hyväksymispäivä arvosana

arvostelija

Mutaatiotestaus oliojärjestelmissä

Eveliina Pakarinen

Aine
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 16. lokakuuta 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Eveliina Pakarinen

Työn nimi — Arbetets titel — Title

Mutaatiotestaus oliojärjestelmissä

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Aine	16. lokakuuta 2015	11

Tiivistelmä — Referat — Abstract

Aineen tiivistelmä

ACM Computing Classification System (CCS):

D.2.4 [Software/Program Verification]

D.2.5 [Testing and Debugging]

D.3.3 [Language Constructs and Features]

Avainsanat — Nyckelord — Keywords
mutaatiotestaus, oliojärjestelmät, Java

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Vaikka testauksen avulla ei voikaan varmistua ohjelmiston oikeellisuudesta, voidaan testausta käyttää välineenä ohjelmiston laadun parantamisessa.

Mutaatiotestauksen periaatteena on ohjelmoijien tekemien ohjelmointivirheiden simulointi [JH11, s. 649]. Tavoitteena mutaatiotestauksessa on tutkia, ovatko ohjelmistoa varten tehdyt testit laadukkaita ja havaitaanko niillä kattavasti ohjelmistossa mahdollisesti esiintyvät virheet ja ongelmat.

Johdanto jäsennelty asianmukaisesti (kenelle, miksi, millaisessa ympäristössä; ratkaisun lähestymistapa; tutkimuskysymys, tulokset ja impakti), pituus 1,5 - 2 s.

2 Testaus oliojärjestelmissä

Olioperustaisen ohjelmoinnin kehityksen myötä klassisia ohjelmistojen testausmenetelmiä on sopeutettu mahdollistamaan *oliojärjestelmien* (*object oriented systems*) kattava ja laadukas testaaminen. Vaikka olioperustainen ohjelmointi ratkaisee joitakin proseduraalisen ohjelmoinnin suunnittelu- ja toteutusongelmia, olio-ohjelmoinnin mukana tulevat uudet haasteet vaativat uusien testaus- ja analysointimenetelmien kehittämistä.

2.1 Testauksen rooli oliojärjestelmissä

Testausta käytetään ohjelmistokehityksessä ohjelmiston laadun varmistamiseen ja auttamaan virheiden havaitsemisessa jo kehitysvaiheen aikana. Ohjelmistojen testaamisen ensisijainen tavoite on siis paljastaa virheitä, joiden havaitseminen muiden laadunvarmistusmenetelmien avulla olisi työlästä tai mahdotonta [Bin99, s. 59]. Testauksen avulla pyritään lisäksi varmistamaan, että ohjelma toimii sille asetettujen vaatimusten mukaisesti.

Olio-ohjelmoinnissa testaukseen tuovat haasteita olio-ohjelmien erityispiirteet, joita ovat muun muuassa kapselointi, perintä, dynaaminen sidonta ja polymorfismi [MP08, s. 86].

2.2 Testauksen tasot

Ohjelmistoja voidaan testata usealla tasolla. Tasoja ovat yksikkö-, integraatio- ja järjestelmätasot ja ne muodostuvat yhdestä tai useammasta ohjelman komponentista, joita tason testeillä testataan [Bin99, s. 45]. Komponentti voi olio-ohjelmissa olla esimerkiksi yksittäinen metodi tai luokka, ohjelman luokkien välinen rajapinta tai jo valmis ohjelmisto.

Alimmalla testauksen tasolla *yksikkötestauksessa* (*unit testing*) [Bin99, s. 45] testataan ohjelman pienimpiä suoritettavissa olevia komponentteja. Olio-ohjelmissa näitä komponentteja ovat yksittäiset metodit ja oliot.

Yksikkötestauksesta seuraava taso ylöspäin on *integraatiotestaus* (*integration testing*) [Bin99, s. 45], jossa tarkastellaan järjestelmän tai sen osien yhteistoimintaa. Integraatiotestauksessa testataan siis järjestelmän osien välisiä rajapintoja ja osien keskinäistä kommunikointia. Olio-ohjelmissa luokkien muodostuminen perinnän avulla ja luokkien koostuminen toisten luokkien olioista aiheuttaa, että integraatiotestaukselle on olio-ohjelmoinnissa tarvetta jo ohjelmoinnin alkuvaiheessa.

Valmista integroitua sovellusta testataan *järjestelmätestauksen* (*system testing*) [Bin99, s. 45] avulla. Tällä testauksen tasolla keskitytään vain valmiissa sovelluksessa esiintyvien piirteiden testaamiseen. Testauksen kohteena voi olla esimerkiksi sovelluksen toiminnallisuus, suorituskyky tai sovelluksen kestämä kuormitus [Bin99, s. 45].

2.3 Testien suunnittelu

Testien suunnitteluun ja kehittämiseen voidaan käyttää erilaisia menetelmiä. Testausmenetelmän avulla kuvataan näkökulmaa, josta esimerkiksi ohjelman lähdekoodia tarkastellaan testejä kehitettäessä [Bin99, s. 51]. Uusia testejä kehitettäessä testausmenetelmiä ovat esimerkiksi white box - ja black box -testaus sekä niitä yhdistävä hybriditestaus. Lisäksi virheisiin perustuvan testausmenetelmän avulla voidaan kehittää jo olemassa olevia testejä.

Ohjelman sisäisen rakenteen eli lähdekoodin tuntemukseen perustuvaa testausmenetelmää kutsutaan *white box -testaukseksi* [Bin99, s. 52]. White box -testausta voidaan käyttää esimerkiksi yksikkötestauksessa apuna testien suunnittelussa, sillä lähdekoodin tuntemus auttaa kehittämään testejä

yksittäisille metodeille ja olioille.

Black box -testaukseksi eli *funktionaaliseksi testaukseksi* [Bin99, s. 52] kutsutussa testausmenetelmässä testejä suunnitellaan ohjelmiston toiminnallisuuden tuntemuksen avulla. Koska valmiin sovelluksen piirteitä testattaessa tutkitaan myös sovelluksen ulkoista toiminnallisuutta, on black box -testausmenetelmästä apua suunniteltaessa testejä esimerkiksi järjestelmätestaukseen.

Gray box - eli *hybriditestauksessa* [Bin99, s. 52] yhdistetään white box - ja black box -testausmenetelmien piirteitä. Sekä white box - että black box -testausmenetelmää voidaan siis käyttää testien suunnittelussa useilla testauksen tasoilla joko erikseen tai molempien piirteitä yhdistäen.

Testausmenetelmää, jossa ohjelman lähdekoodiin lisätään virheitä, kutsutaan *virheperustaiseksi testausmenetelmäksi* (*fault-based testing*) [Bin99, s. 52]. Esimerkkinä virheperustaisesta testausmenetelmästä on mutaatiotestaus, jonka avulla tutkitaan testien kykyä havaita ohjelmistossa olevia virheitä [DLS78, s. 36]. Mutaatiotestauksessa ja perinteisessä ohjelmistotestauksessa testaukseen liittyvät tavoitteet ovat toistensa vastakohtia. Perinteisessä testauksessa keskitytään parantamaan ohjelmiston laatua, kun taas mutaatiotestauksessa kehityksen kohteena ovat olemassa olevat testit ja niiden laatu.

2.4 Testauksen rajoitukset

Yksi testaukseen liittyvistä rajoituksista on, että testauksen avulla ei voi aina todeta ohjelmiston oikeellisuutta. Jotta oikeellisuus voidaan todistaa, vaaditaan, että ohjelman oikea toiminta testataan kaikilla mahdollisilla syötteillä ja niiden kombinaatioilla. Ohjelman oikeellisuuden todistaminen vastaa siis ohjelman kattavaa testaamista. Kattava testaaminen on kuitenkin käytännössä usein mahdotonta toteuttaa muille kuin triviaaleille ohjelmille [Bin99, s. 58]. Oikellisuuden todistamiseen liittyen Edsger Dijkstra totesikin: *"Program testing can be used to show the presence of bugs, but never to show their absence!"* [DDH72, s. 6].

Suoritettujen testien tulosten tulkintaan liittyy myös rajoituksia ja epävarmuutta. Epävarmuus ilmenee, kun testien suorituksen tuloksia varten ei

ole olemassa luotettavia odotettuja tuloksia vertailukohdaksi. Tällöin testauksesta saatujen toteutuneiden tulosten tulkinta ja arviointi on epävarmaa. Toteutuneista tuloksista ei voi siis luotettavasti päätellä, menivätkö testit läpi vai eivät [Bin99, s. 58].

Epävarmuutta liittyy myös testattavan järjestelmän halutulle toiminnallisuudelle asetettuihin vaatimuksiin. Toiminnallisuusvaatimusten todentaminen testauksen avulla ei ole mahdollista, joten vaatimuksia on käytettävä vain vertailukohtana testauksen tulosten tulkinnassa [Bin99, s. 58]. Jos virheellisiä tai puutteellisia vaatimuksia käytetään testejä tehdessä, voi siitä seurata harhaanjohtavia testejä. Testauksen avulla ei myöskään voi paljastaa lähdekoodista puuttuvia osia, sillä olematonta koodia ei voi testata [Bin99, s. 58].

3 Mutaatiotestaus oliojärjestelmissä

Testaukseen sisältyvien rajoitusten lisäksi testaukseen liittyy myös epävarmuutta käytettävän testausjärjestelmän oikeellisuudesta ja oikeellisuuden varmistamisesta [MW78, s. 209]. Tämä herättää kysymyksen siitä, kuka voi "valvoa valvojia" eli kuinka varmistetaan ohjelmiston testien laadukkuus. Yksi mahdollisuus testien kehittämiseen ja niiden laadun parantamiseen on *mutaatiotestaus*. Mutaatiotestauksen avulla voidaan mitata, kuinka tehokkaasti ohjelmiston testeillä havaitaan ohjelmistossa esiintyviä virheitä [JH11, s. 649].

3.1 Mutaatiotestauksen esittely

Mutaatiotestauksesta kirjoitettiin ensimmäisiä kertoja jo 1970-luvulla. DeMillon, Liptonin ja Saywardin artikkeli [DLS78] vuodelta 1978 on yksi ensimmäisistä uraauurtavista mutaatiotestausta esittelevistä artikkeleista. Mutaatiotestauksen tutkimus on lisääntynyt vuosien kuluessa, ja erityisesti 2000-luvulla uusia tuloksia on julkaistu paljon [Off11, s. 1102]. Tutkimuksessa suuntana on ollut etsiä keinoja, joilla mutaatiotestaus voidaan muuttaa käytännölliseksi testausmenetelmäksi [JH11, s. 649].

Mutaatiotestausprosessissa ensimmäinen vaihe on käsitellä ohjelmiston alkuperäistä lähdekoodia *mutaatio-operaattoreilla*, jotka muuntavat koodia

muodostaen siitä virheellisiä versioita [MHK06, s. 869]. Näitä virheellisiä ohjelmakoodin versioita kutsutaan *mutanteiksi*. Mutaatio-operaattorit kuvaavat algoritmeja, joiden avulla lähdekoodia käsitellään koodin muuntamisen aikana.

Mutanttien generoinnin jälkeen ohjelmiston alkuperäiset testit suoritetaan sekä muuntamattoman lähdekoodin että jokaisen mutantin kohdalla [JH11, s. 652]. Tavoitteena on havaita testien avulla lähdekoodiin tehdyt muutokset.

Testien suorituksen jälkeen suorituksesta saatuja tuloksia verrataan toisiinsa. Testituloksia vertailtaessa voidaan päästä kahteen lopputulokseen [DLS78, s. 36]. Alkuperäiselle muuntamattomalle lähdekoodille suoritettujen testien tulos voi:

1. erota yhdelle mutantille suoritettujen testien tuloksesta tai
2. olla sama kuin yhdelle mutantille suoritettujen testien tulos.

Tapauksessa 1. alkuperäiset testit eivät ole menneet läpi mutantin kohdalla. Tämä tarkoittaa, että mutantti on tapettu eli lähdekoodiin tehty muutos on havaittu [DLS78, s. 36].

Tapauksessa 2. alkuperäiset testit ovat menneet läpi mutantin kohdalla eli mutantti on jäänyt eloon. Mutantin jäämiselle eloon on kaksi vaihtoehtoista selitystä [DLS78, s. 36]. Ensimmäinen selitys on, että alkuperäiset testit eivät ole riittävän hyvät, jotta niiden avulla voidaan havaita lähdekoodiin tehty muutos. Toinen selitys on, että mutantin toiminta ei eroa alkuperäisen ohjelman toiminnasta eli kyseessä on *ekvivalentti mutantti*. Ekvivalentit mutantit ovat syntaktisesti erilaisia kuin alkuperäinen ohjelma mutta toiminnaltaan ne ovat samanlaisia alkuperäisen ohjelman kanssa [JH11, s. 652].

Mutaatiotestausprosessi tuottaa lopputuloksena *mutaatiopistemäärän* (*mutation adequacy score*), jonka avulla voi arvioida ohjelmiston testien laadukkuutta ja kykyä havaita lähdekoodissa olevia vikoja [JH11, s. 652]. Mutaatiopistemäärä MP lasketaan kaavalla

$$MP = \frac{T}{K - E}, \tag{1}$$

missä T on tapettujen mutanttien määrä, K on kaikkien mutanttien määrä ja E on ekvivalenttien mutanttien määrä. Mutaatiopistemäärän maksimiarvo

1 saavutetaan, kun testeillä saadaan tapettua kaikki mutantit.

3.2 Mutaatio-operaattorit oliojärjestelmissä

Mutaatio-operaattorit ovat tärkeässä asemassa mutaatiotestauksessa, sillä mutaatiotestauksen tehokkuus riippuu siitä, minkälaisia virheitä mutaatio-operaattoreilla luodaan lähdekoodiin [MKO02, s. 352]. Perinteisesti mutaatiotestausta on hyödynnetty proseduraalisessa ohjelmoinnissa, minkä vuoksi myös mutaatio-operaattorit on kehitetty tukemaan suurinta osaa proseduraalisten ohjelmointikielten piirteistä [MKO02, s. 352].

Olioperustaisiin ohjelmointikieliin sisältyy kuitenkin uusia ominaisuuksia, joiden käytöstä aiheutuu erilaisia virheitä verrattuna proseduraalisessa ohjelmoinnissa esiintyviin virheisiin. Uusien virheiden ilmeneminen olio-ohjelmissa on johtanut uusien olioperustaisten mutaatio-operaattorien kehittämiseen.

Olio-perustaisten ohjelmien mutaatiotestaukseen käytettävää menetelmää kutsutaan *luokkamutaatioksi* [KCM00]. Luokkamutaatiomenetelmän kehittivät Kim, Clark ja McDermid [KCM00] Java-ohjelmointikielessä esiintyvien virheiden pohjalta. Heidän kehittämiensä *luokkamutaatio-operaattorien* avulla mutaatiotestausmenetelmää voidaan soveltaa Java-ohjelmointikielellä toteutettuihin olioperustaisiin ohjelmiin.

Kimin, Clarkin ja McDermidin kehittämät luokkamutaatio-operaattorit ovat toimineet lähtökohtana myös muiden tutkijoiden luokkamutaatiotutkimuksessa ja uusien luokkamutaatio-operaattorien kehittämisessä. Ma, Kwon ja Offutt kehittivät vuonna 2002 Java-ohjelmointikieltä varten joukon uusia luokkamutaatio-operaattoreita, jotka perustuivat aiemmin kehitettyihin operaattoreihin [MKO02, s. 352]. Heidän tavoitteenaan oli parantaa ja kehittää olemassa olevia luokkamutaatio-operaattoreita, jotta Java-ohjelmien luokkien välisten suhteiden testaaminen olisi mahdollista mutaatiotestauksella [MKO02, s. 362].

Taulukossa 1 on listattu Man, Kwonin ja Offuttin kehittämät luokkamutaatio-operaattorit. Operaattorit on jaettu kuuteen ryhmään. Ryhmät perustuvat niihin olio-ohjelmointikielen piirteisiin, joita ryhmän operaattoreilla muunnetaan [MKO02, s. 355].

Mutaatio-operaattorien avulla kuvataan algoritmeja, joilla lähdekoodia

muokataan mutantteja muodostettaessa. Esimerkiksi JSC-operaattori lisää ilmentymämuuttujiin *static*-määreen tehden niistä luokkamuuttujia tai vastaavasti poistaa luokkamuuttujista *static*-määreen tehden niistä ilmentymämuuttujia. JSC-operaattorin ja myös muiden Man, Kwonin ja Offuttin kehittämien luokkamutaatio-operaattorien nimet ja kuvaukset voi nähdä taulukosta 1. Taulukon 1 avulla näkee myös, mihin ryhmiin luokkamutaatio-operaattorit kuuluvat. Tarkemmat tiedot operaattorien toiminnasta löytyvät artikkelista [MKO02].

Ryhmä	**Operaattori**	**Kuvaus**
Pääsynvalvonta	ACM	Access modifier change
Perintä	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overridden method calling position change
	IOR	Overridden method rename
	ISK	*super* keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorfismi	PNC	*new* method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other compatible type
Metodin ylikuormitus	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Javan erityispiirteet	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor create
Yleiset ohjelmointivirheet	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Taulukko 1: Luokkamutaatio-operaattoreita Javalle.

4 Mutaatiotestauksen haasteet

Vaikka mutaatiotestausta voidaan käyttää ohjelmiston olemassa olevien testien kehittämiseen ja testien laadun parantamiseen, ei mutaatiotestausmenetelmän käyttö ole ongelmatonta. Mutaatiotestaukseen liittyy haasteita, jotka estävät menetelmän käyttämisen laaja-alaisesti osana testausprosessia [JH11,

s. 652].

Yksi mutaatiotestauksen ongelma-alueista on tehokkuuteen liittyvät ongelmat. Tehokkuusongelmia esiintyy erityisesti silloin, kun jokaisen mutantin kohdalla suoritetaan kaikki ohjelmistoa varten tehdyt testit [JH11, s. 652]. Testien suoritus jokaisen mutantin kohdalla hidastaa mutaatiotestausprosessia ja vaatii paljon laskentatehoa. Tehokkuusongelman ratkaisemiseksi on kuitenkin esitetty useita ratkaisuehdotuksia [JH11, s. 653]. Ratkaisuehdotuksia ovat esimerkiksi generoitujen mutanttien määrän vähentäminen ja mutanttien ja testien suorituskustannusten pienentäminen.

Toinen mutaatiotestaukseen liittyvistä ongelma-alueista on ekvivalentit mutantit. Ekvivalenttien mutanttien tunnistaminen algoritmien avulla on ratkaisematon ongelma [JH11, s. 657]. Koska ekvivalentteja mutantteja ei voi tunnistaa laskennan avulla, ihmisten on suoritettava ekvivalenttien mutanttien tunnistaminen tutkimalla mutanttia ja vertaamalla sen toimintaa alkuperäisen ohjelmiston toimintaan. Tämä menetelmä johtaa mutaatiotestauksen toteuttamiseen tarvittavan työmäärän kasvuun. Ekvivalenttien mutanttien tunnistamisongelma on kuitenkin herättänyt runsaasti teoreettista kiinnostusta ja mahdollisia tunnistamistekniikoita tutkittu paljon [JH11, s. 657].

Kolmas ongelma-alue liittyy mutaatiotestauksen vaiheeseen, jossa ihmisten on tarkastettava testauksesta saadut tulokset [JH11, s. 652]. Testien tulosten tarkastusvaiheessa tarkastetaan tuloste, joka saatiin, kun testit suoritettiin alkuperäiselle ohjelmalle. Testaustulosten tarkastusvaihe on usein testausprosessin työläin osa [JH11, s. 653]. Tämä ongelma ei liity pelkästään mutaatiotestaukseen vaan ongelma ilmenee myös muissa testausmenetelmissä. Mutaatiotestauksessa ekvivalenttien mutanttien tunnistaminen ja suoritettujen testien suuri määrä lisäävät kuitenkin työmäärää, joka tarvitaan testauksen tulosten tarkastamiseen.

5 Yhteenveto

Conclusion.

Lähteet

Bin99 Binder, Robert V.: *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999, ISBN 0-201-80938-9.

DDH72 Dahl, O. J., Dijkstra, E. W. ja Hoare, C. A. R. (toimittajat): *Structured Programming*. Academic Press Ltd., London, UK, UK, 1972, ISBN 0-12-200550-3.

DLS78 DeMillo, R. A., Lipton, R. J. ja Sayward, F. G.: *Hints on Test Data Selection: Help for the Practicing Programmer*. Computer, 11(4):34–41, huhtikuu 1978, ISSN 0018-9162. `http://dx.doi.org/10.1109/C-M.1978.218136`.

JH11 Jia, Yue ja Harman, Mark: *An Analysis and Survey of the Development of Mutation Testing*. IEEE Trans. Softw. Eng., 37(5):649–678, syyskuu 2011, ISSN 0098-5589. `http://dx.doi.org/10.1109/TSE.2010.62`.

KCM00 Kim, Sunwoo, Clark, John A. ja McDermid, John A.: *Class Mutation: Mutation Testing for Object-oriented Programs*. Teoksessa *Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems*, lokakuu 2000.

MHK06 Ma, Yu Seung, Harrold, Mary Jean ja Kwon, Yong Rae: *Evaluation of Mutation Testing for Object-oriented Programs*. Teoksessa *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, sivut 869–872, New York, NY, USA, 2006. ACM, ISBN 1-59593-375-1. `http://doi.acm.org/10.1145/1134285.1134437`.

MKO02 Ma, Yu Seung, Kwon, Yong Rae ja Offutt, J.: *Inter-class mutation operators for Java*. Teoksessa *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, sivut 352–363, 2002.

MP08 Mariani, Leonardo ja Pezze, Mauro: *Testing Object-Oriented Software*, luku Emerging Methods, Technologies and Process Management in Software Engineering, sivut 85–108. Wiley-IEEE Computer Society Press, 2008.

MW78 Manna, Zohar ja Waldinger, Richard J.: *The Logic of Computer Programming*. IEEE Trans. Software Eng., 4(3):199–229, 1978. `http://doi.ieeecomputersociety.org/10.1109/TSE.1978.231499`.

Off11 Offutt, Jeff: *A mutation carol: Past, present and future*. Information & Software Technology, 53(10):1098–1107, 2011. `http://dx.doi.org/10.1016/j.infsof.2011.03.007`.