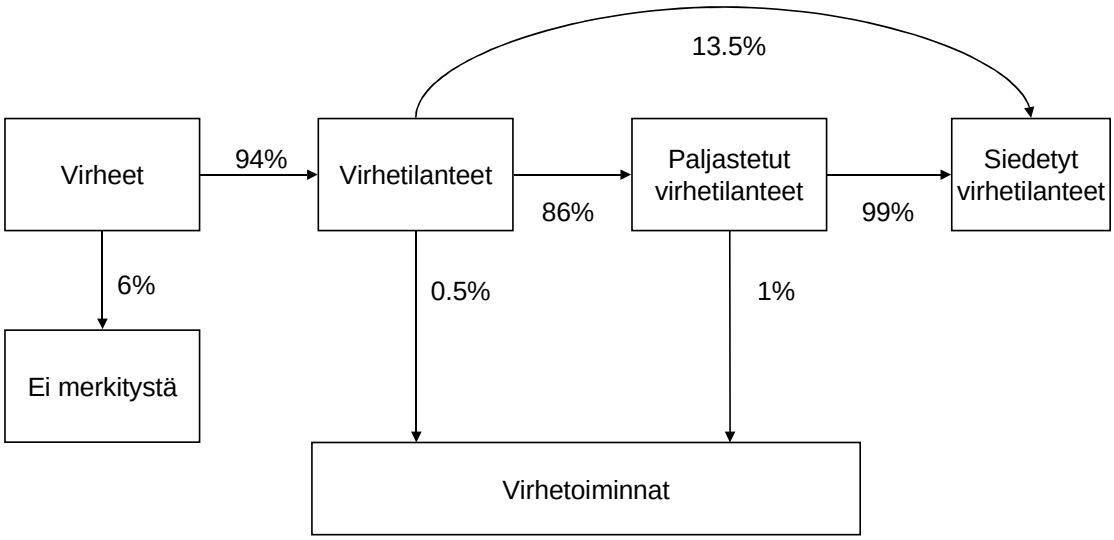
Taulukko 3. Virhetoiminnon vakavuusluokittelu.

Vakavuusluokka	Merkitys
1	Toiminto/tuote on tärkeä, mutta sitä ei pystytä saamaan käyttöön.
2	Toimintoa pystytään käyttämään, mutta sen suoritustaso on selvästi heikentynyt.
3	Toimintoa pystytään käyttämään vain rajoitetusti.
4	Toiminto aiheuttaa vain vähän ongelmia tai ei lainkaan.

Virheensyöttömenetelmässä virheiltä oletetaan, että ne syntyvät tai voivat syntyä ohjelmistokehityksen aikana, voivat aiheuttaa virhetilanteen ja voivat vaikuttaa ohjelmiston käyttäjään vakavuusluokalla 1–3.

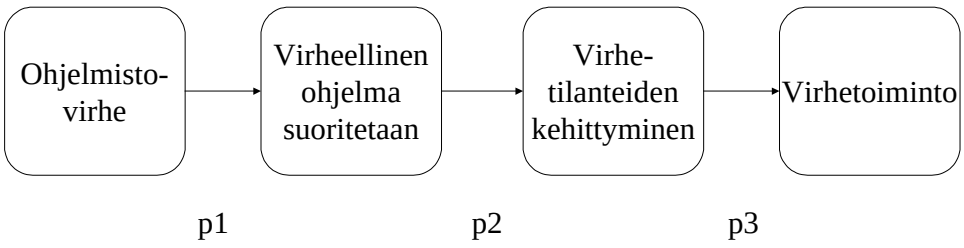
Virhetilanteen syöttömenetelmä kuuluu eksperimentaalisiin menetelmiin. Menetelmän tulokset on helposti analysoitavissa mm. virhesietoisuuden selvittämiseksi tai luotettavuusarvojen laskemiseksi. Edellisestä Arlat et al. (2001) esittää esimerkin, jota esittää kuva 18. Analyysista selviää, että virhetoiminnot tapahtuvat 1,5 %:n kattavuudella, eli sillä osuudella virhetilanne johtaa virhetoiminnoksi huolimatta tarkistuksista ja korjauksista sekä virhesietoisten menetelmien asettamisista.

Esimerkissä virheensietomekanismi käsittelee yksittäisiä virhetilanteita, joita ei ole saatu paljastetuiksi virheendetektointimekanismeilla. Kyse on siis moninkertaisesta virheestä, missä ohjelmavirheen lisäksi virheen käsittelymekanismit ovat virheellisiä tai niiden määrittelyt eivät ole oikeita. Virheensietoisuuden kattavuus on voinut jäädä vajaaksi, vaikka kattavuusluku onkin saatu korkeaksi. Virheensietomekanismi voi käsitellä sellaisia virhetilanteita, joita järjestelmässä ei ole, ilmaisten ylimääräisen kattavuusosuuden. Moninkertaiset virheet ovat Grayn (1990) tutkimuksen mukaan luultua yleisempiä. Hän raportoi, että peräti 20 % virhetoiminnoista aiheutuu vähintään kolmesta perättäisestä virhetilanteesta.



Kuva 18. Esimerkki virheensyöttömenetelmällä tehdyn koetarkastelun analysoinnista, missä selvitettiin virhesietoisuuden kattavuutta. Kaaviossa prosentit kuvaavat kattavuusarvoja virhemekanismin eri vaiheissa. Esimerkiksi 94 % syötetyistä virheistä johtaa virhetilanteeseen, 6 %:n jäädessä merkityksettömiksi. Virhetilanteista 86 % pystytään paljastamaan ja niistä 99 % estämään johtamasta virhetoiminnaksi virhesietoisilla menetelmillä. Virhetoiminta aiheutuu 1,5 %:n kattavuudella (Arlat et al. 1990).

Kuvan kaavio on hyvin yleisellä tasolla kuvatessaan virheensyöttökokeen tuloksia. Yksityiskohtaisempia vastaavia analyyseja ovat tehneet mm. Fabre et al. (1999) ja Hiller et al. (2001).



Kuva 19. Ohjelmistovirheen etenemistodennäköisyys.

Todennäköisyys sille, että virhe etenee virhetoiminnoksi riippuu voimakkaasti käyttöprofiilista, kuten kuvan 19 esityksestä voidaan todeta. Olettamalla, että ohjelmisto si-

sältää virheen, virheellisen kohdan suoritustodennäköisyys on p_1. Kun virheellinen koodi suoritetaan, virhetilanteen syntymistodennäköisyys on p_2 ja kun virhetilanne on olemassa, sen etenemistodennäköisyys virhetoiminnoksi on p_3. Siten virhetoiminnan esiintyminen on näiden todennäköisyyksien p_1, p_2 *ja* p_3 tulo.

4.7 Ohjelmiston yhteisvirheet

Kohdassa 4.5 tarkasteltiin erästä mahdollisuutta virhetilanteiden täsmälliseen mallintamiseen ja kohdassa 4.6 virhetilanteiden syöttämistä. Niillä yhdessä muiden menetelmien kanssa validoidaan ohjelmistojärjestelmän riittävä luotettavuus. Tässä kohdassa käsitellään ohjelmiston yhteisvirheiksi tai yhteisvirhetilanteiksi katsottavia ohjelmistovikoja, jotka oleellisesti kuuluvat kummankin edellisen kohdan aihepiiriin.

Onnettomuuksien on useissa tapauksissa todettu aiheutuneen useasta eri syystä kuten kohdassa 4.2. Ohjelmiston virhetoiminnan alkutapahtuma on yleisemmin laitteistoviassa kuin ohjelmistovirheissä. Laitteistovika aiheuttaa uuden suorituspolun, jota ei ole osattu ottaa testauksissa huomioon. Ohjelmistopolulla sijaitseva virhe voi siten siirtää laitteistoviasta alkanutta vikaketjua eteenpäin. Siten laitteistovika esimerkiksi redundanttisen järjestelmän toisessa kanavassa voi aiheuttaa kummankin kanavan vikautumisen virheellisen ohjelmistotoiminnan kautta.

4.7.1 Riippuvat virhetoiminnot

Järjestelmän vikautumiset aiheutuvat joko satunnaisista (laitteisto)vioista tai systemaattisista virheistä. Edelliset pätevät mille tahansa komponentille, ja niiden vikojen seurauksena on komponentin sisältävän osajärjestelmän, esimerkiksi kanavan, vikautuminen. Monikanavaiselle järjestelmälle riippumattomien satunnaisten laitteistovikojen todennäköisyys kyetään laskemaan nykyisillä menetelmillä. Systemaattisten virheiden osuus yksikanavaisissa järjestelmissä on suhteellisen pieni verrattuna satunnaisiin vikoihin, mutta systemaattisten virheiden esiintymistodennäköisyys kasvaa rinnakkaisilla redundanttisilla järjestelmillä. Systemaattiselle vikautumiselle pätevät myös jossakin määrin satunnaistapahtumat, mm. ohjelman suorituspolun kulkemista virhetilanteen kautta pidetään satunnaisena ilmiönä.

Tietyt vikautumiset ja virhetoiminnat voivat syrjäyttää järjestelmän varmistukset, diversiteetit ja syvyyssuojaukset ja mitätöidä olettamuksen riippumattomuudesta. Nämä riippuvat toiminnat johtuvat ympäristötekijöistä, suunnitteluvirheistä, kalibrointivirheistä ja toiminnallisista poikkeamista. Niitä kutsutaan yhteisvioiksi (Common Cause Failure,

CCF) tai yhteisvioittumistavoiksi (Common Mode Failure, CMF). Luotettavuustarkastelut ovat oikea tapa tunnistaa riippuvia virhetoimintoja.

Tietokoneissa yhteisvika määritellään usealla tavalla. Kaufman ja Johnson (1999) esittivät yhteisvikautumistavan määrittelyn lyhyen historian alkaen Gangloffin (1975) esittämästä tarkastelusta, jossa usea kohde vikautuu samasta syystä. Heidän määrittelynsä ei sisältänyt sellaisia tärkeitä ominaisuuksia kuten aikatekijät ja vikautumistavat. Aikatekijät olivat jo mukana Watsonilla ja Edwardsilla (1979), jotka tarkastelivat komponenttien riippuvia vikatilanteita redundanttisissa erillisissä kanavissa. Heidänkään määrittelyssään ei vielä ollut mukana vioittumistapoja, jotka monen muun vikakäsitteen (mm. esiintymistiheys, syylajit) mukana olivat Colombo & Kellerin (1987) tarkasteluissa. He määrittelivät yhteiseen vikautumistapaan kuuluviksi moninkertaiset, samanaikaiset, samanlaiset ja riippuvat komponentit, jotka vikautuvat samalla tavalla.

Ohjelmiston CMF-tarkastelua ovat suunnanneet Voas et al. (1997), jotka määrittelivät ohjelmiston yhteisvikautumistavan virhetoimintona, joka tapahtuu, kun kaksi tai useampia ohjelmistoversioita vikautuu täsmälleen samalla tavalla samoista syötteistä. Määritelmä ei kuitenkaan ole riittävä, sillä se sulkee pois saman yksittäisen tapahtuman tai syyn seurauksena sattuneet eri ohjelmistotoimintojen vikautumiset. Jälkimmäiselle on tunnusomaista virheen perättäisvikautumisen luonne, jossa yksi ohjelmistovirhe pääsee etenemään jonkun sekundäärisen vian vallitessa kahteen tai useampaan ohjelmistoyksikköön aiheuttaen niiden virhetoiminnot. Tyypillisimmillään tällainen sekundäärinen vika on laitteiston satunnaisvika, mikä tosin saattaa kaataa kaikki ohjelmistotoiminnot yksinkin, mutta tässä käsiteltävässä tapauksessa vain esiintyvän ohjelmiston virhetilanteen kautta.

Taulukko 4. Riippuvat viat, jotka sulkevat pois kaikki riippumattomat viat (Kaufman & Johnson 1999).

<table>
<tr><td colspan="2">Riippuva vika</td><td>Tapahtumajoukko, jonka todennäköisyyttä ei voida ilmaista yksittäisten tapahtumien ehdottomalla vikautumistodennäköisyydellä.</td></tr>
<tr><td></td><td>Yhteisvika</td><td>Samasta syystä peräisin olevat samanaikaiset tai lähes samanaikaiset viat useassa redundanttisessa yksikössä</td></tr>
<tr><td></td><td>Yhteisvioittumistapa</td><td>Usea yksikkö vikautuu samalla tavalla (sama vioittumistapa)</td></tr>
<tr><td></td><td>Perättäisvika</td><td>Kaikki muut riippuvat viat kuin yhteisviat, ts. ne eivät vaikuta redundantteihin komponentteihin</td></tr>
</table>

Valtaosa CMF:ien määritelmistä kehitettiin laitteistojen vikautumisilmiöiden tarkasteluun. Ohjelmiston näkökulmasta CMF määritellään virhetoiminnoksi, joka tapahtuu, kun kaksi tai useampia ohjelmistoversioita vikautuu täsmällisesti samalla tavalla samoista syötteistä. CMF:t, jotka ovat virhetoimintoja varmistusten välillä, kattavat kaikki myös ns. kausaalisuhteet, joilla tarkoitetaan pientä viivettä virhetilanteen etenemisessä ohjelmistoyksiköstä laitteistoyksikön toimintahäiriöksi ja päinvastoin.

CCF on CMF:n lisäksi toinen ohjelmiston luotettavuusanalyyseissa tunnistettava vikautumistyyppi, joka on saanut lähtökohtansa myös laitteistotarkastelun puolelta. Tässä luvussa CCF:t määritellään[15] samanaikaisiksi, kahden tai useamman ohjelmistokomponentin samasta yksittäisestä alkutapahtumasta johtuvaksi riippumattomaksi virhetoiminnaksi, mikä voi johtaa kaikkien komponenttien epäkäytettävyyteen ja aiheuttaa systeemitason vikautumisen (Dhillon & Anude 1994, NRC 1997).

Perättäisvika määritellään laitteistolle yhdestä alkutapahtumasta ja virheellisistä välitapahtumista aiheutuvina vikautumisina. Määrittelyä voidaan soveltaa myös ohjelmistolle, jolloin kyse on perättäisestä virhetilanteesta, joka alkaa yhdestä tai usemmasta ohjelmistovirheestä, mutta johtaa ohjelmiston virhetoimintaan vain jos suoritepolku sisältää ainakin yhden virhetilanteen, joka on lähtöjään toisesta ohjelmistovirheestä, laitteistoviasta tms. Tässä perättäisvirhetilannetta kutsutaan riippuvaksi viaksi, sillä sen toteutuminen edellyttää yhteistä suoritepolkua, jota ilman virhetilanne ei pääsisi etenemään. Virheen siirtyminen ohjelmistokomponentin sisällä ja eteneminen toiseen komponenttiin ovat näitä perättäisiä virhetilanteita.

CMF:n ja CCF:n välinen ero ei ole itsestään selvä. Toisaalta CMF kuuluu CCF:n alaryhmään, toisaalta suhde voi olla päinvastainenkin. Yhden näkökulman mukaan redundanttisten järjestelmien samasta syystä johtuvat vikautumiset samassa ajassa olisivat CMF:iä. Erot näiden kahden riippuvan vikautumistyypin välillä olisivat siten seuraavat:

1. Perättäiset tai sekundääriset vikautumiset redundanteissa komponenteissa. Ne ovat alkuaan riippumattomia vikautumisia, mutta etenevät sekundäärisistä vaikutuksista aiheuttaen lisävikautumisia.

2. CCF:t ovat toiminnallisesti riippumattomien redundanttisten komponenttien vikautumisia, joiden syyt ovat ulkoisissa alkutapahtumissa.

---

[15] CCF:t määritellään muullakin tavalla. Watson & Edwards (1979) määrittelevät CCF:t useaksi mahdollisiksi tai todellisiksi vikautumisiksi. Vastaavasti Watson & Smithin (1980) määrittelevät CMF:t useaksi identtisten redundanttisten komponenttien vikautumisiksi samalla vikautumistavalla sellaisessa kriittisessä aikajaksossa, josta seurauksena on täydellinen systeemin vikautuminen. Kaikki komponentit vikautuvat välittömästi yhteisestä syystä.

3. CMF:t ovat redundanttisten komponenttien vikautumisia, joiden syyt ovat komponentin yksittäisessä vikautumisessa ja etenemisessä toiminnalliseen riippuvuuteen.

Eräs järjestelmän turvallisuuden menetelmä arvioitaessa laitteistovikojen merkitystä ohjelmiston toiminnalle on simuloida viat ohjelmistossa. Päämääränä on tarkastella jatkaako ohjelmisto toimintaansa turvallisesti laitteiston vikaannuttua. Yleensä eri laitteistoviat aiheuttavat erilaisen ohjelmiston käyttäytymisen, mistä seuraa, että turvallisen käyttäytymisen selvittämiseksi olisi syötettävä kaikki mahdolliset laitteistoviat. Mutta jos seurauksena onkin samanlainen käyttäytyminen, kyse on yhtenäisestä etenemisestä, mikä helpottaa huomattavasti järjestelmän riittävän turvallisuuden arvioimista.

Virhetilanne etenee yhtenäisesti (Michael & Jones 1997), kun ne kaikki etenevät samoilla muuttujilla ja syötteillä samaan virhetoimintoon tai niistä yhdestäkään ei seuraa mitään virhetoimintoa. Jos käytännössä kyettäisiin täsmällisesti tunnistamaan sellaisten virhetilanteiden joukko, joiden eteneminen aina annetuissa puitteissa johtaa tietynlaiseen ohjelman käyttäytymiseen, kyettäisiin yksinkertaistamaan sekä suunnittelua että testaamista. Tunnistamisen onnistuessa riittää, kun syötetään vain yksi virhetilanne ohjelmaan ja se kuvaisi kaikkia muita joukon virhetilanteita.

Yhtenäinen virhetilanteen eteneminen perustuu yleiseen väitteeseen ohjelmiston virhetoiminnon syntymisestä. Väitteen mukaan virhetoiminto voi syntyä vain, jos seuraavat kolme tekijää vallitsevat:

1. Vian sisältävä ohjelmakohta suoritetaan.

2. Vika aiheuttaa virhetilanteen.

3. Virhetilanne etenee virhetoiminnaksi.

Todennäköisyyspohjaisissa analyyseissa estimoidaan, millä todennäköisyyksillä vian sisältävä ohjelmakohta suoritetaan, vika etenee virhetilanteeksi ja virhetilanne muuttaa ohjelman käyttäytymistä. Jotta vian eteneminen virhetilanteeksi kyettäisiin estimoimaan, täytyy kaikki viat simuloida ohjelmaan sekä vastaavasti simuloida kaikki virhetilanteet, jotta kyetään päättelemään niiden aiheuttamat mahdolliset virhetilanteet. Syötettäviä vikoja ja virhetilanteita tulisi olemaan niin runsaasti, ettei simuloiminen olisi enää kannattavaa. Yhtenäisten virhetilanteiden löytyminen olisi siten hyvin hyödyllistä. Ne kattaisivat ohjelmoijan tekemät tyypilliset virheet ja laitteiston tyypilliset viat sekä niistä seuraisi samoja tyypillisiä virhetilanteita ja virhetoimintoja.

Tekijöiden (Michael & Jones 1997) kokeellisissa tutkimuksissa havaittiin, että yhtenäisiä virhetilanteita on yllättävän paljon. Niiden etsiminen kokeellisesti on helppoa: syö-

tetään ohjelmaan muutama virhetilanne ja jos ohjelma käyttäytyy niissä samalla tavalla, on otaksuttavaa, että useimmat vastaavat virhetilanteet aiheuttavat saman käyttäytymisen. Kuitenkin Hiller et al. (2001) arvostelevat tekijöiden tutkimusta. Arvostelijoiden tutkimukset eivät tue yhtenäisen virhetilanteen etemisen yleisyyttä, mikä vähentää sen hyödyllisyyttä ohjelmiston luotettavuuden ja turvallisuuden suunnittelussa ja arvioimisessa.

Hiller et al. (2001) esittävätkin oman teoriansa lähestymistavaksi, jossa otetaan huomioon myös virhetilanteiden tunnistaminen. Heidän menetelmässään mitataan ohjelmistomoduulin virhetilanteen läpäisevyyttä, mikä on uusi käsite ohjelmistojen luotettavuusalalla. Tällaisen mittauksen onnistuessa kyettäisiin paikantamaan eteneville virhetilanteille alttiit ohjelmistomoduulit ja asentamaan virhetilanteiden tunnistus- ja toipumismekanismeja juuri oikeaan kohteeseen. Tutkimuksen tuoreudesta johtuu, ettei sitä ole vielä kritisoitu alan kirjallisuudessa.

FMECA (Failure Mode Effect and Criticality Analysis) -tyylisillä tarkasteluilla kyetään jo varhaisista artefakteista tunnistamaan kriittisiä kohtia ja täsmentämään suunnittelua tai virhetilanteiden etsintää koodista, mutta virhetilanteiden siirtymisessä moduulin sisällä ja etenemisessä toisiin moduuleihin FMECA-tarkastelut ovat vielä kehittymättömiä. Niitä tulisi soveltaa erillään muusta tarkastelusta nimenomaan siirtymä- ja etenemistarkasteluihin. Tutkimus- ja kehitystyö etenemisanalyysien suunnittelemiseksi antaakin oman täydennyksensä tunnettuihin analyyseihin.

Edellä mainitut tutkimukset ovat vain yksi poiminta virhetilanteiden etenemistutkimuksista viimeisen kolmenkymmenen vuoden aikana. Lukuisia algoritmeja ja tekniikoita on esitetty (mm. Roth 1980, Goel 1981, Fujiwara & Shimono 1983). Morell et al. (1997) tarkastelivat menettelyn hyödyllisyyttä lähdekoodin analysoinnissa siten, että kyettäisiin edullisesti valitsemaan mahdollisimman kattavat ja samalla vaikeasti normaalimenetelmin tunnistettavat testitapaukset virheiden löytämiseksi. Kokemusperäisten tulosten nojalla Steininger & Scherrer (1997) kykenevät löytämään optimikombinaatioita, joihin laitteistopohjaiset virhetilanteiden havainnointimekanismit tulisi sijoittaa.

Virhetilanteiden eteneminen ohjelmistokomponentista toiseen komponenttiin on yksi niistä syistä, minkä takia valmiista komponenteista kootun järjestelmän luotettavuusanalyysi on vaikeaa. Virhetilanteiden etenemisanalyysit kehittyessään tuovat oman lisänsä ongelmien ratkaisemiseen. Toisen tuovat tietovuoanalyysit ja tila-analyysit, joilla saadaan selville, minne data virheineen siirtyy ja etenee. Näiden lisäksi luotettavuusanalyytikot ovat tutkineet todennäköisyyspohjaisten laskelmien tekemistä komponenteista koostuville järjestelmille.

Ohjelmistojärjestelmän luotettavuus yleensä estimoidaan järjestelmän testaustuloksista kooduista tiedoista (Musa 1998). Testauksissa järjestelmää käsitellään kokonaisuutena, kun taas tutkimuskohteena on ollut myös kehittää malleja, joilla ohjelmistojärjestelmän luotettavuus estimoidaan järjestelmän rakenneosien luotettavuudesta (mm. Woit & Mason 1998, Gokhale & Trivedi 1998, Hamlet et al. 2001). Malli muistuttaisi laitteiston luotettavuusmalleja, mutta ongelmiksi tulevat rakenneosien eli komponenttien riippuvuudet, joita yhteisinä tekijöinä mallissa on vaikea kuvata. Edellä mainitut virhetilanteiden etenemiset ohjelmistomoduulista toiseen ovat osa riippuvuuksista.

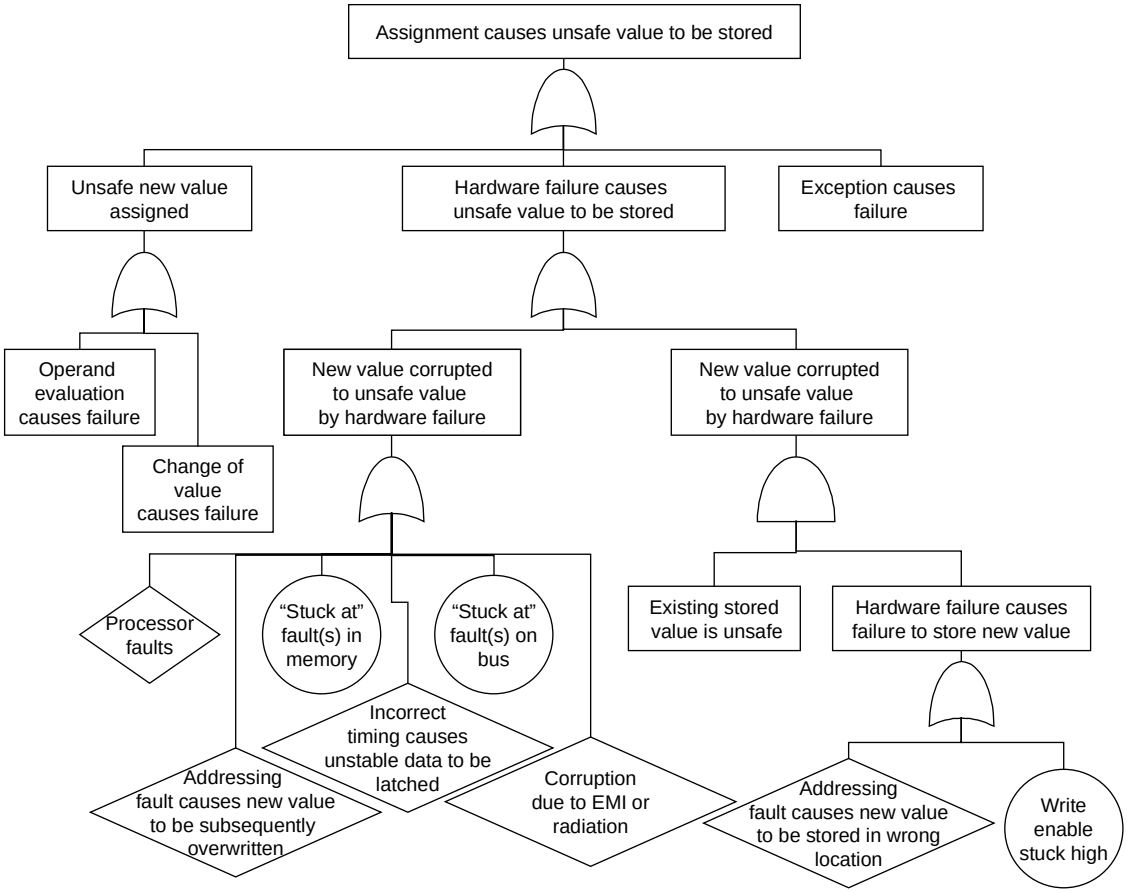
Laitteistopuolella Markov-pohjaisilla malleilla on yleensä laskettu komponenttien luotettavuuksista koostetut järjestelmäluotettavuudet. Mallien toteutusehtoina ovat myös riippuvien tekijöiden mallintamiset. Useat tahot (Lyu 1996) ovat kuitenkin sitä mieltä, että laitteistoille sopivat mallit eivät sovellu ohjelmistoille, joille yleensäkin riippumattomuuden vaatimukset eivät sovellu. Ohjelmistojen kohdalla tarkasteluissa tulisi ottaa huomioon miten riippuvat tekijät mallinnetaan. Markovin mallit eivät sinänsä ota huomioon aikatekijöitä, jotka ovat tärkeitä yhteisvikautumistarkasteluissa. Malleilla voidaan kuitenkin käsitellä perättäisiä tapahtumia ja tätä kautta esittää yhteisvikautumisia. Haittoina ovat kuitenkin mallien kasvaminen liian suuriksi ja monimutkaisiksi yli käsittelyresurssien.

Onnistuessaan ohjelmistojärjestelmän luotettavuuden estimointi siitä koostuvien komponenttien luotettavuuksista toisi kustannustehokkaan menettelytavan, jolla liittää järjestelmään luotettavuussertifioituja COTSeja.

Ohjelmisto liittyy aina laitteistoon. Jos kyse olisi pelkästään ohjelmistosta, olisi mahdollista tarkastella ohjelmiston laatuominaisuuksia millä hyvänsä staattisilla analysaattoreilla. Turvallisuus on kuitenkin järjestelmätason ominaisuus, eikä ohjelmiston täydellinen suorittaminen hyväksytysti ole mahdollista ilman laitteiston täydellisyyden mukaan ottamista. Perinteisillä ohjelmiston staattisilla analyyseilla ei hyväksyttävyyttä kyetä muodostamaan, mutta turvallisuusanalyyseilla tähän pystytään.

Käytännössä tietokoneviat otetaan turvallisuusanalyyseissa huomioon olettamalla, että mikä tahansa vika järjestelmässä riittää aiheuttamaan minkä tahansa vioittumistavan järjestelmän tulosteissa. Siten on tavallista merkitä järjestelmätason vikapuuhun laitteisto-osuudet yksinkertaisesti maininnalla laitteistovika (katso kuva 20). Vikapuita kvantifioitaessa laitteistovioille annetaan konservatiivinen arvo Bayesin malleilla yksittäisistä järjestelmäkomponenttien vikatodennäköisyyksistä. Konservatiivisesta arvosta ei kuitenkaan ole apua ohjelmistosuunnittelijalle, joka kehittää havainnointi- ja suojausstrategioitaan ja haluaa niitä varten tietää täsmällisesti yksittäisten laitteistovikojen vaikutukset ohjelman suoritukseen. Staattisten vikapuiden haitta on lisäksi siinä, ettei niillä

voida ottaa huomioon aikatekijöitä, mikä heikentää vikapuiden käyttökelpoisuutta yhteisvikojen mallintamisessa.



Kuva 20. Vikapuu, jossa laitteistovika myötävaikuttaa sijoituskäskyn epäonnistumiseen (Leveson & Harvay 1983).

Hyviä apuneuvoja täsmälliseen laitteistovikojen huomioon ottamiseen ei suunnittelijalle kuitenkaan löydy turvallisuusanalyyseistakaan. Laitteiston ja ohjelmiston vuorovaikutuksen monimutkaisuus ilmenee kuvan 20 vikapuusta (Leveson & Harvay 1983). Kuva esittää Levesonin ohjelmistovikapuun templettiä sijoituskäskylle. Vikapuussa on mukana ne laitteistoviat, jotka voivat aiheuttaa vaarallisen sijoituskäskyn toteutumisen. Kaikkien käskyjen tai edes osan mallintaminen laitteistoviat huomioonottavalla tavalla on mahdoton tehtävä.

Itse asiassa kaikki kuvan laitteistotapahtumat edustavat yhteisvikoja, jotka voivat mahdollisesti vaikuttaa mihin tahansa ohjelmiston suoritusvaiheeseen. Jotkut, kuten joissakin muisteissa esiintyvät viat, voivat vaikuttaa rajoitetusti, jotkut muut, kuten väylä- ja prosessoriviat voivat vaikuttaa kaikkeen suoritukseen.

5. Ohjelmistomittojen käyttö ohjelmiston luotettavuuden arvioinnin apuna

Mittaaminen on perustavaa laatua oleva toimenpide minkä tahansa prosessin ohjauksen ja hallinnan kannalta. Ohjelmistoon ja ohjelmistokehitykseen liittyviä ominaisuuksia on pyritty mittaamaan yhtä pitkään kuin on ollut järjestelmällistä ohjelmistokehitystäkin. Mittaamisella hankittua tietoutta on käytetty hyväksi arvioitaessa ohjelmiston ulkoisia ominaisuuksia, kuten esimerkiksi ohjelmiston laatua tai ohjelmistoprojektin kustannuksia.

Luotettavuus on laadun keskeinen osa-alue. Ohjelmistomittojen tarjoama informaatio on katsottu merkittäväksi etenkin ohjelmiston elinkaaren varhaisten, ennen ohjelmakoodin suoritusta olevien vaiheiden aikana tapahtuvien laadullisten tarkastelujen kannalta. Ohjelmiston luotettavuutta selittäviä ominaisuuksia ja niitä kuvaavia mittoja on pyritty kartoittamaan, mutta yksiselitteisiä luotettavuutta indikoivia mittoja ei ole pystytty tunnistamaan. Nykykäsityksen mukaisesti luotettavuuden arviointiin tulisi käyttää useita erityyppisiä mittoja, joiden tarjoama informaatio yhdistetään erillisen luotettavuusmallin avulla luotettavuutta indikoivaksi suureeksi (Smidts & Li 2000, Fenton & Neil 1999).

Ohjelmistomittojen ensisijainen käyttö liittyy ohjelmistoprosessin ja -projektin hallinnallisiin toimintoihin. Mittaustulosten yhdistäminen ohjelmistotuotannon resurssien ja kustannusten kartoittamiseen on suhteellisen suoraviivainen ja runsaasti tutkittu alue. Ohjelmistomittojen ja laadun suhde on mutkikkaampi; mittaamista voidaan käyttää toisaalta ohjelmistoprosessin kehittämisen ja kontrolloinnin kautta prosessin lopputuotteiden tasalaatuisuuden varmistamiseen, toisaalta yksittäisen ohjelmistotuotteen verifiointi- ja validointitoimenpiteiden vaatiman informaation tuottamiseen pitkin ohjelmiston elinkaarta.

Ohjelmistomittojen tarjoaman informaation käyttötapa riippuu tarkastelijan suhtautumisesta luotettavuuden arvoon. Luotettavuusarvoiltaan kriittisien ohjelmistojen suhteen kattava mallintaminen on välttämätöntä, ja ohjelmistomitat toimivat osana analyysimateriaalin muodostavaa kokonaisuutta. Jos kattavaa mallintamista ei katsota tarpeelliseksi, ohjelmistomittoja voidaan käyttää ohjelmiston ongelmakohtien tunnistamiseen ja korjaavien toimenpiteiden kohdentamiseen. Muihin laatua ja luotettavuutta varmistaviin menetelmiin nähden mittojen käyttökelpoisuus on suurin ohjelmiston elinkaaren varhaisissa vaiheissa.

5.1 Yleistä

5.1.1 Mittaaminen käsitteenä

Ohjelmiston mittaamisella tarkoitetaan ohjelmiston tai ohjelmistokehitykseen liittyvän ominaisuuden kuvaamista numeerisessa muodossa. Kirjallisuudessa käsitettä käytetään väljähkösti, ja merkitys vaihtelee hieman asiayhteydestä riippuen. Varhaisemmissa julkaisuissa ohjelmiston mittaamisella tarkoitettiin lähinnä lähdekoodista mitattavien staattisten ja sisäisten ominaisuuksien kuvaamista. Myöhemmin ohjelmiston mittaamisen käsite on laajentunut kattamaan koko ohjelmiston elinkaaren ja kehitysympäristön koskien myös subjektiivisia laadun arvottamisen menetelmiä. Ohjelmiston mittaamiseksi voidaan käsittää esimerkiksi seuraavat toimenpiteet:

- Ohjelmistoprojektin kustannusten sekä työmäärän ja ennakointi.
- Tuottavuuden mallintaminen ja mittaaminen.
- Ohjelmistoprosessiin tai tuotteeseen liittyvän tiedon kerääminen.
- Ohjelmistotuotteen laadun tai luotettavuuden mallintaminen ja mittaaminen.

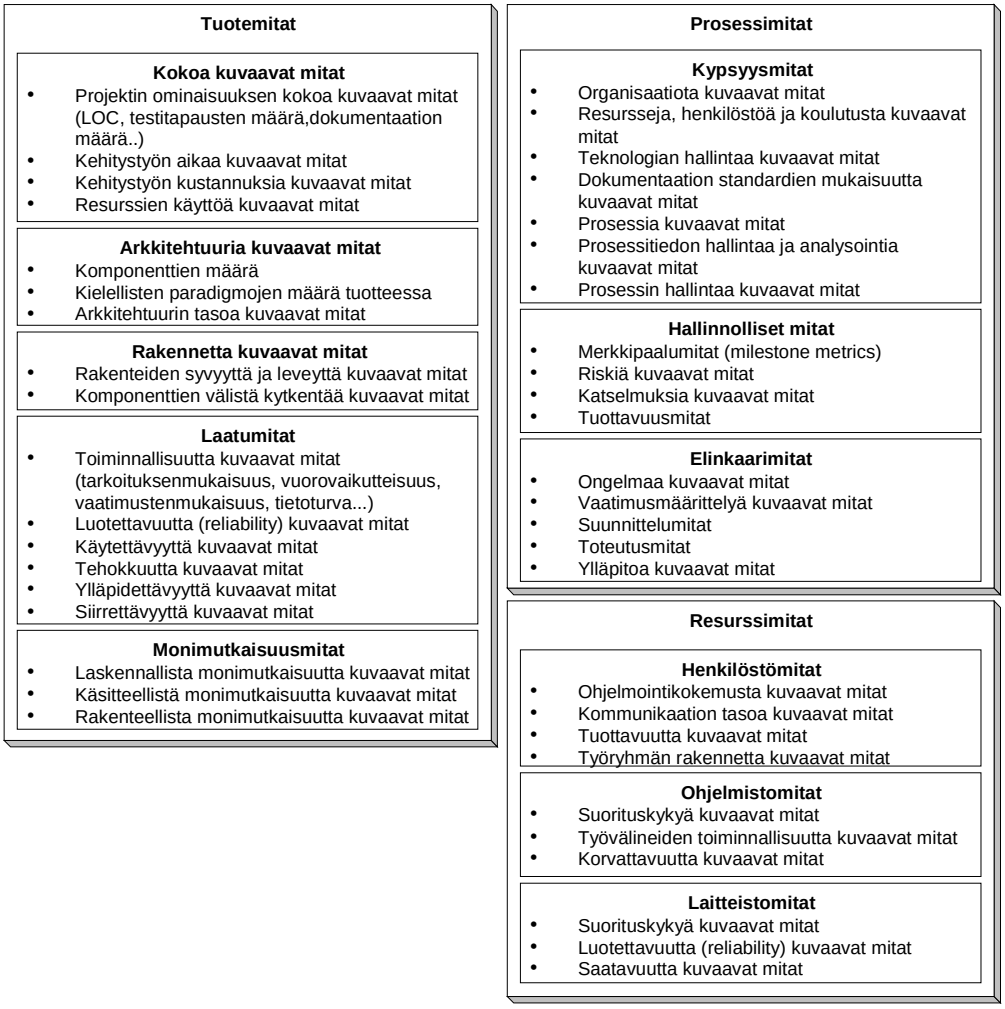
Englanninkielisessä kirjallisuudessa ohjelmiston mittaamisesta käytetään termiä *software measurement* ja ohjelmistomitoista termejä *software measures* tai *software metrics*. Suomen kielessä ohjelmistomitoille ei ole vakiintunut yksittäistä termiä, puhutaan joko *ohjelmistomitoista* tai *ohjelmistomittareista*. Tässä yhteydessä käytetään termiä ohjelmistomitta. Termillä tarkoitetaan numeerista suuretta, joka kuvaa ohjelmistoon tai ohjelmiston kehitysprosessiin liittyvän ominaisuuden astetta (IEEE 610 1990). Termi mittari ilmentää enemmän päätöksenteon apuvälineenä toimivaa osoittajaa tai ilmentäjää (Sadeniemi 1996), josta tässä dokumentissa käytetään nimitystä *indikaattori*.

Käsite metriikka on vakiintunut matemaattisen metrisen avaruuden määritelmän yhteyteen. Suurin osa ohjelmistomitoista ei kuitenkaan toteuta metrisen avaruuden aksioomia. Termi *ohjelmistometriikka* on ohjelmistomitoista puhuttaessa näin harhaanjohtava, ja sen käyttöä tulisi välttää etenkin kvantitatiivisesti orientoituneessa kielenkäytössä.

5.1.2 Ohjelmistomittojen luokittelu

Mitattavan kohteen mukaisesti ohjelmistomitat voidaan jakaa kolmeen ryhmään: prosessi-, tuote- ja resurssimittoihin (Fenton & Pfleeger 1997). Jaottelu on esitetty kuvassa

21. Prosessimitat kuvaavat ohjelmiston kehitysprosessin ominaisuuksia. Prosessimittoihin kuuluvat esimerkiksi ohjelmistoprojektin aikataulua ja kustannuksia kuvaavat mitat.



Kuva 21. Eräs tapa ryhmitellä ohjelmistomittoja (Dumke 1999). Jaottelu ei ole yksiselitteinen, ja yksittäinen mitta voi kuulua myös useampaan ryhmään.

Ohjelmistoartefaktit ovat ohjelmistoprosessin varsinaisen tuotteen eli ohjelmiston muodostavia osia. Artefakteja ovat ohjelmiston lähdekoodi, suorituskelpoiset ohjelmistomoduulit sekä ohjelmistotuotteeseen välittömästi liittyvä dokumentaatio. Tuotemitat ovat mittoja, jotka kuvaavat ohjelmistotuotteen eli eri artefaktien ominaisuuksia. Esimerkiksi dokumentaation tai lähdekoodin kokoa kuvaavat mitat ovat tuotemittoja.

Resurssimitat kuvaavat ohjelmistoprosessin käytössä olevia resursseja, joita ovat mm. käytettävissä oleva henkilöstö, henkilöstön laatu sekä työvälineistö.

Staattiset mitat ovat mittoja, jotka ovat saatavilla ilman ohjelmakoodin suorittamista. Staattisia mittoja ovat esimerkiksi ohjelmiston rakennekaavioista saatavilla olevat rakenteelliset mitat. *Dynaamiset mitat* ovat mittoja, joitten mittaaminen vaatii ohjelman suorittamisen. Dynaamisia mittoja ovat esimerkiksi testituloksista johdetut mitat.

Ohjelmiston ominaisuutta kutsutaan ulkoiseksi, jos ominaisuuden merkityksen voidaan katsoa muuttuvan ohjelmiston ympäristön mukaan. Ulkoisia eli ympäristöriippuvaisia ominaisuuksia ovat mm. laatu ja turvallisuus. Vastaavasti sisäiseksi ominaisuudeksi kutsutaan ohjelmiston ominaisuutta, joka on ohjelmiston ympäristöstä riippumaton. Mitattavan ominaisuuden mukaisesti ohjelmistomittaa kutsutaan sisäiseksi tai ulkoiseksi mitaksi. Esimerkiksi ohjelmiston ohjausvuon rakenne on sisäinen ominaisuus ja sitä kuvaavat mitat sisäisiä mittoja.

Primitiivimitaksi nimitetään mittaa, joka on suoraan mitattavasta kohteesta määritellyllä algoritmilla laskettavissa oleva suure. Primitiivimittoja ovat esimerkiksi tarkastelussa löytyneiden virheiden määrä ja lähdekoodin koodirivien lukumäärä. Primitiivimitoista voidaan matemaattisella kaavalla tai algoritmilla johtaa rakenteellisella abstraktiotasolla ylemmällä olevia, johdettuja mittoja. *Johdettu mitta* on esimerkiksi virheiden määrä per koodirivien lukumäärä. Primitiivimitat ja johdetut mitat tarjoavat harvoin sellaisenaan riittävän selkeää informaatiota päätöksenteon tueksi. Indikaattoreiksi kutsutaan määritellyn mallin avulla primitiivimitoista ja johdetuista mitoista koottuja tunnuslukuja, joita voidaan käyttää päätöksenteon tukena.

5.1.3 Subjektiivisuus mittaamisessa

Ideaalitilanteessa käytettävä mitta on objektiivinen, mittaajasta ja mittauksen kohteen ympäristöstä riippumaton reaalimaailman projektio numeeriseksi luvuksi. Ohjelmistomaailmassa ideaalitilannetta harvoin saavutetaan, ja mittoihin kätkeytyvä subjektiivisuus on pystyttävä huomioimaan etenkin vertailtaessa eri mittausten tuloksia.

Vaikka sisäiset mitat ovat periaatteessa toistettavia ja vertailukelpoisia, voi sisäisenkin mitan mittaus mitasta riippuen sisältää subjektiivisuutta. Esimerkkinä voidaan ajatella määrittelyvaiheessa suoritettavaa toimintopisteanalyysin suorittamista (Function Point Analysis, FPA) (Albrecht 1979). Analyysissä käytettävät primitiivimitat ovat sidoksissa mittaajan kykyyn suorittaa analyysi kattavasti sekä mittaajan käsitykseen siitä, onko ohjelmiston piirre analyysin kannalta relevantti vai ei. Vastaava määrittelyongelma nousee olio-ohjelmoinnissa käytettävän avainluokkien lukumäärän (Number of Key Classes, NKC) suhteen (Lorenz & Kidd 1994). Luokkien jako avainluokkiin ja tukeviin luokkiin ei laajemmissa sovelluksissa ole välttämättä triviaali tehtävä, ja avainluokan määrittelyn väljyydestä johtuen mitta saattaa sisältää subjektiivisuutta.

Subjektiivisuuden muodoksi voidaan luokitella myös käytetyn teknologian sekä sovelluksen tyypin aiheuttama harha. Useat mitat ovat riippuvaisia kielestä sekä kehitysympäristöstä. Esimerkiksi vertailtaessa ohjelmiston toiminnon konekielellä ja korkeamman tason kielellä tehtyä toteutusta mittaamalla koodirivien lukumäärää ovat tulokset radikaalisti erilaiset, vaikka toiminnot käyttäytymiseltään ovatkin identtisiä.

Asiayhteyden mukaan ohjelmistomitoiksi voidaan kutsua myös puhtaasti subjektiivisia mittoja. Tällaisia ovat esimerkiksi asiantuntijoiden kokemukseen ja näppituntumaan perustuva tarkastelukohteen hyvyyden arviointi. Puhtaasti subjektiivisiin arvioihin perustuvat mitat muodostavat kuitenkin oman ohjelmistokehityksen ajattelumaailmasta riippumattoman kognitiivisen tieteenhaaransa.

Nyrkkisääntönä voitaneen ajatella objektiiviseksi mitta, jonka laskenta voidaan suorittaa automaattisesti käyttämällä mitalle määriteltyä algoritmia. Vertailtaessa mittojen tuottamia tuloksia on kuitenkin aina kiinnitettävä huomiota mitattaessa tehtyihin oletuksiin sekä mittaavan algoritmin toimintaan.

5.1.4 Elinkaari

Ohjelmistokehitykseen valitusta elinkaarimallista riippumatta elinkaaresta on tunnistettavissa ainakin seuraavat vaiheet:

- määrittely
- suunnittelu
- toteutus
- testaus
- käyttöönotto ja ylläpito.

Ohjelmiston luotettavuuden kvantitatiivinen arviointi voidaan jakaa luotettavuuden varhaiseksi ennustamiseksi, luotettavuuden ennustamiseksi sekä estimoinniksi sen mukaan, mistä elinkaaren vaiheista tarkasteltavaa tietoa on saatavilla (kuva 22). Estimoinnissa käytettävissä on dynaamista dataa ohjelmiston käyttäytymisestä operatiivisessa ympäristössä. Ennustettaessa käytössä on dynaamista tietoa testaustulosten pohjalta, mitä varhaisen ennustamisen tapauksessa ei ole.

Elinkaaren aktiivisesta vaiheesta ja tarkastelun alla olevan ohjelmistoartefaktin tyypistä riippumatta ohjelmiston laadun ja luotettavuuden kannalta relevantteja mittauksen kohteita ovat seuraavat:

- Artefaktista löytyneiden virheiden lukumäärä.

- Virheiden hakemiseen käytetyt resurssit, esimerkiksi tarkistusten ja katselmointien lukumäärä.

- Virhetiheys eli artefaktista löytyneiden virheiden lukumäärä per tarkasteltavan alueen koko.

- Virheiden jakauma, esimerkiksi elinkaaren vaiheiden tai virheiden tyypin mukaan.

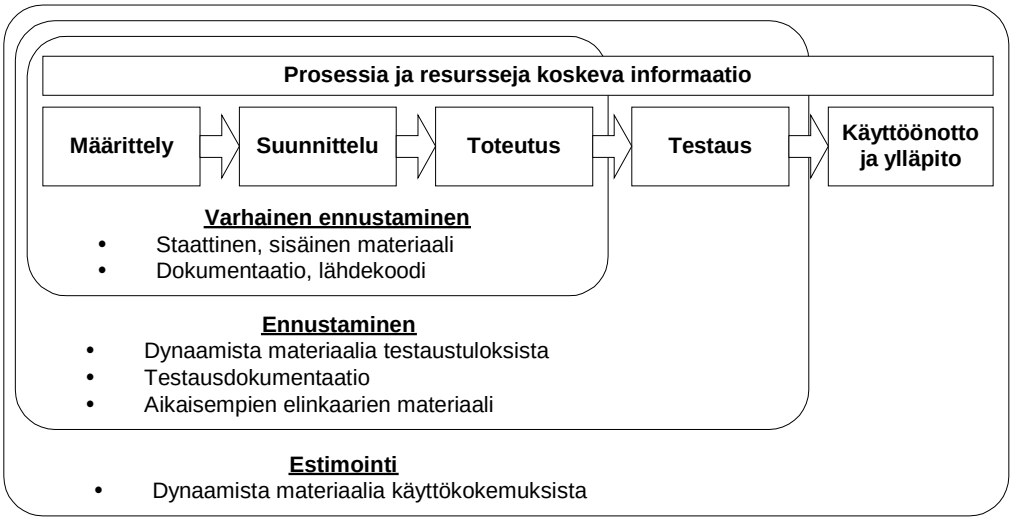
- Artefaktiin myöhemmissä elinkaaren vaiheissa suoritettujen muutosten lukumäärä.

- Artefaktista havaittujen vikojen havaitsemisen ja korjaamisen välinen aika.

- Virheiden istutukseen perustuvat mitat.

Artefaktien laadullisen mittaamisen perusmateriaalia ovat artefakteista löytyneiden virheiden määrä sekä virheiden määrästä johdetut mitat. Ohjelmiston virhetiheyttä pidetään ohjelmiston laatua ilmaisevana *de facto* -standardimittana (Fenton & Pfleeger 1997). Löydettyjen virheiden lukumäärään perustuvia mittoja on kuitenkin tarkasteltava tietyllä kritiikillä: pelkkä löydettyjen virheiden määrä ei kerro ohjelmiston laadusta, vaan sitä on tarkasteltava virheiden etsimiseen käytettyjä resursseja vasten. Jos ohjelmistosta ei ole löydetty yhtään virhettä, on tulos todennäköisemmin riittämättömän tai kokonaan suorittamatta jätetyn virheenetsinnän seuraus. Vastaavasti löydettyjen virheiden suuri määrä voi indikoida yhtä hyvin virheiden etsimisen korkeaa tasoa kuin virhetiheydeltään korkeaa tuotetta (Fenton & Neil 1999).

Ohjelmistoprosessia ja resursseja yleisesti koskeva informaatio on saatavilla mittausta varten koko elinkaaren ajan. Informaation saatavuus on kuitenkin riippuvainen prosessin kypsyydestä (Baumert & McWhinney 1992); Määrittelemätöntä prosessia ei voi mitata. Ohjelmistoprosessista itsestään mitattavia ominaisuuksia ovat esimerkiksi prosessin kypsyys, prosessin aikana suoritettavien yksittäisten toimenpiteiden kuten katselmusten ja tarkastuksien lukumäärä sekä toimenpiteisiin panostettavat resurssit ja panostuksen teho.



Kuva 22. Elinkaaren eri vaiheissa mitattavissa oleva materiaali.

Prosessi- ja resurssi-informaation merkityksellisyys ohjelmiston laadullisessa tarkastelussa lisääntyy, jos varsinaista tuotetta koskevaa informaatiota, lähdekoodia tai dokumentaatiota ei ole syystä tai toisesta riittävästi saatavilla. Ohjelmistoprosessista ja resursseista saatava informaatio on laatuattribuuttien kannalta kuitenkin välillistä. Kypsä ohjelmistoprosessi ei takaa yksittäisen tuotteen laatua *per se*, mutta kypsän ohjelmistoprosessin tuotteet ovat tasalaatuisempia ja todennäköisesti laadultaan parempia verrattuna kypsymättömän prosessin tuotteisiin. Ohjelmistoissa, joissa luotettavuus on kriittinen ominaisuus, luotettavuuden analyysiä ei voida perustaa pelkästään yrityksen prosessin ja prosessin kypsyyteen perustuvan tiedon varaan, vaan tietoa tuotteen käyttäytymisestä tarvitaan. Sen sijaan luotettavuusarvoltaan vaatimattomampien ohjelmistojen osalta pelkkä ohjelmistotoimittajan hallussa oleva sertifikaatti voi olla riittävä välilliseksi informaatioksi hankittavan tuotteen laadusta.

Testaaminen on keskeisin ohjelmiston laadunvarmistuksen keinoista. Staattiset menetelmät tarjoavat arvokasta tietoa luotettavuuden ongelmakohdista varhaisten elinkaaren vaiheiden aikana, mutta varsinaista evidenssiä ohjelmiston luotettavuudesta saadaan vasta dynaamisesta tiedosta testaamisen ja käyttökokemusten myötä. Ohjelmistomittojen käyttö luotettavuuden tarkastelussa on tehokkainta varhaisten elinkaaren vaiheiden aikana, jolloin muuta tietoa ohjelmistosta on vähäisemmässä määrin saatavilla.

5.2 Ohjelmistomittojen tulkinta

Luotettavuusnäkökulmasta tarkasteltuna ohjelmiston mittaamista voidaan lähestyä prosessi- tai tuotehakuisesti. Prosessihakuisen mittaamisen päämäärä on varmistaa ja parantaa ohjelmistoprosessin laatua ja siten välillisesti varmistaa myös prosessin loppu-

tuotteiden tasainen laadukkuus. Tuotehakuisen mittaamisen päämääränä on pyrkiä kehittämään tai arvioimaan yksittäisen ohjelmistotuotteen tai projektin luotettavuutta.

Yksittäisen ohjelmistotuotteen luotettavuutta voidaan analysoida seuraavilla kahdella eri lähestymistavalla (El Emam 2000) riippuen siitä, kuinka suuri varmuus ohjelmiston riittävästä luotettavuudesta halutaan:

- Riskiä karttava lähestymistapa: analyysillä halutaan *varmistaa*, että ohjelmisto on riittävän luotettava ajateltuun sovellusympäristöön.
- Riskejä sietävä lähestymistapa: analyysillä halutaan arvio, onko ohjelmisto liian epäluotettava ajateltuun sovellusympäristöön.

Riskejä karttavassa lähestymistavassa halutaan riittäväksi määritellyllä varmuudella arvio ohjelmiston luotettavuudesta käyttökohteessaan ennen julkaisu- tai käyttöönottotilannetta. Käytännössä riittävän varmuuden saavuttaminen tarkoittaa ohjelmiston toiminnan ja ohjelmiston sovellusympäristön järjestelmäkokonaisuuden käyttäytymisen kattavaa mallintamista. Kattava mallintaminen vaatii huomattavia resursseja sekä asiantuntemusta ja tulee kysymykseen lähinnä luotettavuusarvoltaan kriittisiä ohjelmistoja analysoitaessa.

Mittaamisen kannalta keskeisimmäksi kysymykseksi nousee tällöin luotettavuuden kannalta relevanttien ohjelmistomittojen valinta sekä mittojen tarjoaman informaation yhdistävän mallin löytäminen. On ilmeistä, etteivät pelkät varhaisissa elinkaaren vaiheissa saatavilla olevat mitat tarjoa riittävää tietoutta varsinaisen ohjelmistoprosessin lopputuotteen luotettavuudesta, vaan ohjelmistotuotteen luotettavuutta varmistettaessa on käytettävä myös dynaamista tietoa ohjelmiston käyttäytymisestä. Koska ohjelmiston ulkoisien ominaisuuksien kuvaaminen täsmällisillä ohjelmistomitoilla on hankalaa, on malliin käytännössä yhdistettävä myös subjektiivisen asiantuntija-arvioinnin avulla tuotettuja suureita.

Riskejä sietävässä lähestymistavassa resurssit pyritään käyttämään ohjelmiston epäluotettavuuden osoittamiseen: ohjelmisto tulkitaan luotettavaksi, jos ohjelmistossa ei havaita epäluotettavia piirteitä. Korkeata varmuutta ohjelmiston luotettavuudesta tällä lähestymistavalla ei kuitenkaan saada, vaan epäluotettavuuden olemassaolo on todennäköisempää kuin riskejä karttavassa näkökulmassa. Koska ohjelmiston luotettavuutta ei pyritä osoittamaan, vaan sitä tutkitaan haettujen vastaesimerkkien kautta, on analyysiin tarvittavien resurssien ja asiantuntemuksen määrä luonnollisesti vähäisempi.

Ohjelmistomittojen käyttö on riskejä sietävässä näkökulmassa kustannustehokkaampaa. Empiiristen havaintojen perusteella 80 prosenttia ohjelmiston ongelmista on peräisin vain 20 prosentista ohjelmistomoduuleita ja noin puolet moduuleista ei aiheuta ongel-

mia lainkaan (Boehm & Basili 2001). Ohjelmiston riskialttiit ongelmakohdat ovat usein tunnistettavissa jo elinkaaren varhaisissa vaiheissa, jolloin niiden korjaaminen on kustannuksia ajatellen edullisinta. Ongelmakohtien tunnistamiseen voidaan käyttää joukkoa sisäisiä ja staattisia ohjelmistomittoja, kuten monimutkaisuusmittoja, joiden kerääminen on asianmukaisilla työkaluilla taloudellista (Tian et al. 2001).

Mittauksen valmisteluvaiheessa on kiinnitettävä riittävä huomio siihen, millainen ohjelmiston osa tulkitaan riskialttiiksi. Kun ohjelmiston riskialttiin osan tunnuspiirteet ovat selkeästi määriteltyjä, on mittaustulosten tulkinta suoraviivaista. Riskialttiuden määrittelemisen apuna voidaan käyttää esimerkiksi Goal-Question-Metrics-menetelmää (Basili & Weiss 1984).

Korkeampaan luotettavuuteen pyrittäessä kustannustehokas tapa on käyttää kaksisuuntaista analyysiä. Ohjelmiston analysointiin käytetään ensimmäisessä vaiheessa kustannuksiltaan taloudellisempaa riskejä sietävää lähestymistapaa. Analyysin myötä ohjelmistoa voidaan kehittää edelleen, kunnes riskejä sietävästä tarkastelutavasta ei enää saada uutta informaatiota ja lisävarmuuden saamiseksi on suoritettava kattava luotettavuusanalyysi.

5.3 Mittojen käyttö luotettavuuden analyysissä – State of the Art

Keskeisimmät kysymykset käytettäessä ohjelmistomittoja luotettavuuden arvioinnin apuna ovat:

- Mitkä ohjelmistomitoista ovat sopivia informaation lähteitä kuvattaessa ohjelmiston luotettavuutta?
- Millä mallilla primitiivi- ja sekundaarimittojen tuottama informaatio yhdistetään luotettavuutta ilmaisevaksi indikaattoriksi?

Lukuisista yrityksistä huolimatta yksittäistä luotettavuutta kuvaavaa mittaa ei ole pystytty identifioimaan. Mm. ohjelmiston fyysistä kokoa sekä rakenteellisen monimutkaisuuden eri piirteitä on ehdotettu luotettavuuden selittäjiksi. Luotettavuus on kuitenkin ulkoinen ominaisuus, jonka kokonaisvaltainen määritteleminen pelkästään sisäisten ominaisuuksien ja mittojen perusteella on käytännössä mahdotonta.

Monimutkaisuus on ohjelmiston luotettavuuden kannalta merkittävin yksittäinen tekijä (Zhang & Pham 2000). Kehitetyt monimutkaisuusmitat eivät kuitenkaan kuvaa monimutkaisuutta kokonaisuudessaan, vaan projisoivat monimutkaisuuden eri osia mitattaviksi suureiksi. Ohjelmiston luotettavuuden analysoinnin kannalta monimutkaisuusmitat

toimivat epäterveiden oireiden ilmentäjinä, mutta pelkästään niiden perusteella ei voida tehdä johtopäätöksiä varsinaisesta luotettavuudesta.

Nykytietämyksen nojalla luotettavuuden kuvaamiseen on käytettävä useita mittoja, joiden informaatiota yhdistämällä voidaan tuottaa arvio luotettavuudesta. Yksimielisyyttä ohjelmiston luotettavuuden kannalta relevanteista mitoista ei ole olemassa, mutta asiantuntijalausuntojen perusteella on koottu ohjenuoraksi sopiva standardi IEEE 982.1 (1998), jossa esitellään 39 mittaa luotettavan ohjelmiston valmistamisen ja tunnistamisen avuksi (taulukko 5). Osittain em. standardia pohjanaan käyttäen Marylandin yliopiston tutkijaryhmä on eri asiantuntijaryhmiin ulotettujen haastattelujen perusteella arvottanut mittojen relevanttiutta sekä luotettavuuteen että mittauksen helppouteen ja kustannuksiin nähden (Smidts & Li 2000). Tuloksissa arvostetuimmiksi selvisivät perinteiset virheiden ja virhetoimintojen lukumäärään perustuvat mitat, kun taas teoreettisempien mittojen tarjoamaa informaatiota ei katsottu niin tärkeäksi. Arvostus riippui myös vahvasti aktiivisesta elinkaaren vaiheesta, koska mitä myöhäisempi elinkaaren vaihe on, sitä tarkempaa tietoa ohjelmiston olemuksesta on saatavilla. Tutkimus keskittyi pääasiallisesti lauserakenteisiin ohjelmointikieliin. Oliokielille standardia vastaavaa ohjenuoraa ei toistaiseksi ole olemassa, mutta käyttökelpoisia suosituslistoja mittojen käytölle on kirjallisuudessa saatavilla, mm. (Lorenz & Kidd 1994).

Alan tutkimuksen volyymista huolimatta yleistä ohjelmiston luotettavuutta kuvaavaa mallia ei ole pystytty kehittämään. Ohjelmistot ovat yksilöitä, joiden käyttäytyminen luotettavuuden kannalta painottuu eri tavalla. Käyttökelpoisella tasolla oleva ohjelmistojärjestelmän luotettavuusmalli on siten aina yksilöllinen, mikä vaikeuttaa aikaisempien kokemusten hyväksikäyttöä ja aiheuttaa väistämättä kustannuksia räätälöintiin vaadittavan työn määränä.

Varteenotettavana vaihtoehtona luotettavuusmallin rakentamiseksi on esitetty bayesilaisten uskomusverkkojen hyväksikäyttöä (Fenton & Neil 1999). BBN:t (Bayes Belief Nets) ovat sattumansolmuista ja solmujen välisistä vuorovaikutuksista koostuvia graafeja. Bayesilaisten uskomusverkkojen datana voidaan käyttää monipuolisesti tietoa sekä ohjelmistoprosessista että -tuotteesta. Uskomusverkkojen muita hyviä puolia on mm. niiden kyky käsitellä syy-seuraussuhteita, epävarmuutta sekä epätäydellistä informaatiota ja subjektiivista tietoa.

Taulukko 5. Ohjelmistomittoja luotettavan ohjelmiston tuottamiseksi (IEEE 982.1 1998).

MEASURE CATEGORIES	PRODUCT						PROCESS			LIFECYCLE			
	Errors, Faults, Failures	MTTF, Failure rate	Reliability Growth & Pro-	Remaining Product Faults	Completeness & Consis-	Complexity	Management Control	Coverage	Risk, Benefit, Cost evalua-	Requirement	Design	Implementation	Testing & Operational
Fault Density	X									X	X	X	X
Defect Density (Design)	X										X	X	X
Cumulative Failure Profile	X												X
Fault-Days Number	X						X			X	X	X	X
Functional Test Coverage					X			X	X				X
Modular Test Coverage					X			X	X				X
Cause and Effect Graphing					X			X		X	X	X	X
Requirements Traceability	X				X			X			X	X	X
Defect Indices	X						X			X	X	X	X
Error Distribution(s)							X			X	X	X	X
Software Maturity Index			X						X				X
Man Hours per Major Defect Detected							X		X		X	X	X
Number of conflicting Requirements	X				X			X		X	X	X	X
Number of Entries and Exits per Module						X					X	X	X
Software Science Measures (Halstead)				X		X			X			X	X
Graph-Theoretic Complexity for Architecture						X					X	X	X
Cyclomatic Complexity					X	X					X	X	X
Minimal Unit Test Case Determination					X	X					X	X	X
Run Reliability			X						X				X
Design Structure						X			X		X	X	X
Mean Time to Discover the Next K Faults							X		X				X
Software Purity Level			X						X				X
Estimated Number of Faults Remaining (Error Seeding)				X						X	X	X	X
Requirements Compliance	X				X			X		X	X	X	X
Test Coverage					X			X					X
Data or Information Flow Complexity						X					X	X	X
Reliability Growth Function			X										X
Residual Fault Count				X					X				X
Failure Analysis Using Elapsed Time			X	X									X
Testing Sufficiency			X					X					X
Mean Time To Failure		X	X										X
Failure Rate		X							X				X
Software Documentation and Source Listings					X			X				X	X
RELY (Required Software Reliability)								X	X	X	X	X	X
Software Release Readiness									X				X
Completeness								X		X	X	X	X
Test Accuracy				X	X			X					X
System Performance Reliability			X										X
Independent Process Reliability			X										X
Combined Hardware and Software (System) Operational Availability			X										X

Prosessin- ja projektinhallinnallisessa mielessä ohjelmistomittojen yhteys taloudellisiin tekijöihin on suhteellisen selvä; yleispäteviä kustannusmalleja on kehitetty lukuisia (Fenton & Pfleeger 1997), ja niiden hyödyllisyydestä on selkeätä evidenssiä. Kuitenkin esimerkiksi Rubin Systemsin keräämästä datasta ilmenee että vuonna 1995 mittausohjelman käynnistäneistä yrityksistä neljä viidestä katsoi mittausohjelmansa käyttöönoton

epäonnistuneen, ja epäonnistujien määrä on tutkimuksen mukaan kasvusuunnassa (Pitts 1997). Koska ohjelmistomittojen ja luotettavuuden suhde ei ole selkeä, hyödyn saaminen mittausdatasta luotettavuuden kattavan mallintamisen ja jatkuvan parantamisen kannalta on sekä ohjelmiston valmistajan että tarkastelijan näkökulmasta vaativa tehtävä.

Mittaaminen on prosessi, joka vaatii mittaajalta riittävät resurssit ja riittävän asiantuntemuksen sekä ohjelmistokehityksestä että ohjelmistosta itsestään (Baumert & McWhinney 1992, Schneidewind 2001). Prosessin kypsyys asettaa rajoituksia mitattavan materiaalin kannalta. Keskeisin tekijä ohjelmiston ja ohjelmistoprosessin mittaamisen onnistumisen kannalta on kuitenkin tavoitehakuisuus. Mitattavan tiedon tulee tuottaa arvoa, ja mitattavan tiedon tulee olla luonteeltaan helposti hyväksikäytettävissä määriteltyjä tavoitteita varten (Goethert & Hayes 2001, Niessink & Vliet 2001).

Mittaamisen aiheuttamia kustannuksia ja resurssien tarvetta voidaan pienentää automatisoimalla mittausprosessiin liittyviä toimenpiteitä. Erilaisia ohjelmiston mittaamiseen soveltuvia työkaluja on markkinoilta saatavilla, joskin käyttöönoton kynnystä on nostanut sekä hinta että huono yhteensopivuus muiden käytössä olevien työvälineiden kanssa. Tilanne on kuitenkin paranemassa, mikä näkyy esimerkiksi ohjelmointiympäristöjen kehittymisenä integroitaessa niihin erilaisia mittausominaisuuksia.

6. Johtopäätökset

Väitetään, että ohjelmiston prosessimalleja ja -menetelmiä kehitetään liikaa, kun vanhatkaan eivät mene kaupaksi. CMMI, SPICE ja vastaavat kypsyys- ja kyvykkyysmallit ovat liian raskaita ja vaativia. Ehkä vanhat menettelyt ovatkin liian painottuneita perinteiseen laitteistoperäiseen ajatteluun, kuten on laita ohjelmiston luotettavuuden malleissa ja menetelmissä. Toisen väittämän mukaan vanhaan ajatteluun tulisi tehdä selkeä pesäero. Tähän pyritään uusilla menetelmillä, kuten eXtreme Programming, Rapid Development ja Unified Modeling Language. Tässä julkaisussa tarkasteltiin ohjelmiston luotettavuuden pesäeroa laitteistoperäiseen ajatteluun seuraavien kolmen teeman välityksellä: 1) Miksi hyvät menetelmät eivät mene kaupaksi? 2) Automaattinen testaaminen luotettavuuden ilmaisijana. 3) Ohjelmiston virhemekanismit.

Uusia suuntauksia yritettäessä kannattaa em. ensimmäisen teeman ajatus olla mielessä. Alusta lähtien on otettava käyttäjän (so. ostajan) todelliset tarpeet ja ympäristö huomioon eikä vain kuvitella, mitä ne mahdollisesti ovat tai mikä olisi käyttäjälle valmistajan ja kehittäjän mielestä hyväksi. Epäonnistumisia on sattunut monella ohjelmiston luotettavuusprosesseja sivuavilla aloilla:

- Formaalit menetelmät eivät ole saavuttaneet lähestulkoonkaan sitä suosiota mitä niille on povattu jo pari vuosikymmentä.

- Monet ohjelmistoprosessien arviointiin ja parantamiseen tähtäävät ohjeet, kuten CMMI ja SPICE, ovat laajassa käytössä, mutta ne paisuvat paisumistaan ohjelmistojärjestelmien laajentuessa ja monimutkaistuessa.

- Metriikat eivät ole yleisessä käytössä, vaikka niitä on kehitetty ja tutkittu yhtä paljon kuin muuta kehitysprosesseja yhteensä.

- Kehittyneet testausmenetelmät koskevat lähinnä suuria ohjelmistotaloja, pienissä ja keskisuurissa ei yleensä ole merkittävää testauskulttuuria.

Formaalien menetelmien käyttöönottoon on haettu apua koulutuksesta, mutta ohjelmistotekniikalle matemaattinen lähestymistapa on todettu vieraaksi. Matemaattiset menetelmät mielletään ylimääräisiksi apuneuvoiksi, joihin ei olla valmiita kouluttautumaan todellisen ohjelmointityön sijasta.

Ylimääräisyys todellisen toiminnan sijasta on syynä moneen muuhunkin käyttämättömyyteen. Ohjelmiston kehitysstandardit arviointimalleineen ovat yksiä niistä, vaikka laatuajattelulla onkin sijansa ohjelmiston kehittäjien keskuudessa.

Ohjelmiston kypsyyden ja kyvykkyyden mallien yleistyminen käyttäjien keskuudessa viittaa tarpeeseen ja mallien jatkuva kehittyminen viittaa uusien selkeytettyjen mallien kehitystarpeeseen. Tulevaisuudessa tarvitaan "yleismalleja" sektorikohtaiseen ohjelmistoprosessien, tuotteiden ja projektien mittaukseen.

Malleja ohjelmiston mittaamiseksi on kehitetty paljon. Mallien käyttämättömyydessä ei ole kyse mittareiden puuttumisesta, vaan yrityksen ja projektitiimien mittauskulttuurin puuttumisesta. Mittaamisesta ja kehittyneistä mittareista ei kuitenkaan ole apua, jos sopivaa mittauskulttuuria ei ole ensin perustettu. Se perustetaan kouluttamalla tiimit ohjelmiston mittaamiseen käyttämällä aluksi yksinkertaisia mittaustyökaluja sekä selkeyttämällä yhteyksiä mittareiden ja liiketoiminnan tavoitteiden välille. Mittaustietojen keruun pitäisi kuulua oleellisena osana ohjelmiston kehitysprosessiin, ei ylimääräisenä tekijänä, mikä on omiaan vähentämään kiinnostusta mittareiden käyttöönottoon.

Tietokoneet ovat vallanneet useita manuaalisesti suoritettuja tehtäviä. Elektroniikan valmistuksessa hyödynnetään testejä mikroelektroniikasta piirikorttien koontaan. Tehtaissa tietokoneet ohjaavat ja valvovat laitteiden valmistusta ja valmistuskustannukset ovat vähentyneet huomattavasti. Koska automatisoinnista on tullut menestystekijä monella alalla, myös ohjelmiston testaamista automatisoidaan. Sopii kuitenkin kysyä, kuulostaako järkevältä testata kriittistä tietokoneohjelmaa toisella tietokoneohjelmalla. Mikähän on testausohjelman kriittisyys tässä tapauksessa?

Testaaminen on työlästä ja ehkä pitkästyttävääkin. Ohjelmiston testaaminen automaattisella testiohjelmalla vähentää ihmisen tekemiä testausvirheitä, sillä ohjelma ei vahingossa jätä väliin testitapauksia tai tulosta raportteja epätarkasti. Testausprosessin kelpoisuuden arviointia edesauttavat automaattiset tietokantatallennukset, joista koottua tilastoa voidaan myös hyödyntää jo kehitysprosessin aikana. Toisaalta manuaalisissa testeissä testitapausten valinnat ovat aina jossakin määrin satunnaisia, mikä tekee virheiden etsimisen laajaskaalaisemmaksi ja vaihtelevammaksi kuin automaattisissa testauksissa. Koska ohjelman vaihtelevuus suorituksen aikana on tavallisesti olematon, joitakin manuaalitesteissä löydettyjä virheitä ei ehkä havaita automaattisin testein. Automaattinen ohjelmistotestaus ei koskaan täysin korvaa manuaalitestaamista.

Ohjelmiston automaattiseen testaamiseen siirrytään yleisimmin kahdesta syystä: joko tuote on aivan liian monimutkainen manuaalisesti testattavaksi tai testausaikaa halutaan huomattavasti lyhentää. Lääkintälaitteiden valmistuksessa on vielä kolmaskin merkittävä syy automaatioon siirtymisessä: yhdysvaltalaisen lääkintäviranomaisen FDA:n vaatimusten täyttyminen saadaan helpommin toteutetuksi automaattisella testaamisella ja dokumentoinnilla.

Kun ison vaivannäön jälkeen testiautomaatio saadaan kohdalleen, tuotteistakin tulee aikaisempia turvallisempia ja luotettavampia. Automatisoimalla mahdollisesti pystytään myös vähentämään ohjelmiston kehityskustannuksia ja parantamaan ohjelmiston laatua ja leikkaamaan markkinoille pääsyn aikaa. Automatisoinnin kohteina ovat staattiset analyysit, testaukset ja formaalit menetelmät, jotka ovat olleet erillisiä tekniikoita. Aivan viime aikoina niitä on pyritty integroimaan siten, että menetelmien väliset rajaviivat ovat alkaneet vähentyä. Käyttäjien ja soveltajien tarpeet ovat tärkeitä ohjelmiston valmistajille. Vaateita integroida valmistajan arviointitekniikoita käyttäjän ja soveltajan kehitysprosesseihin on alkanut ilmaantua. Kuitenkin on monia avoimia kysymyksiä automaattisten testausten ja verifiointien välillä sekä staattisten analyysien hyödyntämisessä testauksissa ja verifioinneissa. Voidaan luetella ainakin seuraavat merkittävät tutkimuskohteet:

- staattiset analyysitekniikat supistettaessa ohjelmaa
- staattiset analyysitekniikat paljastettaessa virheitä
- automaattinen testaaminen ja testitapausten generointi
- automaattinen regressiotestaaminen
- verifiointi- ja testitekniikoiden integroiminen
- analyysitekniikat tuettaessa verifiointitestaamista
- teolliset koejärjestelyt
- teknologian siirto-ongelmat.

Mikä olisi sitten testaamattomuuden hinta? Yhdysvalloissa on käyty oikeutta vastaukseksi. Oikeusprosesseissa vastapuolet ovat vedonneet etupäässä IEEE:n standardeihin, mutta myös eurooppalaisia on kelpuutettu osoitettaessa niitä alan state-of-artiksi.

Uusissa suuntauksissa pitää kuitenkin aina olla selkeästi tukevalla pohjalla: ohjelmisto on aina laitteistossa. Onko siis järkevää erottaa ohjelmiston luotettavuuskäsitteitä jyrkästi järjestelmän luotettavuuskäsitteistä? Tällaiseen mahdolliseen järjettömyyteen ei ohjelmiston luotettavuuden arviointi ole vielä kovin pitkälle ennättänyt. On Bayesin uskomusverkkomenetelmä, joka tukeutuu siihen ajatukseen, että koodissa on aina virhe, tai jos ei ole, niin sitä ei tiedetä. Asiantuntijoiden arviot ovat tämän hetken parasta antia pääteltäessä mahdollisen virheen olemassaoloa ja etenemistä virhetoiminnoksi.

Sarjaa ”virheestä sisäiseen virhetilanteeseen ja ulkoiseen virhetoimintaan” kutsuttiin tässä julkaisussa virhemekanismiksi. Virhemekanismien ymmärtämiseen kohdistuvat kaikki testaus-, analyysi- ja suunnittelumenetelmät. Yleisiä virhetopologioita on kehitetty, mutta niillä ei ole ollut laaja käyttöä. Geneeristen luokitteluiden sijaan tulisikin kehittää sovellusalakohtaista teoreettista mallinnusta virhemekanismeista. Tutkimus- ja kehitystuloksista olisi apua pääteltäessä menetelmien virheiden paljastamisen, virhesietoisuuden ja toipumisen kattavuutta. Teoreettiselta pohjalta on myös hyvä ponnistaa kehitettäessä sopivia menetelmien valintaprosesseja tietyn kokoiselle ja tietyn kulttuurin omaavalle yritykselle.

Virhemekanismien täsmällinen ymmärtäminen edesauttaisi testaus- ja analyysisuunnittelussa kattavuusolettamusten teoreettista laatimista. Jos nämä olettamukset ovat tiedossa ennen kehityssuunnittelua, ohjelmiston luotettavuuden suunnittelussa ja erityisesti arvioinnissa oltaisiin hyvin tukevalla pohjalla, joka johtaisi kustannustehokkaaseen kehitys- ja kelpoistusprojektiin. Kattavuus on kuitenkin paljon käytetty mutta huonosti tunnettu termi.

Ohjelmistomittoja on perinteisesti käytetty ohjelmistoprosessin ja -projektin hallinnan apuna. Mittojen käyttöä ohjelmistojen luotettavuuden arvioimiseksi sekä parantamiseksi on ehdotettu etenkin, koska mitat tarjoavat informaatiota ohjelmistosta jo elinkaaren varhaisissa vaiheissa ennen suorituskelpoisen ohjelmistokoodin olemassaoloa.

Ohjelmistomittojen tarjoamaa informaatiota voidaan käyttää sekä taloudellisemmin ohjelmiston riskialttiiden osien tunnistamiseen että myös kattavaan luotettavuuden analysoimiseen. Kattavan luotettavuuden analyysin problematiikka redusoituu kahteen kysymykseen: mitkä ovat relevantit mitat ilmaisemaan ohjelmiston luotettavuutta sekä millä mallilla alemman tason ohjelmistomittojen tuottama informaatio yhdistetään intuitiivisesti käsitettävissä olevaksi luotettavuutta indikoivaksi arvoksi.

Yksittäistä luotettavuutta kuvaavaa ohjelmistomittaa ei ole olemassa, vaan luotettavuuden kannalta keskeinen informaatio on hankittava yhdistämällä useiden eri mittojen tuloksia. Ohjelmistot ovat pitkälle yksilöllisiä, ja yleispätevää mittojen tuottamaa informaatiota yhdistävää luotettavuusmallia ei ole pystytty toistaiseksi kehittämään, mutta esimerkiksi bayesilaisten uskomusverkkojen käyttöä tarkoitukseen tutkitaan.

Lähdeluettelo

Albrecht, A.J. 1979. Measuring application development. Proceedings of IBM applications development. Joint SHARE/GUIDE Symposium, Monterey CA. S. 83–92.

Arlat, J., Aguera, M., Crouzet, Y., Fabre, J.-C., Martins, E., Powell, D. 1990. Experimental evaluation of the fault tolerance of an atomic musticast protocol. IEEE Transaction on Reliability. Vol. 39, no. 4, s. 455–467.

Arlat, J., Costes, A., Crouzet, Laprie, J.-C., Powell, D. 1993. Fault injection and dependability evaluation of fault-tolerant systems. IEEE Transaction on Computers. Vol. 42, no. 8, s. 913–923.

Arlat, J., Kanoun, K., Madeira, H., Busquets, J. V., Jarboui, T., Johansson, A., Lindström, R. 2001. Dependability benchmarking: state of the art. Report no. IST-2000-25425. 63 s.

Avižienis, A., Laprie, J.-L., Randell, B. 2001. Fundamental concepts of dependability. LAAS Report no. 01–145. 21 s.

Basili, V., Weiss, D.A. 1984. Methodology for collecting valid software engineering data. IEEE Transactions on Software Engineering, Vol. 10, no. 6, s. 728–738.

Baumert, J., McWhinney, M. 1992. Software measures and the Capability Maturity Model. Software Engineering Institute Technical Report, CMU/SEI-92-TR-25, ESC-TR-92-0.

Beizer, B. 1990. Bug taxonomy and statistics. Teoksessa: Software Testing Techniques. Van Nostrand Reinhold. 503 s. ISBN 0-442-20672-0.

Benso, A., Rebaudengo, M., Reorda, M.S. 1999. FlexFi: a flexible Fault Injection environment for microprocessor-based systems. Teoksessa: Felici, M. et al. (Ed.) SAFECOMP 1999: 18th International Conference on Computer Safety, Reliability and Security, (Lecture Notes in Computer Science, Springer Verlag). S. 323–335.

Boehm, B., Basili, V. 2001. Software defect reduction top 10 list. IEEE Computer. Vol. 34, no. 1, s. 135–137.

Bouricius, W.G., Carter, W.C., Schneider, P.R. 1969. Reliability modeling techniques for self-repairing computer systems. Proceedings of the 24th ACM Annual Conference, March 1969. S. 295–309.

Chau, P. 1996. An empirical investigation of factors affecting the acceptance of CASE by system developers. Information and Management, Vol. 30, s. 269–280.

Chillarege, R. et al. 1992. Orthogonal defect classification - a concept for in process measurements. IEEE Transanction on Software Engineering, Vol. 18, no. 11, s. 943–956.

Christmansson, J., Chillerege, R. 1996. Generation of an error set that emulates software faults - based on field data. 26th International Syposium on Fault-tolerant Computing. Sendai, Japan, June 1996. S. 304–313.

Clark, J.A., Pradhan, D.K. 1995. Fault injection – a method for validating computer-system dependability. IEEE Computer, Vol. 28, no. 6, s. 47–56.

Colombo, A.G., Keller, A.Z., editors. 1987. Reliability modeling and applications: Proceedings of the ISPRA Course. ISPRA, Italy, D. Reidel Publishing Company, s. 1–32.

Daiqui, S. 1996. Application of an integrated, modular, metric based system and software test concept. Final report. ATECON-project, ESSI Number 10464.

DeMillo, R. A., Lipton, R. J., Sayward, F. G. 1978. Hints on test data selection: hints for the practicing programmer. Computer, Vol. 11, no. 4, s. 34–41.

Dhillon, B.S., Anude, O.C. 1994. Common-cause failure analysis of a redundant system with repairable units. International Journal of System Science, Vol. 25, no. 3, s. 527–540.

Dumke, R. 1999. Software metrics classification. University of Magdeburg, Software Measurement Laboratory. Saatavilla: http://ivs.cs.uni-magdeburg.de/sw-eng/us/index.shtml

Dustin, E., Rashka, J., Paul, J. 1999. Automated software testing. introduction, management, and performance. 1. p. New York: Addison Wesley. 575 s. ISBN 0-201-43287-0.

El Emam, K. 2000. A methodology for validating software product metrics. National Research Counsil of Canada, CNRC Technical Report NRC 44142. 39 s.

El Emam, K., Garro, I. 2000. Estimating the extent of standards use: the case of ISO/IEC 15504. The Journal of Systems and Software. Vol. 53, s. 137–143.

EN-IEC 61508, standard. 2001. Functional safety of programmable electronic systems: International Electrotechnical Commission. Parts: 1–7.

ESA, European Space Agency. 1996. Ariane 5 flight 501 failure. Ariane 501 Inguiry Board report. Paris, 19. July 1996, s. 14 ja 46.
Saatavilla: http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf.

Fabre, J.-C., Salles, F., Rodríguez, M., Arlat, J. 1999. Assessment of COTS microkernels by fault injection. Teoksessa: Avizienis, A., Kopetz, H., Laprie, J.-C. Dependable Computing and Fault-Tolerant Systems. IEEE CS Press. S. 25–44.

Fayad, M., Tsai, W., Fulghum, M. 1996. Transition to object-oriented software development. Communication of the ACM, Vol. 39, no. 2, s. 109–121.

Fenton, N., Neil, M. 1999. Software metrics: successes, failures and new directions. Journal of Systems and Software, Vol. 47, no. 2–3, s. 149–157.

Fenton, N., Pfleeger, S. 1997. Software metrics: A rigorous & practical approach. International Thomson Computer Press, London. S. 638. ISBN 0-534-95600-9.

Fewster, M. 2001. Common mistakes in test automation. Software Test Automation Conference & EXPO, San Jose, CA. March 5–8, 2001. 8 s.

Fujiwara, H., Shimono, T. 1983. On the acceleration of test generation algorithms. IEEE Transactions on Computers, Vol. 32, no. 12, s. 1137–1144.

Gaburro, P. 1996. automated testing for the man machine interface of a training simulator. SIMTEST-project, ESSI Number 10824.

Gangloff, W.C. 1975. Common-Mode Failure analysis. IEEE Transactions on Power Apparatus and Systems, Vol. PAS-94, no 1, s. 27–30.

Goel, P. 1981. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. IEEE Transaction on Computers, Vol. 30, no. 3, s. 215–222.

Goethert,W., Hayes, W. 2001. Experiences in implementing measurement programs. Software Engineering Institute. CMU/SEI-2001-TN-026.

Gokhale, S.S., Trivedi, K.S. 1998. Dependancy characteriszation in path-based approach to architecture-based software reliability prediction. Proceedings of the Workshop on Application-Specific Software Engineering and Technology (ASSET’98). 4 s.

Gormley, S., Keating, F., O'Sullivan, F. 1995. Automated life-cycle approach to software testing in the finance & insurance sector. ALCAST-project, ESSI Number 10146. Saatavissa http://www.tekesetx.net/.

Gray, J. 1990. A census of tandem system availability between 1985 and 1990. IEEE Transaction on Reliability, Vol. 39, no. 4, s. 409–432.

Green, G.C., Hevner, A.R. 2000. The successful diffusio of innovations: guidance for software development organisations . IEEE Software, Vol. 17, no. 6, s. 96–103.

Hamlet, D., Mason, D., Woit, D. 2001. Theory of software reliability based on components. Proceedings of the 23rd International Conference on Software Engineering ICSE'2001. 10 s.

Harju, H. 2000. Ohjelmiston luotettavuuden kvalitatiivinen arviointi. Espoo: Valtion teknillinen tutkimuskeskus. VTT Tiedotteita 2066. 111 s. ISBN 951-38-5766-2. Saatavilla: http://www.inf.vtt.fi/pdf.

Harju, H. 2002. Kustannustehokas ohjelmiston luotettavuuden suunnittelu ja arviointi. Osa 1. VTT Tiedotteita 2151. 114 s. + liitt. 15 s. ISBN 951-38-6062-0. Saatavilla: http://www.inf.vtt.fi/pdf.

Hiller, M., Jhumka, A., Suri, N. 2001. An approach for analysing the propagation of data errors in software. Proceedings of International Conference on Dependable Systems and Networks (DCN-2001), Göteborg, Sweden. IEEE CS Press. S. 161–170.

Holloway, C., Butler, R. 1996. Impediments to industrial use of formal methods. computer, Vol. 29, no. 4, s. 25–26.

IEC 61713, standard. 2000. Software dependability through the software life-cycle processes – application guide. International Electrotechnical Commission. 67 s.

IEEE 1061, standard. 1998. IEE standard for a software quality metrics methodology. 20 s. ISBN 0-7381-1059-6.

IEEE 610, standard. 1990. IEEE standard computer dictionary: a compilation of ieee standard computer glossaries. Institute of Electrical and Electronics Engineers. New York, NY, 18 January 1991.

IEEE 982.1, standard. 1998. Standard dictionary of measures to produce reliable software. Institute of Electrical and Electronics Engineers. New York, NY. 36 s.

Iivari, J. 1996. Why are CASE tools not used. Communication of ACM, Vol. 39, no. 10, s. 94–103.

ISO 9126, standard. 1991. Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their use, International Standards Organisation (ISO). Geneva.

Jézequel, J.-M., Meyer, B. 1997. Design by contract: the lessons of Ariane. IEEE Computer, Vol. 30, no. 2, s. 129–130.

Kaner, C. 1996. Software negligence and testing coverage. 15 s. Saatavilla internetistä: http://www.kaner.com/coverage.htm.

Kaufman, L.M., Johnson, B.W. 1999. Embedded digital system reliability and safety analysis. U.S. Nuclear Regulatory Research, NUREG/GR-0020, UVA Technical Report 991221.0. 75 s.

Land, K. 1997. Results of the IEEE survey of software engineering standards users. 3rd IEEE International Symposium on Software Engineering Standards (ISESS '97). Walnut Creek, CA. June 1–6, 1997.

Land, K. 1999. Second software engineering standards users' survey. Fourth IEEE International Symposium on Software Engineering Standards. Curitiba, Brazil. May 17–21, 1999.

Laprie, J.-C. 1998. Dependability: basic concepts and terminology. Teoksessa: Dependability Handbook. (Toulouse): Laboratory for Dependability Engineering. 290 s. (LAAS Report no. 98-346.)

Leveson, N.G, Harvay, N.P.E. 1983. Analysing software safety. IEEE Transaction on Software Engineering, Sept. 1983, s. 569–579.

Leveson, N.G. 1986. Software safety: Why, what and how? ACM Computing Surveys, Vol. 18, no. 2, s. 125–163.

Leveson, N.G. 1995. Safeware: System Safety and Computers. A guide to preventing accidents and losses caused by technology. 1. p. New York: Addison-Wesley. 680 s. ISBN 0-201-11972-2.

Lorenz, M., Kidd, J. 1994. Object Oriented Software Metrics. Prentice Hall, New York. ISBN 0-13-179292-X.

Lyu, M., ed.1996. Handbook of Software Reliability Engineering. McGraw-Hill, New York. ISBN 0-07-039400-8.

Michael, C.C., Jones, R.C. 1997. On the uniformity of error propagation in software. In Proceedings of the 12th Annual Conference on Computer Assurance, COMPASS ’97, Gaithersburg, MD. S. 68–69.

Morell, L.J. 1988. Theoretical Insights into Fault-based testing. In: Proceedings, 2nd Workshop on Software Testing, Verifcation, and Analysis. S. 45–62.

Morell, L.J., Murrill, B., Rand, R. 1997. Perturbation analysis of computer programs. Proceedings of the 12th Annual Conference on Computer Assurance, COMPASS’97. S. 77–78.

Musa, J.D. 1993. Operational Profiles in Software Reliability Engineering. IEEE Software, Vol. 10, no. 2, s. 14–32.

Musa, J.D. 1998. Software Reliability Engineering. New York: McGraw-Hill. 391 s. ISBN 0-07-913271-5.

Nelson, V.P., Caroll, B.D. 1982. Tutorial: Fault-Tolerant Computing, IEEE Computer Society Press.

Niemi, E. 2001. Sulautettujen järjestelmien tuotekehityksen testauksen automatisointi ja etäkäyttö. PROTEST-projekti. OAMK Raahen tietokonealan yksikkö. ETX-Järjestelmät ja ohjelmistot tulosseminaari 26.4.2001.

Niessink, F., Vliet, H. 2001. Measurement program success factors revisited. Information and Software Technology, Vol. 43, no. 10, s. 617–628.

NRC, National Research Council. 1997. Digital Instrumentation and Control Systems in Nuclear Power Plants: Safety and Reliability Issues. National Academy Press. S. 43–51.

Parnas, D.L. 1998. ’Formal methods’ technology transfer will fail. Journal of Systems Software, Vol. 40, no. 3, s. 195–198.

Perry, D.E., Evangelist, M. 1987. An empirical study of software interface faults – An update. Proceedings of the 20th Hawaii International Conference on System Sciences, January 1987, Vol. II. S. 113–126.

Pitts, D.R. 1997. Metrics: problem solved? CrossTalk 10, Vol. 12, s. 28–30.

Powell, D. 1995. Failure mode assumptions and assumption coverage. Teoksessa: Predictably Dependable Computing Systems. (Eds. Randell. B. et al.). Springer-Verlag, Berlin. S. 123–154.

Powell, D., et al. 1995. Estimators for fault tolerance coverage evaluation. IEEE Transaction on Computer, Vol. 44, no. 2, s. 261–274.

Prechelt, L. 2000. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl. Submission to IEEE Computer. 7 s. (14.4.2000 .)

Pressman, R. S. 1997. Software Engineering: A Practioner's Approach. 4. p. McGraw-Hill. 885 s. IBSN 0-07-114603-2

Rifkin, S. 2001. Why software process innovations are not adopted. IEEE Software, Vol. 18, no 5, s. 110–112.

Rosenblum D.S., Weyuker, E.J. 1997. Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies. IEEE Transactions on Software Engineering, Vol. 23, no. 3, s. 146–156.

Roth, J.P. 1980. Computer Logic, Testing and Verification. Computer Science Press, 1980.

Rothermel, G., Mary Jean Harrold, M.J. 1996. Analyzing aegression test selection techniques. IEEE Transactions on Software Engineering, Vol. 22, no. 8, s. 529–551.

Sadeniemi, M. (päätoim.). 1996. Nykysuomen Sanakirja. 14. p. WSOY.

Schneidewind, N. 2001. Knowledge requirements for software quality measurement. Empirical Software Engineering, Vol. 6, no 3, s. 201–205.

Shick, G.J., Wolverton, R.W. 1973. Assessment of software reliability. Proceedings of Operations Research, Physical-Verlag, Wurzburg–Wien. S. 395–422.

Smidts, C., Li, M. (toim.). 2000. Software engineering measures for predicting software reliability in safety critical digital systems. University of Maryland. UMD-RE-2000-23 NUREG/GR-0019.

Steininger, A., Scherrer, C. 1997. On finding an optimal combination of error detection mechanisms based on results of fault injection experiments. Proceedings of the 27th International Symposium on Fault-Tolerant Computers. S. 238–247.

Tian, J., Nguyen, A., Allen, C., Appan, R. 2001. Experience with identifying and characterizing problem-prone modules in telecommunication software systems. Journal of Systems and Software, Vol. 57, no. 3, s. 207–215.

Valmari, A., Helovuo, J. 2001. Reaktiivisten järjestelmien automaattinen testaus. RATE-projekti, TTKK. ETX-Järjestelmät ja ohjelmistot, tulosseminaari 26.4.2001.

Voas, J. M. 1992. PIE: A dynamic faulure-based technique. IEEE Transactions on Software Engineering, Vol. 18, no. 8, s. 717–727.

Voas, J., Ghosh, A., Charron, F., Kassab, L. 1997. Reducing uncertainty about Common Mode Failures. Teoksessa: Proceedings of ISSRE, November 1997. Saatavilla: http://www.rstcorp.com.

Wallace, D.R., Kuhn, D.R. 2000. Lessons from 342 medical device failures. Saatavilla: http://hissa.nist.gov/effProject/

Watson, I.A., Edwards, G.A. 1979. Common-mode failures in redundant systems. Nuclear Technology, Vol. 46, no. 2, s. 183–191.

Watson, I.A., Smith, A.M. 1980. Common-cause failures – A dilemma in perspective. Proceedings Annual Reliability and Maintainability Symposium. S. 332–339.

Woit, D., Mason, D. 1998. Component independence for software system reliability. In: 2nd International Software Quality Week Europe, QWE'98, 9–13 November 1998, Brussels, Belgium.

Zhang, X., Pham, H. 2000. An analysis of factors affecting software reliability. The Journal of Systems and Software, Vol. 50, s. 43–56.

Julkaisija

Vuorimiehentie 5, PL 2000, 02044 VTT
Puh. (09) 4561
Faksi (09) 456 4374

Julkaisun sarja, numero ja raporttikoodi

VTT Tiedotteita 2193
VTT–TIED–2193

Tekijä(t)
Harju, Hannu & Koskela, Mika

Nimeke

Kustannustehokas ohjelmiston luotettavuuden suunnittelu ja arviointi
Osa 2

Tiivistelmä

Ohjelmistojen käyttäminen kriittisiin sovelluksiin on jatkuvassa kasvussa. Päinvastoin kuin laitteistoviat, ohjelmistoviat ovat systemaattisia ja ne voivat piileksiä pitkiä aikoja ennen paljastumistaan. Tämä tiedote on toinen osa tutkimussarjassa, jossa käsitellään ohjelmiston luotettavuuden kustannustehokasta suunnittelua ja arviointia. Osan kaksi teemoina ovat uusien menetelmien vähäisen käytön syyt, automaattinen testaaminen luotettavuuden ilmaisijana, ohjelmiston virhemekanismit sekä ohjelmistomittojen käyttö ohjelmiston luotettavuuden arvioinnin apuna.

Kaikki kehittyneet ohjelmistoprosessit ja -menetelmät lupaavat vähentää kustannuksia, vaivannäköä tai virheitä sekä parantaa laatua ja kasvattaa luotettavuutta. Huolimatta näistä toivottavista ominaisuuksista yritykset eivät ole omaksuneet nykyaikaisia menetelmiä käyttöönsä. Formaalit menetelmät, mittausprosessit, standardit ja ohjeet sekä jopa automaattiset testausmenetelmät eivät ole useimpien ohjelmistokehittäjien suosiossa.

Testaaminen on taitolaji. Jos oletetaan, että pienellä joukolla testitapauksia on löydettävä useimmat ohjelmistovirheet, testitapausten valitseminen on tärkeässä asemassa. Automaattiseen testaamiseen asetetaan suuria odotuksia. Sen odotetaan kasvattavan testikattavuutta ja siten parantavan luotettavuuden osoittamista, mutta automaatiossa taidontarve on toinen verrattuna perinteiseen testaamiseen.

Kattavuus on monitahoinen käsite. Puhutaan esimerkiksi testikattavuudesta ja koodikattavuudesta, jotka kummatkin sisältävät useita ominaisuuksia. Eroa kattavuuden ja kattavuusolettamuksien välillä ei kuitenkaan tehdä, koska virhemekanismin teoreettinen tuntemus ei ole hyvin kehittynyt. Virhemekanismi on kuitenkin kaiken testaamisen ja suunnittelun perustekijöitä etsittäessä virheitä ja suojauduttaessa niiltä.

Ohjelmistomittoja on perinteisesti käytetty ohjelmistoprosessin ja -projektin hallinnallisiin toimintoihin. Mittojen käyttöä ohjelmiston luotettavuuden arvioinnin apuna on tutkittu runsaasti, mutta käytännön ohjelmistotyöhön sopivia menetelmiä on verrattain vähän, mikä johtuu mitattavissa olevien ohjelmiston ominaisuuksien epämääräisestä suhteesta luotettavuuteen. Mittaustieto on kuitenkin merkittävä informaation lähde varsinkin ohjelmiston varhaisissa elinkaaren vaiheissa tapahtuvien ohjelmiston riskiosien tunnistamisessa ja korjaavien toimenpiteiden kohdentamisessa.

Avainsanat
software dependability assessment, software metrics, software reliability engineering, automated software testing, software measurement data

Toimintayksikkö
VTT Tuotteet ja tuotanto, Tekniikantie 12, PL 1301, 02044 VTT

ISBN			Projektinumero
951–38–6135–X (nid.) 951–38–6136–8 (URL: http://www.inf.vtt.fi/pdf/)			G2SU00241
Julkaisuaika	Kieli	Sivuja	Hinta
Maaliskuu 2003	Suomi, engl. tiiv.	107 s.	C

Projektin nimi	Toimeksiantaja(t)
Cost-effective reliability design and assessment of software	VTT, Tekes
Avainnimeke ja ISSN VTT Tiedotteita – Meddelanden – Research Notes 1235–0605 (nid.) 1455–0865 (URL: http://www.inf.vtt.fi/pdf/)	Myynti: VTT Tietopalvelu PL 2000, 02044 VTT Puh. (09) 456 4404 Faksi (09) 456 4374

Published by

Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
Phone internat. +358 9 4561
Fax +358 9 456 4374

Series title, number and report code of publication

VTT Research Notes 2193
VTT–TIED–2193

Author(s)
Harju, Hannu & Koskela, Mika

Title

Cost-effective reliability design and assessment of software Part 2

Abstract

Software is increasingly being used in critical applications. Unlike most hardware failures, software failures are systematic and software faults may lie hidden for a long time before being revealed. This publication is a second part of research project which study cost effective design and assessment of software dependability. Three specific themes are introduced: why software methods are not used in practice, test automation in demonstrating software dependability, failure mechanisms and software metrics utilised in software dependability assessment.

All advanced software prosesses and methods offer to reduce cost or effort, or defects, and to increase quality and reliability. In spite of these desireable features, most modern software methods are not adopted by organisations. Formal methods, measurement prosesses, standards and guidelines, and even automatic testers have not won favour with most of the software developers.

Testing is a skill. If a small number of test cases is expected to find most of the faults in the software, the task of selecting test cases is an important one. Automating tests is expected to increase testing coverage which means better demonstration for dependability, but the skill that automation needs differs from the skill of traditional testing.

Coverage is a multidimensional concept. We speak about testing coverage or code coverage, which both include tens of features, but we seldom make difference with coverage and coverage assumption. It is because the software failure mechanism, that is, the sequence fault – error – failure is not theoretically very well known, even if the very same failure mechanism is the basis for finding bugs by testing and shielded from their concequenses.

Software metrics has been traditionally used in issues of software process development and project management. Research of utilising software metrics in software dependability assessment has been done but few practical methods exist because of difficulties to describe dependability in terms of measurable properties. Software measurement data however offers valuable information in the early phases of software lifecycle in particular, where error-prone component identification and corrective operations are taken place.

Keywords
software dependability assessment, software metrics, software reliability engineering, automated software testing, software measurement data

Activity unit
VTT Industrial Systems, Tekniikantie 12, P.O.Box 1301, FIN–02044 VTT, Finland

ISBN	Project number
951–38–6135–X (soft back ed.) 951–38–6136–8 (URL: http://www.inf.vtt.fi/pdf/)	G2SU00241

Date	Language	Pages	Price
March 2003	Finnish, Engl. abstr.	107 p.	C

Name of project	Commissioned by
Cost-effective reliability design and assessment of software	VTT, Tekes

Series title and ISSN	Sold by
VTT Tiedotteita – Meddelanden – Research Notes 1235–0605 (soft back edition) 1455–0865 (URL: http://www.inf.vtt.fi/pdf/)	VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374

Tutkimus jakautui neljään osaan, joissa käsiteltiin valmistusalihankinnan asetusaikoja, tuplavarasto-ongelmaa, valmistuksen ja suunnittelun yhteistyötä sekä alihankintayhteistyön investointikysymyksiä. Tutkimukseen osallistui 11 suomalaista pk-yritystä. Kaikkien osien tulokset osoittivat, että kehitettävää on huomattavasti konstruktioissa ja valmistusprosesseissa, logistisissa prosesseissa sekä valmistuksen ja suunnittelun yhteistyössä. Yritysten välinen yhteistyö on kehitystoiminnan avain. Investointien osalta tarvitaan lisätutkimusta, jotta voitaisiin paremmin ymmärtää niihin liittyviä ongelmia. Tämän tutkimuksen perusteella voidaan kuitenkin olettaa, että lisäämällä pää- ja alihankkijan välistä yhteistyötä myös investoinneissa voidaan saavuttaa hyötyjä.

Tätä julkaisua myy
VTT TIETOPALVELU
PL 2000
02044 VTT
Puh. (09) 456 4404
Faksi (09) 456 4374

Denna publikation säljs av
VTT INFORMATIONSTJÄNST
PB 2000
02044 VTT
Tel. (09) 456 4404
Fax (09) 456 4374

This publication is available from
VTT INFORMATION SERVICE
P.O.Box 2000
FIN–02044 VTT, Finland
Phone internat. + 358 9 456 4404
Fax + 358 9 456 4374

ISBN 951–38–6135–X (nid.)
ISSN 1235–0605 (nid.)
ISBN 951–38–6136–8 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0865 (URL: http://www.inf.vtt.fi/pdf/)