hyväksymispäivä arvosana

arvostelija

Mutaatiotestaus oliojärjestelmissä

Eveliina Pakarinen

Aine
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 4. 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	4. lokakuuta 2015	7

Tiivistelmä — Referat — Abstract

Aineen tiivistelmä

Avainsanat — Nyckelord — Keywords
mutaatiotestaus, oliojärjestelmät, Java

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto [(1s)]

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. Olio-ohjelmoinnissa testauksen tekevät haastavaksi muun muuassa kapselointi, perintä, dynaaminen sidonta ja polymorfismi [MP08, s. 86].

Testausta käytetään ohjelmistokehityksessä ohjelmiston laadun varmistamiseen ja auttamaan virheiden havaitsemisessa jo kehitysvaiheen aikana. Ohjelmistojen testaamisen ensisijainen tavoite on paljastaa virheitä, joiden havaitseminen muiden laadunvarmistusmenetelmien avulla olisi työlästä tai mahdotonta. Lisäksi testauksen avulla on pyritty varmistamaan, että ohjelma toimii sille asetettujen vaatimusten mukaisesti [Bin99, s. 59] (lähde pitää siirtää johonkin toiseen kohtaan/lisätä jokaisen lauseen perään lähde).

Testaukseen kuuluu kuitenkin myös (tiettyjä) rajoituksia. Sen avulla ei voi esimerkiksi todeta ohjelmiston oikeellisuutta. Ohjelmistojen oikeaksi todistamiseen ja testauksen rajoituksiin liittyen Dijkstra kirjoitti/on kirjoittanut seuraavan korollaarin: *"Program testing can be used to show the presence of bugs, but never to show their absence!"* [DDH72, s. 6].

Vaikka testauksen avulla ei voikaan varmistua ohjelmiston oikeellisuudesta, voidaan testausta käyttää välineenä ohjelmiston laadun parantamisessa. Testaukseen sisältyvien rajoitusten lisäksi siihen 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 strategia testien laadun varmistamiseen on *mutaatiotestaus*. Mutaatiotestauksessa ohjelmiston alkuperäistä lähdekoodia käsitellään *mutaatio-operaattoreilla* (*mutation operators*), jotka muuntavat koodia muodostaen siitä virheellisiä versioita. Näitä virheellisiä ohjelmakoodin versioita kutsutaan *mutanteiksi* [MHK06, s. 869].

Mutaatiotestaus on *virheperustainen testausmenetelmä* (*fault based jo-*

takin), jonka 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.

Kuvaus aineesta ja sen osioista?

2 Testaus oliojärjestelmissä (5s) [2s]

Tähän tietoa siitä, mikä on oliojärjestelmä. Aiheen esittely.

Olioperustaisen ohjelmoinnin kehityksen myötä klassisia ohjelmistojen testausmenetelmiä (avaa lisää näitä klassisia menetelmiä) on sopeutettu uusiin vaatimuksiin, joita *oliojärjestelmien* (*object oriented systems*) kattava ja laadukas testaaminen tuo mukanaan. Vaikka olioperustainen ohjelmointi ratkaisee joitakin proseduraalisen ohjelmoinnin suunnittelu- ja toteutusongelmia, sen mukana tulevat uudet haasteet vaativat uusien testaus- ja analysointimenetelmien kehittämistä.

Testausta käytetään ohjelmistokehityksessä ohjelmiston laadun varmistamiseen ja samalla auttamaan virheiden havaitsemisessa jo kehitysvaiheen aikana. Ohjelmistojen testaamisen ensisijainen tavoite on siis paljastaa virheitä, joiden havaitseminen muiden laadunvarmistusmenetelmien avulla olisi työlästä tai mahdotonta. Lisäksi testauksen avulla on pyritty varmistamaan, että ohjelma toimii sille asetettujen vaatimusten mukaisesti [Bin99, s. 59] (lähde pitää siirtää johonkin toiseen kohtaan/lisätä jokaisen lauseen perään lähde).

2.1 Testauksen erityispiirteet oliojärjestelmissä/yleisesittely (onko tarpeellinen omana osionaan?)

Onko tämä tarpeellinen oma osio vai lisätäänkö tiedot luvun introon?

Näitä olio-ohjelmoinnin erityispiirteitä ovat muun muuassa kapselointi, perintä, *dynaaminen linkitys* (*dynamic binding*) ja polymorfismi [MP08, s. 86].

2.2 Testauksen tasot

Avataanko testauksen tasoja enemmän? Kuinka paljon niistä on kerrottava?

Ohjelmistoja voidaan testata monella tasolla. Testauksen tasot muodostetaan määrittämällä joukko ohjelman osia eli komponentteja, joita halutaan testata. Alimmalla testauksen tasolla testattavat ohjelman osat ovat pienimpiä mahdollisia suoritettavissa olevia komponentteja. Tämän tason testausta kutsutaan *yksikkötestaukseksi* (*unit testing*) ja testattavat osat ovat olio-ohjelmissa esimerkiksi yksittäisiä metodeja tai luokkia [Bin99, s. 45].

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

Valmista integroitua sovellusta testataan *järjestelmätestauksen* (*system testing*) 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 suunnittelumallit

Kuinka paljon näitä avataan lisää? Miten paljon on relevanttia tutkielman kannalta?

Testien suunnittelussa ja kehittämisessä voidaan myös käyttää erilaisia suunnittelumalleja, joiden avulla kuvataan testien suunnitteluun käytettävää näkökulmaa. Ohjelman sisäisen rakenteen eli lähdekoodin tuntemukseen perustuvaa testien suunnittelumallia kutsutaan *white box -testaukseksi. Black box -testaukseksi* tai *funktionaaliseksi testaukseksi* kutsutussa suunnittelumallissa testejä suunnitellaan analysoimalla ohjelmiston ulkoista toiminnallisuutta [Bin99, s. 52].

2.4 Testauksen rajoitukset

Kuinka paljon rajoituksista on kerrottava? Niitä on aika paljon erilaisia. Mitkä niistä ovat relevantteja tämän tutkielman kannalta?

Testauksessa on kuitenkin myös rajoituksia. Sen avulla ei voi esimerkiksi todeta ohjelmiston oikeellisuutta. Ohjelmistojen oikeaksi todistamiseen ja testauksen rajoituksiin liittyen Dijkstra kirjoitti seuraavan korollaarin: *"Program testing can be used to show the presence of bugs, but never to show their absence!"* [DDH72, s. 6].

3 Mutaatiotestaus oliojärjestelmissä (5-7s) [3s]

Haasteisiin voidaan vastata mutaatiotestauksen avulla

Vaikka testauksen avulla ei voikaan varmistua ohjelmiston oikeellisuudesta, voidaan sitä käyttää välineenä ohjelmiston laadun parantamisessa. Testaukseen liittyy kuitenkin myös epävarmuus käytettävän testausjärjestelmän oikeellisuudesta ja oikeellisuuden varmistamisesta [MW78, s. 209]. Tämä herättää kysymyksen siitä, kuka voi valvoa valvojia eli kuinka varmistetaan ohjelmiston testien laadukkuus.

3.1 Mutaatiotestauksen yleisesittely

Laajentaa sopivista kohdista yleisesittelyä.

Yksi strategia testien laadun varmistamiseen on *mutaatiotestaus*. Mutaatiotestauksessa ohjelmiston alkuperäistä lähdekoodia käsitellään *mutaatio-operaattoreilla* (*mutation operators*), jotka muuntavat koodia muodostaen siitä virheellisiä versioita. Näitä virheellisiä ohjelmakoodin versioita kutsutaan *mutanteiksi* [MHK06, s. 869].

Mutanttien generoinnin jälkeen ohjelmiston alkuperäiset testit suoritetaan jokaisen mutantin kohdalla. Tavoitteena on, että testien avulla havaitaan lähdekoodiin tehdyt muutokset. Jos alkuperäiset testit eivät mene läpi, se tarkoittaa, että mutantti on tapettu eli lähdekoodiin tehdyt muutokset on havaittu [KCM00, s. 9].

Mutaatiotestausprosessi tuottaa lopputuloksena *mutaatiopistemäärän*

(*mutation adequacy score*), jonka avulla voidaan arvioida ohjelmiston testien laadukkuutta ja kykyä havaita lähdekoodissa olevia vikoja. Mutaatiotestaus on *virheperustainen testausmenetelmä* (*fault based jotakin*), jonka taustaperiaatteena 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.

3.2 Mutaatiotestauksen piirteet oliojärjestelmissä (erot muihin järjestelmiin?)

Erityisesti juuri nuo mutaatio-operaattorit ja minkälaisia erilaisia niitä on. Esittelyä tulee mutaatio-operaattoreista.

4 Mutaatiotestauksen haasteet (5-7s) [3s]

Misi mutaatiotestaus ei ole päätynyt suureen suosioon/käyttöön? Valitaan muutama haaste ja esitellään niitä? Kerrotaan että myös muita haasteita, mutta ei esitellä niitä?

4.1 Ekvivalentit mutantit

Ratkaisematon ongelma. Pitäisi avata ja kehittää esimerkki.

4.2 Tehokkuusongelmat

Laitteisto, testien määrä, ihmisten aika tulee vastaan. Esimerkkejä.

4.3 Muita ongelmia?

Varmasti on muita, mutta kuinka paljon niitä mahtuu tähän esitelmään?

4.4 Käytännöntoteutus (miten käytännössä on yritetty kiertää haasteet)

Onko tämä relevantti aihealue käsitellä?

5 Yhteenveto [(1s)]

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.

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*, sivut 9–12, 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`.

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`.