hyväksymispäivä arvosana

arvostelija

Ohjelmistoarkkitehtuurien harjoitustyö

Eveliina Pakarinen

Harjoitustyö
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 18. helmikuuta 2017

Sisältö

1 Johdanto

Perinteisiä ohjelmistojen testausmenetelmiä on olioperustaisen ohjelmoinnin kehityksen myötä sopeutettu uusiin olio-ohjelmoinnin mukana tuleviin haasteisiin. Olio-ohjelmoinnin avulla voidaan ratkaista joitakin proseduraalisen ohjelmoinnin ongelmia [MP08, s. 86]. Olio-ohjelmoinnin piirteet, kuten kapselointi ja perintä, aiheuttavat kuitenkin uusia ongelmia, jotka vaativat uusien testaus- ja analysointimenetelmien kehittämistä.

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 [MP08, s. 86].

2.1 Testauksen rooli

IEEE:n standardin mukaan *ohjelmointivirheen* (*error*) aiheuttamaa ohjelmiston lähdekoodiin päässyttä virheellistä kohtaa kutsutaan *viaksi* (*fault*) [IEE10, s. 5]. Lähdekoodissa oleva vika saattaa ohjelman suoritusaikana ilmetä *virheenä* (*failure*) [IEE10, s. 5]. Virhe ilmenee, kun ohjelma ei suorituksen aikana toimi odotetulla tavalla.

Testausta käytetään ohjelmistokehityksessä ohjelmiston laadun varmistamiseen ja auttamaan lähdekoodissa esiintyvien vikojen havaitsemisessa jo kehitysvaiheen aikana. Ohjelmistojen testaamisen ensisijainen tavoite on siis paljastaa vikoja, joiden havaitseminen muiden laadunvarmistusmenetelmien avulla olisi työlästä tai mahdotonta [Bin99, s. 59]. Testauksen avulla pyritään myös 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].

3 Mutaatiotestaus

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 menetelmä testien laadun selvittämiseen on *mutaatiotestaus*. Mutaatiotestauksen avulla voidaan mitata, kuinka tehokkaasti ohjelmiston testeillä havaitaan ohjelmistossa esiintyviä vikoja [JH11, s. 649].

3.1 Historia ja teoreettinen perusta

Mutaatiotestauksesta kirjoitettiin ensimmäistä kertaa jo 1970-luvulla. Richard DeMillon, Richard Liptonin ja Frederick Saywardin artikkeli "*Hints on Test Data Selection: Help for the Practicing Programmer*" [DLS78] vuodelta 1978 on yksi ensimmäisistä uraauurtavista mutaatiotestausta esittelevistä artikkeleista. Mutaatiotestauksen tutkimus on lisääntynyt vuosien kuluessa, ja erityisesti 2000-luvulla uusia tutkimuksia ja tuloksia on julkaistu paljon [Off11, s. 1102]. Tutkimuksessa suuntana on ollut etsiä keinoja, joilla mutaatiotestaus voidaan muuttaa käytännölliseksi testausmenetelmäksi [JH11, s. 649].

Jefferson Offutt määritteli artikkelissaan yksinkertaiset ja monimutkaiset viat seuraavasti [Off92, s. 6]:

- Lähdekoodissa olevan *yksinkertaisen vian* voi korjata tekemällä yksittäisen muutoksen lähdekoodin lauseeseen.
- *Monimutkaista vikaa* ei voi korjata tekemällä yksittäistä muutosta lähdekoodin lauseeseen.

3.2 Perinteinen mutaatiotestausprosessi

Perinteisen mutaatiotestausprosessin syötteinä käytetään alkuperäistä ohjelmistoa ja ohjelmistoa testaavia testejä. Mutaatiotestausprosessin aikana olemassa olevien testien laatua kehitetään vaiheittain. Perinteisen mutaatiotestausprosessin työvaiheet on esitelty kuvassa

Perinteisessä mutaatiotestausprosessissa ensimmäinen työvaihe on käsitellä ohjelmiston alkuperäistä lähdekoodia *mutaatio-operaattoreilla*, jotka muuntavat koodia muodostaen siitä viallisia versioita [MHK06, s. 869]. Näitä viallisia ohjelmakoodin versioita kutsutaan mutanteiksi. Mutaatio-operaattorit kuvaavat algoritmeja, joiden avulla lähdekoodia käsitellään koodin muuntamisen aikana [OU01, s. 35].

Testien suorituksen jälkeen alkuperäiselle lähdekoodille ja mutanteille suoritettujen testien tuloksia verrataan toisiinsa. Testituloksia verrattaessa 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.

Esimerkki 1.

```
public class Kauppa {
   private String osoite;
   public void setOsoite(String uusiOsoite){
      this.osoite = uusiOsoite;
   };
   public String getOsoite(){
      return this.osoite;
   };
};

public class Elainkauppa extends Kauppa {
   private String osoite; //IHI-operaattorin lisaama peittava kentta
   private String erikoisala = "kissanruuat";
   public String toString(){
      return "Liikkeen osoite on " + getOsoite() +
             " ja sen erikoisalana on " + this.erikoisala;
   };
};
```

On todistettu, että ekvivalenttien mutanttien tunnistaminen algoritmisesti yleisessä tapauksessa on ratkaisematon ongelma [OMK06, s. 79]. Ongelma on kuitenkin herättänyt runsaasti teoreettista kiinnostusta, ja mahdollisia tunnistamistekniikoita on tutkittu paljon [JH11, s. 657]. Ekvivalenttien mutanttien tunnistamiseksi on kehitetty heuristiikkoja, joiden avulla tunnistamisongelma voidaan ratkaista osittain [OMK06, s. 79].

4 Yhteenveto

Olioperustaisen ohjelmoinnin mukana tulevien uusien haasteiden kohtaaminen vaatii muutoksia ohjelmistojen testausmenetelmiin. Sekä perinteisiä olemassa olevia että uusia testausmenetelmiä kehitetään, jotta olio-ohjelmia voidaan testata kattavasti ja laadukkaasti.

Perinteisessä ohjelmistotestauksessa keskitytään ohjelmiston toiminnallisuuden testaamiseen ja vikojen etsimiseen ohjelmistosta. Perinteiseen ohjelmistotestaukseen liittyy kuitenkin haasteita ja rajoituksia, jotka aiheuttavat testaukseen epävarmuutta. Yksi haasteista on määrittää, ovatko testauksessa käytettävät testit ja testausjärjestelmä riittävän luotettavia, jotta niiden avulla ohjelmistoa voidaan testata laadukkaasti.

Mutaatiotestaus on vikaperustainen testausmenetelmä, joka tarjoaa ratkaisun testien laadun määrittämiseen liittyvään haasteeseen. Mutaatiotestauksen avulla voidaan mitata ohjelmistoa varten tehtyjen testien kykyä havaita ohjelmistossa esiintyviä vikoja. Mutaatiotestausmenetelmän avulla on siis mahdollista kehittää olemassa olevia testejä ja parantaa testien laatua.

Mutaatiotestauksen käyttö yleisesti osana testausprosessia on kuitenkin vähäistä mutaatiotestaukseen liittyvien haasteiden ja ratkaisemattomien ongelmien takia. Mutaatiotestausta on tutkittu paljon, ja tutkimuksen tavoitteena on ollut etsiä ratkaisuja mutaatiotestaukseen liittyviin ongelmiin. Tutkimuksesta saatujen tietojen pohjalta ongelmiin on kehitetty osittaisia ratkaisuja, joiden avulla mutaatiotestausprosessi on mahdollista automatisoida lähes kokonaan.

Jotta mutaatiotestausmenetelmää voitaisiin tulevaisuudessa hyödyntää aiempaa enemmän tutkimuskäytön ulkopuolella, mutaatiotestauksen käyttöönoton helpottamiseksi olisi kehitettävä automatisoituja mutaatiotestaus-

järjestelmiä, joissa hyödynnetään mutaatiotestauksen ongelmien ratkaisumenetelmiä. Automatisoitujen mutaatiotestausjärjestelmien avulla mutaatiotestausmenetelmän laaja-alainen käyttö osana ohjelmistokehitystä voisi tulevaisuudessa olla mahdollista.

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.

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`.

IEE10 *IEEE Standard Classification for Software Anomalies*. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), sivut 1–23, Jan 2010.

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`.

Off92 Offutt, A. Jefferson: *Investigations of the Software Testing Coupling Effect*. ACM Trans. Softw. Eng. Methodol., 1(1):5–20, tammikuu

1992, ISSN 1049-331X. http://doi.acm.org/10.1145/125489.125473.

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.

OMK06 Offutt, Jeff, Ma, Yu Seung ja Kwon, Yong Rae: *The Class-level Mutants of MuJava*. Teoksessa *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, sivut 78–84, New York, NY, USA, 2006. ACM, ISBN 1-59593-408-1. http://doi.acm.org/10.1145/1138929.1138945.

OU01 Offutt, A. Jefferson ja Untch, Ronald H.: *Mutation Testing for the New Century*. luku Mutation 2000: Uniting the Orthogonal, sivut 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001, ISBN 0-7923-7323-5. http://dl.acm.org/citation.cfm?id=571305.571314.