Mutaatiotestaus oliojärjestelmissä

Eveliina Pakarinen

Referaatti
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 16. syyskuuta 2015

Olioperustaisen ohjelmoinnin kehityksen myötä klassisia ohjelmistojen testausmenetelmiä on sopeutettu uusiin vaatimuksiin, joita oliojärjestelmien kattava ja laadukas testaaminen vaatii. Vaikka olioperustainen ohjelmointi ratkaisee joitakin proseduraalisen ohjelmoinnin suunnittelu- ja toteutusongelmia, se tuo mukanaan myös uusia haasteita, jotka vaativat uusien testaus- ja analysointimenetelmien kehittämistä. Näitä olio-ohjelmoinnin erityispiirteitä ovat muun muuassa kapselointi, perintä, dynaaminen linkitys ja polymorfismi [MP08, s. 86].

Testausta on käytetty 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].

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

Ohjelmistoja voidaan testata monella tasolla ja testien suunnittelussa voidaan käyttää erilaisia strategioita. Testauksen tasot koostuvat joukoista ohjelman 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]. Tällä testauksen tasolla testien suunnittelussa voidaan käyttää ohjelman sisäiseen rakenteeseen perustuvaa suunnittelumallia, jota kutsutaan *white box -testaukseksi* (*white box testing*). Tässä mallissa ohjelmiston lähdekoodia käytetään apuna testien valmistamisessa [Bin99, s. 52].

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 niiden koostuminen toisten luokkien olioista aiheuttaa sen, että integraatiotestaukselle on tarvetta jo aikaisin olioperustaisen ohjelmoinnin alkuvaiheessa [Bin99, s. 45].

Jo 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 sen kestämän kuormituksen määrä [Bin99, s. 45]. Tämän tason testien suunnittelustrategiana voidaan käyttää ohjelmiston ulkoisen toiminnallisuuden analysointia. Tällaista testausstrategiaa kutsutaan *black box -testaukseksi* (*black box testing*) tai *funktionaaliseksi testaukseksi* (*functional testing*) [Bin99, s. 52]. Kuitenkin sekä white box - että black box -testausta voidaan käyttää kaikilla testauksen tasoilla testien suunnittelun apuna.

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

Yksi strategia testien laadun varmistamiseen on *mutaatiotestaus* (*mutation testing*) [**?**]. Mutaatiotestauksessa ohjelmiston alkuperäistä lähdekoodia muunnetaan *mutaatio-operaattorien* (*mutation operators*) avulla lisäämällä siihen virheitä. Näitä virheellisiä ohjelmakoodin versioita kutsutaan *mutanteiksi* (*mutants*) [MHK06]. Mutanttien generoinnin jälkeen ohjelmiston alkuperäiset testit suoritetaan jokaisen mutantin kohdalla, ja toivotaan, että testit eivät mene läpi. Jos testi tuottaa väärän lopputuloksen, se tarkoittaa, että mutantti on tapettu eli lähdekoodiin tehdyt muutokset on havaittu.

Mutaatotestausprosessi tuottaa lopputuloksena *mutaatiopistemäärän* (*mutation adequacy score*), jonka avulla ohjelmiston testien laadukkuutta ja kykyä havaita lähdekoodissa olevia vikoja voidaan arvioida. Mutaatiotestaus on virheperustainen testausmenetelmä, jonka taustaperiaatteena on ohjelmoijien tekemien ohjelmointivirheiden simulointi [JH11]. Tavoitteena mutaatiotestauksessa on tutkia, ovatko ohjelmistoa varten tehdyt testit laadukkaita ja havaitsevatko ne kattavasti ohjelmistossa mahdollisesti esiintyvät

virheet ja ongelmat.

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`.

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`.