

hyväksymispäivä

arvosana

arvostelija

Mutaatiotestaus oliojärjestelmissä

Eveliina Pakarinen

Tutkielma

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Helsinki, 8. marraskuuta 2015

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
Tutkielma		8. marraskuuta 2015	17
Tiivistelmä — Referat — Abstract			
<p>Olioperustaisen ohjelmoinnin kehityksen myötä testausmenetelmiä on sopeutettu uusiin vaatimuksiin, joita olio-ohjelmoinnin erityispiirteet ovat tuoneet mukanaan. Mutaatiotestaus on virheperustainen testausmenetelmä, jonka avulla ohjelmiston olemassa olevien testien laatua voidaan kehittää ja parantaa. Mutaatiotestaus esiteltiin ensimmäistä kertaa jo 1970-luvulla.</p> <p>Perinteisesti mutaatiotestausta on käytetty testien kehittämisessä proseduraalisella ohjelmoinnilla tuotetuille ohjelmille. Olioperustaisten ohjelmien mutaatiotestaukseen on kuitenkin kehitetty luokkamutaatioksi kutsuttu mutaatiotestausmenetelmä. Mutaatiotestaukseen liittyy ratkaisemattomia ongelmia, jotka estävät mutaatiotestauksen laajamittaisen käytön osana ohjelmistotestausta.</p> <p>ACM Computing Classification System (CCS):</p> <p>D.2.4 [Software/Program Verification]</p> <p>D.2.5 [Testing and Debugging]</p> <p>D.3.3 [Language Constructs and Features]</p>			
Avainsanat — Nyckelord — Keywords			
mutaatiotestaus, oliojärjestelmät, Java			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — Övriga uppgifter — Additional information			

Sisältö

1	Johdanto	1
2	Testaus oliojärjestelmissä	2
2.1	Rooli	2
2.2	Tasot	2
2.3	Testien suunnittelu	3
2.4	Rajoitukset	4
3	Mutaatiotestaus	5
3.1	Historia ja periaatteet/teoreettinen perusta	5
3.2	Perinteinen mutaatiotestausprosessi	6
3.3	Uusi mutaatiotestausprosessi	8
4	Mutaatiotestaus oliojärjestelmissä	9
4.1	Mutaatiotestauksen piirteet	9
4.2	Mutaatio-operaattorit	9
4.3	Automatisoidut mutaatiotestausjärjestelmät	11
5	Mutaatiotestauksen haasteet	12
5.1	Ekvivalentit mutantit	12
5.2	Tehokkuusongelmat	13
5.3	Manuaalinen työ	14
6	Yhteenveto	15
	Lähteet	16

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.

Ohjelmistokehitysprosessissa testausta voidaan käyttää ohjelmistossa olevien virheiden havaitsemiseen jo kehitysvaiheen aikana. Testauksen avulla voidaan myös parantaa ohjelmiston laatua ja varmistaa, että ohjelma toimii sille asetettujen vaatimusten mukaisesti.

Testaukseen liittyy kuitenkin myös rajoituksia. Testauksen avulla ei esimerkiksi voi aina todistaa ohjelmiston oikeellisuutta [Bin99, s. 58]. Lisäksi epävarmuutta liittyy käytettävän testausjärjestelmän oikeellisuuden ja luotettavuuden varmistamiseen.

Yksi menetelmä ohjelmiston testien laadun tutkimiseen ja parantamiseen on mutaatiotestauksen käyttö osana ohjelmiston testausprosessia. Mutaatiotestauksessa tavoitteena on tutkia, ovatko ohjelmistoa varten tehdyt testit laadukkaita ja havaitaanko niillä kattavasti ohjelmistossa mahdollisesti esiintyvät virheet ja ongelmat [JH11, s. 649].

Mutaatiotestauksen periaatteena on simuloida ohjelmoijien tekemiä yleisiä ohjelmointivirheitä muokkaamalla ohjelmiston alkuperäistä lähdekoodia [JH11, s. 649]. Mutaatiotestauksessa lähdekoodista muodostetaan muunnettuja versioita eli mutantteja.

Perinteisesti mutaatiotestausta on käytetty testien kehityksessä proseduraalisella ohjelmoinnilla tuotetuille ohjelmille. Olioperustaisen ohjelmoinnin kehittyessä mutaatiotestausta on sopeutettu uusiin vaatimuksiin. Luokkamutaation avulla mutaatiotestausta voidaan soveltaa olio-ohjelmissa testien laadun varmistamiseen [KCM00].

Vaikka mutaatiotestausta voidaan käyttää apuna ohjelmiston olemassa olevien testien kehittämisessä, liittyy mutaatiotestausmenetelmän käyttöön myös ratkaisemattomia ongelmia, jotka estävät mutaatiotestauksen laaja-alaisen käytön [JH11, s. 652]. Mutaatiotestauksen suoritus vaatii paljon laskentatehoa, mikä on yksi ongelmista. Lisäksi mutaatiotestausprosessissa

ihmiseltä vaadittava työpanos on suuri.

Mutaatiotestausta on tutkittu paljon [JH11, s. 649]. Tutkimuksen avulla on etsitty keinoja ratkaista mutaatiotestaukseen liittyviä ongelmia, jotta mutaatiotestaus voidaan muuttaa käytännölliseksi testausmenetelmäksi.

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 Rooli

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 myös varmistamaan, että ohjelma toimii sille asetettujen vaatimusten mukaisesti.

Olio-ohjelmoinnissa testaukseen tuovat haasteita olio-ohjelmien erityispiirteet, joita ovat muun muassa kapselointi, perintä, dynaaminen sidonta ja polymorfismi [MP08, s. 86]. **Lisää tähän? Binderin kirjasta löytyy tietoa.**

2.2 Tasot

Ohjelmistoja voidaan testata usealla tasolla. Testauksen tasoja ovat yksikkö-, integraatio- ja järjestelmätasot. Tasot 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*) testataan ohjelman pienimpiä suoritettavissa olevia komponentteja [Bin99, s. 45]. Olio-ohjelmissa näitä komponentteja ovat yksittäiset metodit ja oliot. **Lisää tietoa yksikkötestauksesta.**

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

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

2.3 Testien suunnittelu

Testien suunnitteluun ja kehittämiseen voidaan käyttää useita menetelmiä. Testausmenetelmän avulla kuvataan näkökulmaa, josta esimerkiksi ohjelman lähdekoodia tarkastellaan testejä kehitettäessä [Bin99, s. 51]. Ohjelman sisäisen rakenteen tai ulkoisen toiminnallisuuden tuntemusta voidaan käyttää testausmenetelmissä apuna uusia testejä kehitettäessä. Lisäksi olemassa olevia testejä voidaan kehittää testausmenetelmien avulla.

Ohjelman sisäisen rakenteen eli lähdekoodin tuntemukseen perustuvaa testausmenetelmää kutsutaan *white box -testaukseksi* [Bin99, s. 52]. **Lisää tästä.** 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* kutsutussa testausmenetelmässä testejä suunnitellaan ohjelmiston toiminnallisuuden tuntemuksen avulla [Bin99, s. 52]. **Lisää tuosta.** Koska valmiin sovelluksen piirteitä testattaessa tutkitaan myös sovelluksen ulkoista toiminnallisuutta,

on black box -testausmenetelmästä apua suunniteltaessa testejä esimerkiksi järjestelmätestaukseen.

Gray box - eli *hybriditestauksessa* yhdistetään white box - ja black box -testausmenetelmien piirteitä [Bin99, s. 52]. Sekä white box -testausta että black box -testausta voidaan 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 mutaatio-testaus, jonka avulla tutkitaan testien kykyä havaita ohjelmistossa olevia virheitä [DLS78, s. 36]. Testaukseen liittyvät tavoitteet eroavat mutaatio-testauksessa ja perinteisessä ohjelmistotestauksessa toisistaan. Perinteisessä testauksessa keskitytään parantamaan ohjelmiston laatua, kun taas mutaatiotestauksessa kehityksen kohteena ovat olemassa olevat testit ja niiden laatu.

2.4 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]. Oikeellisuuden todistamiseen liittyen Edsger Dijkstra totesi: *"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 vertailukohdaksi luotettavia odotettuja tuloksia. Tällöin testauksesta saatujen toteutuneiden tulosten tulkinta ja arviointi on epävarmaa. Toteutuneista tuloksista ei voi siis **aina (?)** luotettavasti päätellä, menivätkö testit läpi vai eivät [Bin99, s. 58].

Epävarmuutta liittyy myös testattavan järjestelmän halutulle toiminnal-

lisuudelle 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].

Rajoituksista selvennettävä noita 2 viimeistä kappaletta. Niissä voi olla jotakin epäselvyyksiä. Ainakin mulla vielä on joten vois selventää niitä hivin vuoksi.

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 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 Historia ja periaatteet/teoreettinen perusta

Mutaatiotestauksesta kirjoitettiin ensimmäistä kertaa jo 1970-luvulla. Richard DeMillon, Richard Liptonin ja Frederick Saywardin artikkeli [DLS78] vuodelta 1978 on yksi ensimmäisistä urauurtavista 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].

historiasta ehkä lisää ja tutkimuksesta

Mutaatiotestaus perustuu kahteen hypoteesiin: the Competent Programmer Hypothesis ja the Coupling Effect (Hypothesis).

Lisää juttua siitä, että mutaatiotestaus kasvattaa testitkattavuutta.

3.2 Perinteinen mutaatiotestausprosessi

Ohjelmisto ja ohjelmistoa varten tehdyt testit ovat perinteisessä mutaatiotestausprosessissa prosessin lähtökohtia. Perinteisen mutaatiotestauksen aikana olemassa olevien testien laatua kehitetään vaiheittain. Perinteisen mutaatiotestausprosessin työvaiheet on esitelty kuvassa 1. Työvaiheet on merkitty kuvaan sinisillä **laatikoilla**. Työvaiheiden aikana käytettäviä resursseja, **kuten esimerkiksi** ohjelmiston lähdekoodia ja testituloksia, kuvataan vihreillä ja keltaisilla laatikoilla. **Mihin kuva sijoitetaan?**

Perinteisessä 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 muuntamattomalle lähdekoodille [JH11, s. 652]. Tavoitteena on varmistaa, että testit voidaan suorittaa alkuperäiselle ohjelmalle ilman virheitä. Jos testit eivät mene läpi eli ohjelmassa havaitaan virheitä, virheet korjataan ennen kuin mutaatiotestausta jatketaan.

Seuraavaksi testit suoritetaan jokaisen elävän mutantin kohdalla [OU01, s. 35]. Kun testit suoritetaan mutanteille, tavoitteena on selvittää, havaitaanko testien avulla lähdekoodiin tehdyt muutokset.

Testien suorituksen jälkeen alkuperäiselle lähdekoodille ja mutanteille suoritettujen testien 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 kohdal-

la. Tämä tarkoittaa, että mutantti on tapettu eli lähdekoodiin tehty muutos on havaittu [DLS78, s. 36].

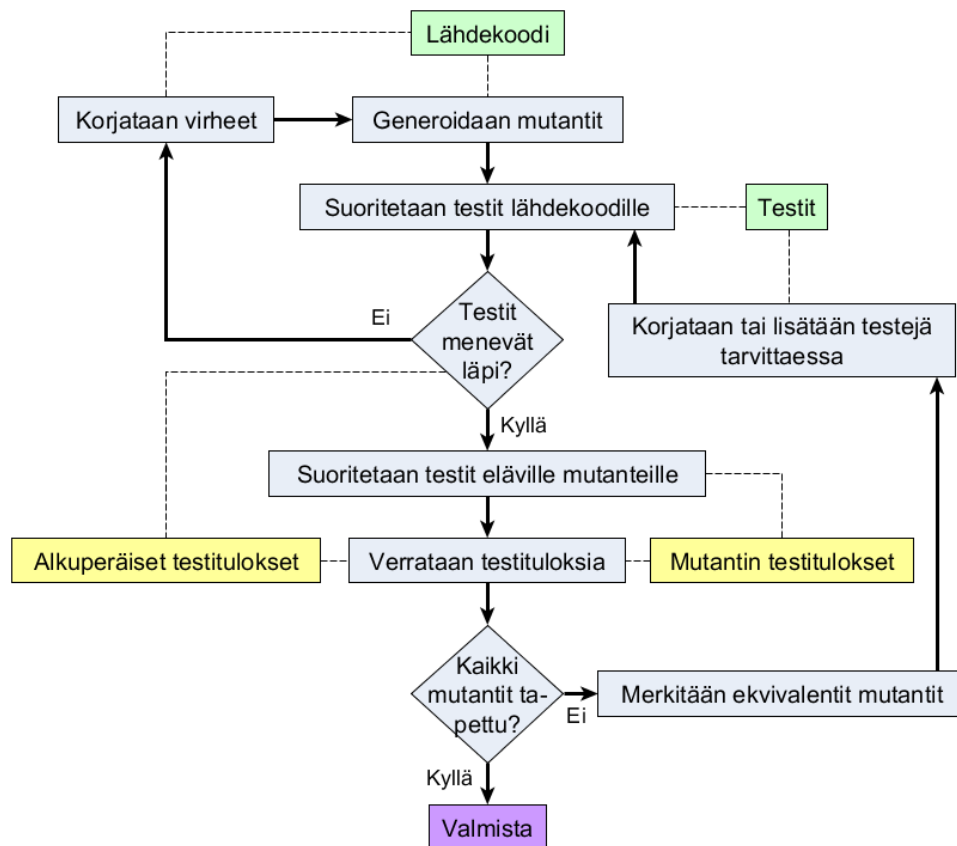
Tapauksessa 2. alkuperäiset testit ovat menneet läpi mutantin kohdalla eli mutantti on jäänyt eloon. Mutantin jäämiselle eloon on kaksi vaihtoehtoista selitystä [DLS78, s. 36]. Ensimmäinen selitys on, että alkuperäiset testit eivät ole riittävän hyvät, jotta niiden avulla voidaan havaita lähdekoodiin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

Jos testituloksia verrattaessa huomataan, että mutantteja on jäänyt eloon, seuraava vaihe mutaatiotestausprosessissa on etsiä ja merkitä ekvivalentit mutantit [OU01, s. 36]. Tämän jälkeen testien laatua voidaan kehittää joko muokkaamalla olemassa olevia testejä tai lisäämällä uusia testejä, jotta **eloon jääneet (yhteen/erikseen?)** mutantit saadaan tapettua.

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

$$MP = \frac{T}{K - E} \quad (1)$$

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 [OU01, s. 36]. Testit ovat tällöin *adequate relative to the mutants* riittävät havaitsemaan kaikki mutantit. Perinteinen mutaatiotestausprosessi päättyy, kun kaikki *ei-ekvivalentit (miten tämä kirjoitetaan?)* mutantit on tapettu. Tämä on merkitty kuvaan 1 violetilla laatikolla.



Kuva 1: Perinteinen mutaatiotestausprosessi.

3.3 Uusi mutaatiotestausprosessi

Onko tämä hyvä omana kappaleenaan vai voiko tai pitääkö laittaa perinteisen prosessin loppuun? Omaa muotoilua perustuen **Uniting orthogonal 2000** artikkeliin → uusi mutaatiotestausprosessi: Kuvasta 1 huomataan, että perinteinen mutaatiotestausprosessi sisältää silmukoita. Ensimmäinen silmukka sisältää mutanttien generoinnin, testien suorituksen alkuperäiselle lähdekoodille ja testien tulosten tarkastamisen jälkeen virheiden korjauksen. Näistä työvaiheista alkuperäisten testitulosten tarkastaminen ja alkuperäisen ohjelman virheiden korjaaminen vaativat manuaalista työtä.

Toinen silmukka eli mutaatiotestauksen pääsilmukka muodostuu työvai-

heista, jotka suoritetaan alkuperäisten testitulosten läpimenotarkastuksen jälkeen. Toiseen silmukkaan sisältyvät myös ensimmäisen silmukan työvaiheista testien suoritus lähdekoodille ja testien läpimenotarkastus. Tämä tarkoittaa, että silmukat risteävät näiden työvaiheiden kohdalla. Toisessa silmukassa manuaalista työtä vaativat ekvivalenttien mutanttien etsiminen ja merkitseminen ja uusien testien lisääminen tai olemassa olevien testien kehittäminen.

En tiedä, onko tämä uusi prosessi jo käytössä? Jotta perinteistä mutaatiotestausprosessia voidaan tehostaa, manuaaliset työvaiheet on poistettava mutaatiotestauksen pääsilmukasta [OU01, s. 41]. Tällöin perinteisen mutaatiotestausprosessin tilalle muodostuu uusi testausprosessi, jossa osa työläistä manuaalisista työvaiheista suoritetaan automaattisesti.

Automatisoitavia työvaiheita uudessa mutaatiotestausprosessissa ovat ekvivalenttien mutanttien tunnistaminen ja testien generointi ja kehittäminen. Ainoaksi merkittäväksi manuaaliseksi työvaiheeksi jää tarkistaa, ovatko alkuperäiselle lähdekoodille suoritettut testit menneet läpi. Jotta alkuperäisen lähdekoodin testitulosten tarkastusvaihe saadaan pois mutaatiotestauksen pääsilmukasta, suoritetaan läpimenotarkastus uudessa mutaatiotestausprosessissa pääsilmukasta poistuttaessa. Työvaiheiden automatisointi ja niiden suoritussyjärjestyksen muuttaminen tekevät uudesta mutaatiotestausprosessista tehokkaamman verrattuna perinteiseen mutaatiotestausprosessiin [OU01, s. 41].

4 Mutaatiotestaus oliojärjestelmissä

Kappaleen estittelyteksti tähän. **Kuinka pitkä tämän on oltava?**

4.1 Mutaatiotestauksen piirteet

MuJava-artikkelista -> inter-intra/class-method testaus.

4.2 Mutaatio-operaattorit

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.

Olioperustaisten ohjelmien mutaatiotestaukseen käytettävää menetelmää kutsutaan *luokkamutaatioksi* [KCM00]. Luokkamutaatiomenetelmän kehittivät Sunwoo Kim, John Clark ja John McDermid Java-ohjelmointikielessä esiintyvien virheiden pohjalta [KCM00]. 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. Yu Seung Ma, Yong Rae Kwon ja Jeff 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 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. Taulukosta nähdään myös, mihin ryhmiin luokkamutaatio-

operaattorit kuuluvat. Tarkemmat tiedot operaattorien toiminnasta löytyvät **Man, Kwonin ja Offuttin** artikkelista ”Inter-class mutation operators for Java” [MKO02].

Ryhmä	Operaattori	Kuvaus
Kapselointi	AMC	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	<i>super</i> keyword deletion
	IPC	Explicit call of a parent’s constructor deletion
Polymorfismi	PNC	<i>new</i> 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 kuormitus	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAQ	Argument order change
	OAN	Argument number change
Javan erityispiirteet	JTD	<i>this</i> keyword deletion
	JSC	<i>static</i> 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 [MKO02].

4.3 Automatisoidut mutaatiotestausjärjestelmät

Mutaatiojärjestelmät opetuksessa: PIT.

Mutaatiojärjestelmä tutkimuksessa: MuJava.

5 Mutaatiotestauksen haasteet

Vaikka mutaatiotestausta voidaan käyttää ohjelmiston olemassa olevien testien kehittämiseen ja testien laadun parantamiseen, ei mutaatiotestausmenetelmän käyttö ole ongelmaton. Mutaatiotestaukseen liittyy haasteita, jotka estävät menetelmän käyttämisen laaja-alaisesti osana testausprosessia [JH11, s. 652].

5.1 Ekvivalentit mutantit

Ekvivalenttien mutanttien tunnistaminen on yksi mutaatiotestaukseen liittyvistä ongelma-alueista. Ekvivalentit mutantit ovat mutantteja, jotka ovat toiminnaltaan samanlaisia kuin alkuperäinen ohjelma, mutta syntaktisesti erilaisia alkuperäisen ohjelman kanssa [JH11, s. 652]. Ekvivalentteja mutantteja muodostuu, kun alkuperäistä lähdekoodia käsitellään mutaatio-operaattoreilla.

Taulukossa 1 esitelty mutaatio-operaattori IHI lisää perivään aliluokkaan piilottavan muuttujan määrittelyn (**inserts hiding variable declaration**). Jos piilotettu muuttuja on ylliluokassa määritelty *private*-määreellä (**private access level**), kaikki IHI-operaattorilla tuotetut mutantit ovat ekvivalentteja mutantteja [OMK06, s. 80]. Esimerkissä 1 IHI-operaattorilla on muodostettu mutantti, jossa *Kauppa*-luokan aliluokaan *Elainkauppa* on lisätty piilottava muuttuja *osoite*. *Elainkauppa*-luokassa ylliluokan *osoite*-muuttuja saadaan käyttöön ylliluokan metodilla *getOsoite()*. Piilottavan muuttujan *osoite* lisääminen ei siis muuta luokan *Elainkauppa* toimintaa, joten kyseessä on ekvivalentti mutantti.

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 muuttuja
    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]. Ekvivalenttien mutanttien tunnistamiseksi on kuitenkin kehitetty heuristiikkoja, joiden avulla ongelma voidaan ratkaista osittain. Tunnistamisongelma on herättänyt runsaasti teoreettista kiinnostusta, ja mahdollisia tunnistamistekniikoita on tutkittu paljon [JH11, s. 657].

Yksi vaihtoehto ekvivalenttien mutanttien tunnistamisongelman ratkaisemiseksi on estää ekvivalenttien mutanttien syntyminen mutanttien generointivaiheessa [OMK06, s. 80]. **Muita vaihtoehtoja ovat Compiler optimization techniques, algorithms in Mothra, infeasible constraints ideaan perustuva väline, Program slicing.**

Heuristiikkojen (ja laskennan) avulla voidaan tunnistaa vain osa ekvivalenteista mutanteista. Jos kaikki ekvivalentit mutantit halutaan tunnistaa, tunnistaminen on suoritettava manuaalisesti tutkimalla mutanttia ja vertaamalla mutantin toimintaa alkuperäisen ohjelmiston toimintaan. Tämä menetelmä johtaa mutaatiotestauksen toteuttamiseen tarvittavan työmäärän kasvuun.

5.2 Tehokkuusongelmat

Tehokkuuteen liittyvät ongelmat ovat toinen mutaatiotestauksen ongelma-alueista. 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.

Ratkaisuehdotukset mutaatiotestauksen vaatiman laskentatehon (**computational cost**) vähentämiseksi on jaettu kolmeen osa-alueeseen [OU01, s. 37]. Osa-alueisiin sisältyy menetelmiä, joiden avulla mutaatiotestauksen laskentakustannuksia pienennetään. Mutaatiotestauksessa laskentaa voidaan tehostaa tekemällä mutaatiotestausta *viisaammin* (*do smarter*), *nopeammin* (*do faster*) ja *pienemmällä mutanttimäärällä* (*do fewer*) kuin ennen/aiemmin.

Do smarter laskentatyön jakaminen usealle laitteelle tai weak mutation.

Do faster mutanttien generoinnin/kääntämisen ja suorituksen tehostaminen, jotta voitaisiin suorittaa niin nopeasti kuin mahdollista. MSG metodi (MuJava) tai compiler-integrated program mutation tai separate compilation.

Do fewer vähemmän mutantteja ilman "mutaatiokattavuuden" kärsimistä suuresti. Mutant sampling, selective mutation,

5.3 Manuaalinen työ

Kolmas mutaatiotestauksen ongelma-alue liittyy perinteiseen mutaatiotestausprosessiin sisältyvään manuaaliseen työhön. Perinteisessä mutaatiotestausprosessissa useaan prosessin työvaiheeseen sisältyy manuaalista työtä (**tätä käsittelem kappaleessa 3.3 Uusi mutaatiotestausprosessi**) → **pitääkö se käsitellä uudestaan tässä kohtaa?**. Kuvassa 1 (sivulla 8) on esitelty perinteinen mutaatiotestausprosessi. Kuvassa esiteltyyn kaavioon on merkitty työvaihe, jossa alkuperäisten testien suorituksen jälkeen tarkastetaan, ovatko testit menneet läpi.

Testien läpimenotarkastus 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.

Uusien, mutantteja tappavien testien kehittäminen on myös yksi perin-

teisen mutaatiotestausprosessin työläistä manuaalisista työvaiheista [OU01, s. 39]. Mutaatiotestauksessa tavoitteena on kehittää testejä, joiden avulla saadaan tapettua mahdollisimman paljon mutantteja (mutation adequate). Manuaalisesti tällaisten testien kehittäminen on työlästä. Lisäksi sopivien syötteiden löytäminen testejä varten on yksi vaikeimmista testauksen vaiheista [OU01, s. 39]. Testien automaattista generointia varten on kuitenkin kehitetty menetelmiä, joiden avulla testien generointi voidaan mutaatiotestausprosessissa automatisoida osittain.

6 Yhteenveto

Olioperustaisen ohjelmoinnin mukana tulevien uusien haasteiden kohtaaminen vaatii muutoksia myös 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 virheiden etsimiseen ohjelmistosta. Perinteiseen ohjelmistotestaukseen liittyy kuitenkin haasteita, 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 virheperustainen testausmenetelmä, joka tarjoaa ratkaisun testien laadun määrittämiseen liittyvään haasteeseen. Mutaatiotestauksen avulla voidaan mitata ohjelmistoa varten tehtyjen testien kykyä havaita ohjelmistossa esiintyviä virheitä. Mutaatiotestausmenetelmän avulla on siis mahdollista kehittää olemassa olevia testejä ja parantaa testien luotettavuutta.

Mutaatiotestauksen käyttö yleisesti osana testausprosessia on vähäistä mutaatiotestaukseen liittyvien ratkaisemattomien ongelmien takia. Mutaatiotestausta on kuitenkin tutkittu paljon. Tutkimuksen avulla voi olla mahdollista löytää ratkaisuja mutaatiotestausta vaivaaviin ongelmiin, jotta mutaatiotestauksen laaja-alainen käyttö 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.
- 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>.
- 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>.