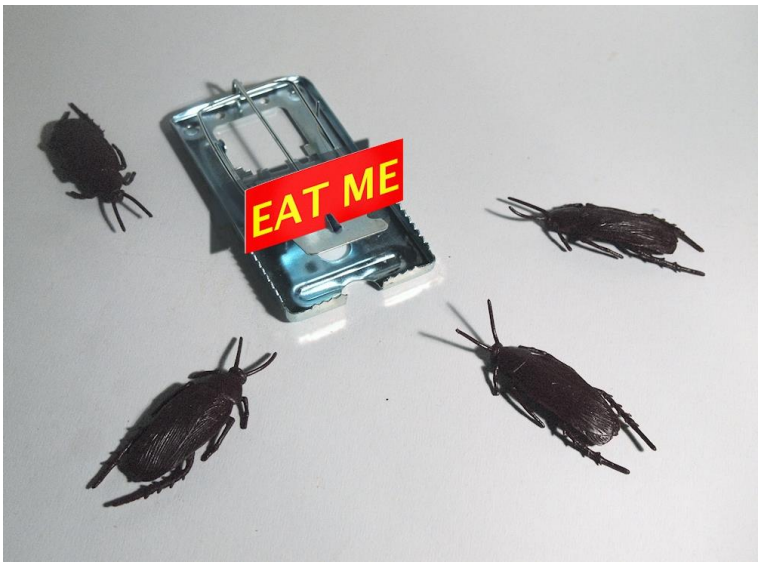
LAATU JA TESTAUS

Testitapaukset – kyllä vai ei

1/2014



TestausOSY
Finnish Association of Software Testing

JULKAISIJA

Testauksen osaamisyhteisö (TestausOSY)

TOIMITUSKUNTA

Tuula Pääkkönen

Agnes Nummi

Erkki Pöyhönen

JULKAISUPAIKKA

Verkko: http://www.testausosy.fi

JULKAISUTIHEYS

Kolmesti vuodessa noin neljän kuukauden välein

Lehti on vapaaehtoisvoimin laadittu. Artikkelien oikeudet kuuluvat kirjoittajille.

Visuaalinen ilme: Maaret Pyhäjärvi

Tekninen taittotyö: Matti Vuori

Kansikuva: Matti Vuori

Sisällysluettelo

3. Pääkirjoitus. *Tuula Pääkkönen.*
4. 90 ajatusta testitapauksista. *Matti Vuori*.
11. Versionvaihtoa, videoita ja virheilmoituksia. *Tommi Räihä & Riku Rytkönen*.
14. Jotta Tšernobyl ei unohtuisi. *Heikki Virtanen*.
16. Testien automaattinen generointi – kun aika ja mielikuvitus eivät riitä. *Antti Kervinen*.

Pääkirjoitus

Tuula Pääkkönen, TestausOSY-aktiivi

Kun Laatu- ja Testaus-lehden toimituskunta kokoontui syksyn alussa pohtimaan miten edettäisiin lehden teon kanssa, yhtenä aiemmin toivottuna aiheena esille nousi testitapaukset testaamisessa. Sitä jäimme miettimään, mikä nykyään tilanne testitapausten kanssa on, ja aiheuttavatko ne intohimoja puoleen ja toiseen vai ovatko niin perusinfrastruktuuria testaamisessa, että niitä ei enää tule edes ajatelleeksi? Vai onko niin, että testitapauksia tehdään, kun niitä on aina tehty, jolloin niitä luodaan, päivitetään ja ajetaan vakiotoimintomallina jatkuvasti.

Aihe kuitenkin näytti kiinostavan ja tässä lehdessä tuleekin erilaisia näkökulmia asiaan. Käymme läpi monia erilaisia ajatusta testitapauksista – testitapauksen määritelmästä , historiaan ja merkitykseen ja aina mallipohjaiseen testaukseen saakka.

Urheiluanalogiaa hyödyntävässä artikkelissa pohditaan mitä tapahtuu ottelutilanteesa kun projektin budjetti on sidottu aikatauluun ja laatuun, kuka voittaa?

Muistatko kevään jolloin Tšernobylin ydinvoimalaonnettomuus tapahtui? Sinä keväänä uutiset idästä pysäyttivät pienemmänkin kulkijan pohtimaan mikä kaikki voikaan maailmassa mennä pieleen, kun kaikki meneekin pieleen. Pohdintoja mitä ko. onnettomuudesta voisi tai olisi pitänyt oppia ja kuinka testitapaukset liittyvät tähän tapaukseen löydät lehden sivuilta myös.

Kun mielikuvitus loppuu silloin voi käyttää testien generointia laajentamaan testaamisen näkemystä, mahdollisesti antaen uusia tapoja myös käydä testattavaa järjestelmää kattavammin läpi.

Kiitoksia kaikille kirjoittajille laajasta näkökulmien kirjosta! Laatu- ja Testaus-lehden tekijät ja kirjoittajat toivovat ajatuksia herättäviä lukuhetkiä.

Tuula Pääkkönen on toiminut IT-alalla sekä yksityisellä, että julkisella puolella nykyään yli 16 vuotta. Hän on työskennellyt ohjelmistokehityksen eri rooleissa, kehittäjästä, testaajaan. Näissä rooleissa hän on pohtinut kuinka asiat voisi tehdä paremmin, mm. pohtien testausprosesseja, testauksen automaatiota, mallipohjaista testausta ja ketteriä menetelmiä.

90 ajatusta testitapauksista

Matti Vuori, Tampereen teknillinen yliopisto, Tietotekniikan laitos

Testitapaus on klassisin testien määrittelyn konstruktio ja niitä suunnitellaan ja suoritetaan joka hetki valtavia määriä. Jossain mielessä testitapaukset ovat joskus maailman tärkein asia ja toisaalta liiallinen keskittyminen niihin saattaa aiheuttaa monen muun aivan oleellisen asian ohittamisen. Testitapausten luonne on ehdottomasti miettimisen arvoinen. Luokaamme siis pieni katsaus niihin erilaisista näkökulmista.

Perusidea

1. Testitapauksessa etsitään jokin testattava asia, jonka toimintaa koetellaan jollakin täsmällisellä tavalla, usein ”syötteellä”. Syötteen antamisen jälkeen katsotaan, miten järjestelmä käyttäytyi ja oliko käyttäytyminen odotettua. Stereotyyppisesti vertaillaan todellista tulosta odotettuun.
2. Suoritus tehdään hallitussa tilanteessa ja siksi tapauksen määrittelyyn liittyykin kohteen sopiva alustus ja koko ympäristön saattaminen otolliseen tilaan.
3. Samoin sen suorituksen jälkeen systeemi palautetaan usein johonkin tilaan – esim. siivotaan pois luodut tiedostot – jotta muille testitapauksille on sopivan ”puhdas” alkutilanne.
4. Ja vaikka stereotyyppisesti katsellaan vain ”tuloksia”, käytännössä pitää katsella kaikkia kohteen reaktioita ja systeemin muutoksia ja havaintoja kokonaisuuden toiminnasta.
5. Jos testitapaus suoritetaan automaatiolla, havainnointi on vaikeampaa kuin silloin, jos testauksen tekee ihminen. Toisaalta automatiikka näkee paremmin systeemin sisäisiä asioita, esim. tapahtumia tietokannassa.
6. Se tapa, jolla testitapaus suoritetaan, on käsitteellisesti usein erotettu ”testiproseduuriksi”, mutta käytännössä esitellään suoritustekniikkaan sidottuja askeleita, joihin kaikkiin voi liittyä omia tarkistuksiaan.
7. Testitapauksen suoritustapa on siis ”vapaasti” valittavissa. Ajatus on hieno, koska sama tapaus voidaan laittaa ihmisen, instrumentoidun automaation tai vaikka robotin suoritettavaksi täsmälleen samanlaisena.
8. Klassisesti ajatellaankin, että testitapauksia suoritetaan ensin manuaalisesti ja sitten, kun niiden toistotarve ilmaantuu, testaus voidaan automatisoida.
9. Tämä on järkevää erityisesti regressiotestausta varten, mutta samalla tietysti menetetään läsnä olevan ihmisen havainnointikyvyn käytön mahdollisuus.

Testitapauksella on monta roolia

10. Testitapaus on luonnollisesti testisuunnittelun arkifakta ja sellaisena osa testiarkkitehtuuria.
11. Se on testaajan ajattelun ja tahdon ilmentymä.
12. Se on testauksen työmäärän yksikkö ja kustannustekijä.

13. Se on testauslogistiikan seurattava elementti. Niitä ajetaan testauksen kohteelle virtana, erilaisina paketteina.

Standardin määritelmä

14. Uusi standardi ISO/IEC/IEEE 29119-1[1] määrittää testitapauksen näin: “set of test case preconditions, inputs (including actions, where applicable), and expected results, developed to drive the execution of a test item to meet test objectives, including correct implementation, error identification, checking quality, and other valued information”.
15. ISTQB:n sanaston uusin versio[2] määrittää sen näin: “ test case: A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. [After IEEE 610].].”
16. ISTQB:n sanasto erottelee lisäksi kaksi testitapauksen abstraktiotasoa: a) high level test case: “A test case without concrete (implementation level) values for input data and expected results. Logical operators are used; instances of the actual values are not yet defined and/or available. See also low level test case.” Synonyymejä: abstract test case, logical test case. b) low level test case: “A test case with concrete (implementation level) values for input data and expected results. Logical operators from high level test cases are replaced by actual values that correspond to the objectives of the logical operators.” Synonyymi tälle on “concrete test case”.

Historia

17. En tiedä, kuka keksi käsitteen, mutta sitä käyttivät ainakin jo Herbert Leeds ja Gerald Weinberg vuonna 1961 kirjassaan ”Computer programming fundamentals”.
18. He eivät määritelleet testitapaukselle rakennetta tai muutakaan formaalia – vain sen mihin niillä pyritään ja testataan (normaalitilanteet, äärirajat, poikkeukset).
19. ”Määrittelyjen” aikakausi alkoi myöhemmin, kun pyrittiin tiukasti standardoimaan testitapaustenkin rakenne – ja yhdenmukaistamaan kaikki muukin testauksen määrittely ja dokumentointi.
20. Tämä työ tuotti alan gurujen kirjoissaan esittämiä kuvauksia ja myöhemmin IEEE 829:n ensimmäisen version[3] määrittelyt testitapaukselle ja testiproseduurille.
21. Standardoitu rakenne auttaa testitapausten kuvauksen harmonisointia organisaatioissa, niiden suunnittelun systematisointia, suunnittelun mahdollistamista kaikenlaisella työvoimalla, testitapausten auditoitavuutta ja potentiaalia niiden suorittamisen automatisointiin.
22. Koska testitapaus on ollut toiminnallisen testauksen ydintä ja toiminnallinen testaus on ollut dominoiva testityyppi, on selvää, että testitapauksilla on ollut erittäin vahva rooli testauskulttuurissa.
23. Kautta vuosikymmenten on tehty paksuja kirjoja tekniikoista, joilla testitapauksia suunnitellaan.
24. Testauksen opetuksessa ne ovat usein kurssien ensimmäisiä asioita.

Vahvoja merkityksiä

25. Testitapauksella on vahvoja merkityksiä ohjelmistokehityksen ja teollisuuden kulttuurissa.
26. Heti kun sana edes mainitaan, syntyy assosiaatio, että nyt otetaan laatu vakavasti ja on jo pohdittu tarkasti, miten systeemiä aiotaan koetella. Testitapaus on aivan eri asia kuin abstrakti ”testaus”.
27. Termiä käyttävä henkilö todistaa sillä oman ammattilaisuutensa.
28. Samoin termiin liittyy magiaa. Testitapauksen olemassaolo varmistaa laatua symbolisesti. Se on kuin Mustanaamion hyvä merkki ohjelmiston leuassa.
29. Ja koska se on hyvä merkki sinänsä ja hyvään liittyy kulttuurissamme se, että enemmän on parempi kuin vähemmän, mitä enemmän testitapauksia on, sitä paremmin asioiden koetaan olevan.

[1] ISO/IEC/IEEE 29119-1 Software and systems engineering -- Software testing -- Part 1: Concepts and definitions

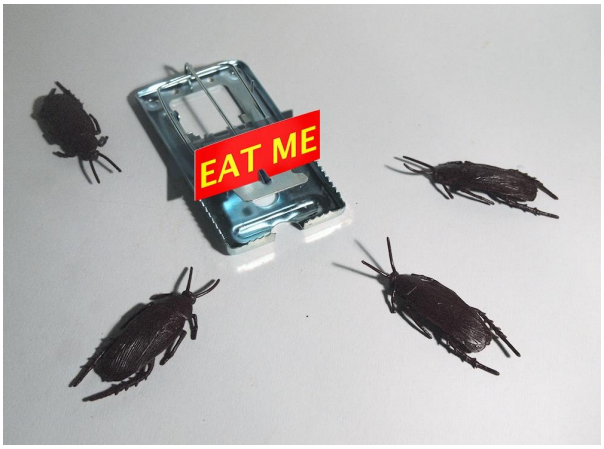
[2] Standard Glossary of Terms used in Software Testing. Version 2.4. International Software Testing Qualifications Board ISTQB. 50 p.

[3] IEEE 829-1983 - IEEE Standard for Software Test Documentation

30. Testitapauksella on siis kulttuurinen rooli monella tasolla: se on osa retoriikkaa, osa uskomusjärjestelmää ja osa arkista toimintaa.
31. Jos laatukulttuuri ei organisaatiossa ole vahvaa, voi puheella hämätä itseään ja toisia.
32. Mutta puhe oikeilla termeillä tukee laatukulttuurin kehittymistä.

Yksin ja joukossa

33. Testitapauksella on yksilöllinen luonne. Jokainen niistä tekee aivan tiettyä asiaa, jolla on jokin selkeä tarkoitus.
34. Mutta samaan aikaan ne ovat kuin parvi, jonka voima tulee siitä, että pieniä asioita on kenties valtava määrä.
35. Testitapaukset ovat kuin ansoja, joihin odotetaan bugien astuvan.



36. Tämä on erilainen taktiikka kuin tutkivassa testauksessa, jossa testaaja on kuin villipetoa aktiivisesti jäljittävävä metsästäjä.
37. Sellainenkin metsästäjä voi jättää regressioloukun sopivaan paikkaan, jos peto vaikka palaisi aiempaan ruokapaikkaansa...
38. Ja tietenkin, jos metsää ansoitetaan, kannattaa ensin tehdä ansoja isoimmille saaliseläimille – pienempiä pyydetään sitten myöhemmin.

Kaikella on hintansa

39. Jokaisen testitapauksen suunnittelu, toteutus, ylläpito ja suorittaminen maksaa.
40. Siksi kannattaa laatia vain sellaisia testitapauksia, jotka tuottavat relevanttia tietoa tai muuta hyötyä.
41. Joskus se kustannus tulee ajasta, sillä aika on rahaa. Testitapausten suorittaminen ja palautteen saaminen on tärkeää olla nopeaa, että asiat etenevät.

Käytöstä mittarina

42. Testitapausten läpäisyn tai läpäisemättömyyden on arveltu olevan hyvä laadun mittari.
43. Tässä on perää silloin, kun jokainen testitapaus uniikisti testaa jotain relevanttia asiaa relevantilla tavalla, mikä ei ole missään nimessä varmaa.
44. Yksikkötestauksen tasolla testitapauksilla on historiallisesti enemmän relevanssia kuin ylemmillä testaustasoilla, koska matalalla tasolla ei oikeasti vielä ”tiedetä”, millaisen todellisen virheen läpäisemätön testitapaus aiheuttaisi käyttäjän tai prosessien toiminnassa.
45. Mutta samalla tämä koettu merkitys on tuottanut patologioita, joissa testitapausten määrää on pidetty testauksen laadun mittarina, mitä on nykyisin opittu kritisoimaan.

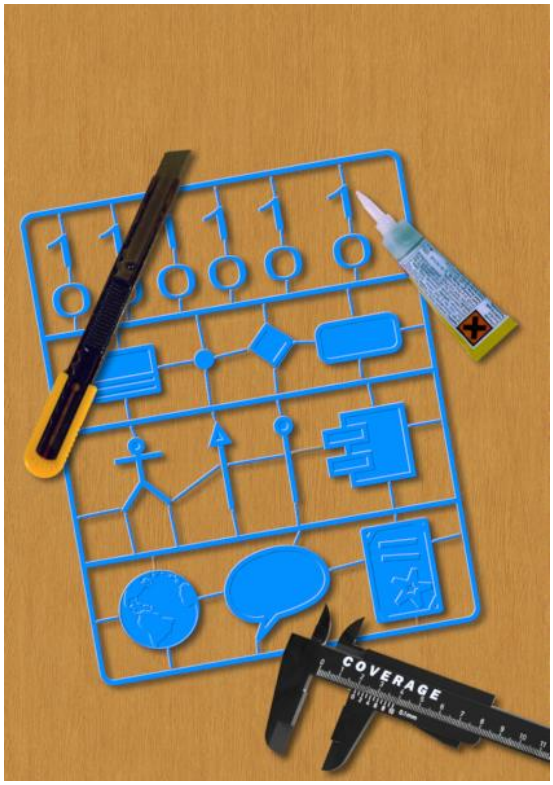
Luonteen kritiikkiä

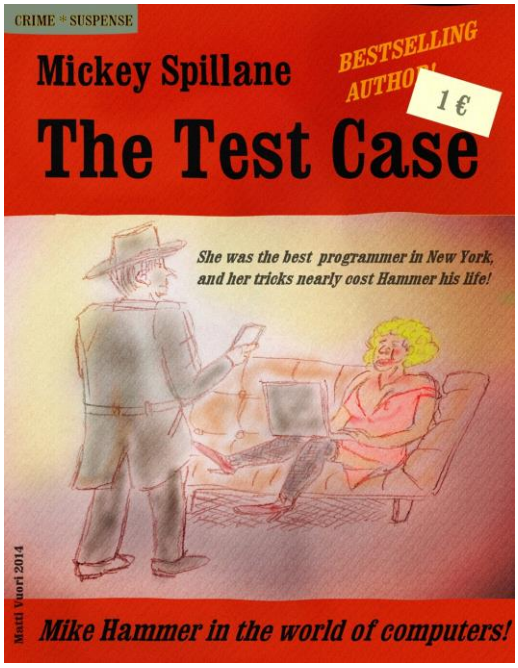
46. Testitapaukset ovat kuin maalilaukaukset jääkiekossa ja niiden harjoittelu. Ollaan ikään kuin kontekstivapaasti sopivassa paikassa maalin edessä ja pyritään laukaisemaan kiekko maalivahdin ohi valitsemalla se "syöte" (eli mihin ammutaan) sopivasti. Mutta maailma ei aina ole sellainen, eikä maali ole niin selkeä. Ja jääkiekostakin on 1960-luvun jälkeen ymmärretty paljon muuta kuin tuo elementti.

47. Niinpä nykyään korostetaankin joukkuepeliä, pelitapaa jne. keskeisempänä. Niiden elementtien avulla päästään useimmin siihen tilanteeseen, että kiekko saadaan maaliin lähes luonnostaan. Kun luodaan hyvä pelitilanne (skenaario, käyttötapaus, tarina – testiympäristössä) ja tarkka silmä avoimille paikoille (ymmärrys siitä mitä testataan, test conditions), oikea laukaus ("ratkaiseva syöte") löytyy lähes suunnittelematta, ikään kuin edellä kehitellyn väistämättömänä jatkumona. Kyse ei silloin ole laukaisutaidosta, vaan osaamisesta olla oikeaan aikaan, oikeassa valmiustilassa maalin edessä paikassa, johon kiekko tulee.

Maailmankuvan ikoni

48. Testitapausten maailmankuva on analyyttinen. Siinä kaikki voidaan jakaa ja palastella aina vain pienempiin kohteisiin ja toiminta kohdistaa löydettyyn jakamattomaan asiaan. Tehdään systeemin jäsennystä, jonka perusteella sen piirteet jaetaan. Niistä tunnistetaan kohteita, joita voisi testaaja ja relevantteja testattavia asioita. Funktionaalisille kohteille mietitään sopivia syötteitä ja tapoja suorittaa se syötteen syöttäminen. Alimmalla tasolla tehdään kaikki mahdollinen ja sitten luodaan katsaus ylempään kokonaisuuden tasoon.
49. Joskus tämä maailmankuva on niin vahva, että





ajatellaan, että mitään muuta ei tarvita laadun näkemiseen kuin testitapausten tarjoamia fragmentteja toiminnasta. Se on sama kuin kehittäisi tiimiä vain kehittämällä yksilöitä, kiinnittämättä huomiota tiimin toimintaan tiiminä.
50. Testitapausten ajattelutapa on myös idealistinen. Niissä näkyy teknologia puhtaana ja ilman ulkoisia vaikutuksia. Tapahtuu ajattelun puhdistumista mennessä alimmille testaustasoille, joissa vain puhdas suorituslogiikka on tarkastelun kohteena.
51. Puhtauteen liittyy myös se, että periaatteessa testiproseduuri on erotettu testitapauksesta.
52. Systeeminen ajattelu tulee toki esille testauksen kokonaisuudessa, jossa ylemmillä testaustasoilla otetaan koko järjestelmän toiminta huomioon ja realismia tuodaan mukaan joka kerroksella huipentuen hyväksymistestauksen ja järjestelmän validoinnin näkökulmaan, jossa olemassa oleva todellisuus on systeemin toiminnan referenssi.
53. Maailmankuvaan liittyy myös deterministisyyden ajatus. Kaikki on päätettyä ja harkittua. Sattumalla ei ole sijaa ja se pyritään karsimaan pois testauksestakin.

Estetiikasta

54. Edellä mainitut puhtaus ja ideaalisuus ovat selkeitä testitapauksen esteettisiä periaatteita.

55. Yksinkertaisuus on myös tärkeää. Yksinkertainen testitapaus, jolla on selkeä tarkoitus on kuin hyvä työkalu, hyvä kirves tai puukko.
56. Toisaalta testitapauksiin liittyy myös rumuutta. Kun testitapauksessa ollaan erityisen kovakouraisia – esimerkiksi ”vedetään töpseli irti”, se ei ole kaunista, vaan melkein hirvittää testaajaakin. Toisaalta silloin on jo mukana väistämättömyyden ja kohtalon kauneutta.
57. Näistä esteettisistä periaatteista on paljon todellista hyötyä, sillä ne luovat testitapauksille tavoitteellisuutta, tehokkuutta, ylläpidettävyyttä ja auditoitavuutta.

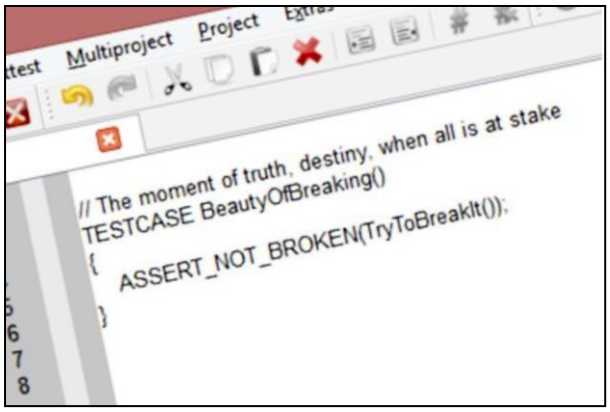
The case of the case

58. Termissä on kaksi osaa: testi ja tapaus. Tapaus on maailmassamme monessa käytössä: oikeustapaus, sairaustapaus, tapaustutkimus, käyttötapaus.
59. ”Tapaus” on yhdentyyppisen asian yksilöllinen ilmentymä, joka muussa käytössä paketoi kaiken ilmentymään liittyvän: tilanteen, ongelman, tiedot, kohteen, ratkaisun, mukana olevat toimijat, historian.
60. Se on kuin pienoismaailma, jonka puitteissa asia otetaan esille, käsitellään ja ratkaistaan.
61. Yhden tapauksen puitteissa ei käsitellä muita tapauksia.
62. Käsitteen avulla saadaan siis asia hienosti hallittavaksi.
63. Toisaalta se antaa asioista liian hyvin paketoidun kuvan: jo yksikin testitapaus luo assosiaation, että sen ratkaisu on jotain oleellista. Tämä näkyy mm. tilanteissa, joissa vaaditaan, että jokaista vaatimusta varten pitäisi olla ainakin yksi testitapaus. Yksikin niistä ”riittää”!
64. Voisiko miettiä muita käsitteitä?
65. Tilastollisesta ja diagnostisesta näkökulmasta jokainen testitapaus on vain näyte. Siis testinäyte, laatunäyte, näyte käyttäytymisestä.
66. Se käsite antaa ymmärtää, että yksi näyte on vain yksi monista, vaihtoehtoinen muille, eikä yksinään kerro vielä mitään. Sen avulla ei voida luoda minkäänlaista varmuutta systeemin toiminnasta tai laadusta.
67. Näytteenotto on kuitenkin ”observoivaa” – siinä vain tarkkaillaan asioita, eikä pyritä vaikuttamaan kohteeseen. Puuttuu hyvän testitapauksen ”tappamisen meininki”…
68. Entä testitarina? Elämmehän aikaa, jolloin suositaan tarinoita niiden kertovuuden ja rikkauden ja vuoksi. Tarina olisi kuitenkin selvästi jotain testitapausta suurempaa – pieni skenaario jossain testipisteessä. Mutta sehän kuulostaisi hyvin mielekkäältä.
69. Vähän tätä lähellä ovat skenaariot ja käyttötapausten testaus, mutta ne ovat toiminnan maailmasta lähteviä termejä.

Fokusmekanismeja eri testityypeissä

70. Testitapaukset ovat tyypillisiä systemaattisessa toiminnallisuustestauksessa. Siinä testaus huipentuu siihen, että ihminen suunnittelemalla löytää syötteen, jonka sitten syöttää testauksen kohteeseen ja tarkistaa mitä tapahtui.
71. Siihen päästään luomalla jäsennys jostakin näkökulmasta (mm. toiminta, rakenne, story/työnkulku, tietovirta, vaatimukset) ja tunnistamalla niiden kautta sopiva ”testipiste”, josta päästään tarttumaan kohteeseen – vaikkapa tiedon syöttö lomakkeelle. Siitä on helppo tunnistaa testattavia asioita ja suunnitella testitapauksia ja niiden syötteitä.
72. Kohdetta havainnoidaan ennen ja jälkeen ja havaintojen "erotus" kertoo siitä, mitä on tapahtunut. Erityisesti pyritään usein keksimään syötteitä vastaava ”odotettu tulos”, jonka vertailu todelliseen on hyvä mittari testin onnistumiselle.
73. Tutkivassa toiminnallisuustestauksessa tehdään samanlaista jäsennystä, mutta testausta ohjaa suoraviivaisten testipisteiden sijaan jokin strategia kohteen käyttäytymisen selvittämiseen. Sen puitteissa tehdään dynaamisesti ohjattua kokeilua, saadaan siitä havaintoja ja iteroidaan, kunnes käyttäytyminen on kylliksi ymmärretty.
74. Kokeilu on lähellä testitapausten ”just-in-time”-suunnittelua, mutta siitä puuttuu mentaalista painolasti odotettujen havaintojen ja tilanteen kontrollin suhteen.

75. Suorituskykytestaus ja kuormitustestaus ovat luonteeltaan tästä erilaisia. Testauksen fokuksena ei ole mikrotapahtuma, vaan laajempi skenaario – käytön volyymi, ympäristö, taustakuorma, tilanne kokonaisuutena. Siihen kontekstiin suunnitellaan mielenkiinnon (tiedontarpeen) kohteena olevia käyttötapauksia (tms.), jotka suoritetaan hallitusti ja samalla tehdään mittauksia. Mittaustuloksia ei hyväksytä suoraan, vaan ne vaativat analyysiä ja tulkintaa.
76. Mallipohjainen testaus lähtee testattavan käyttäytymisen rajauksesta ja testimallin laadinnasta. Usein tässä testauksessa ei ole testitapauksia, vaan ohjelmaa suoritetaan kaikki relevantteja suorituspolkuja pitkin ja suoritusta vertaillaan mallin antamaan ”oikean” käyttäytymisen referenssiin.



77. Mallipohjainen testaus on yksi esimerkki testauksesta, jossa myös sattumalla on sijaa – sekä yhden systeemin toiminnassa että systeemien vuorovaikutuksessa.
78. Toinen tapa tuoda testaukseen satunnaisuutta on ”monkey”-testaus, jossa ohjelmaa käytetään käyttöliittymästä täysin satunnaisella tavalla. Siihen maailmaan ei testitapausten käsite istu luontevasti.
79. Fuzz-testaus soveltaa myös usein satunnaisuutta testidatan muuntelussa. Onko jokainen muunnos testitapaus? Ei ole aivan mielekkäintä ajatella niin. Muuntelutaktiikka on tärkein alimman tason määrittely.
80. Tietoturvatestaus alkaa riskianalyysillä ja sen osana uhkamallinnuksella. Konkreettista testausta tukee hyökkäysvektorien tunnistaminen. Niiden avulla voidaan tunnistaa testipisteitä ja tehdä testausta toiminnallisuustestauksen logiikalla.
81. Tietenkin tietoturvatestauksen osana – tai vahvasti synergisenä parina – on esimerkiksi käytettävyystestaus, koska käyttäjän näkökulma ja virheet ovat kriittinen alue tietoturvassa.
82. Käytettävyystestauksessa lähdetään yleensä laajemmilta vesiltä. Testattavana on ihminen-teknologia-järjestelmän toiminnan jokin tavoitteellinen skenaario, jonka annetaan kul-

kea ihmisen tahdon mukaisesti. Tärkein läsnä oleva ihminen on koehenkilö, joka tekee tehtäväänsä sen tavoitteisiin pyrkien yrittämättä tehdä mitään erityisen testaavaa. Testisuunnittelija miettii etukäteen skenaariot ja luo olosuhteet sellaisiksi, että mahdollisiin ongelmiin tullaan tekemisen kuluessa törmäämään. Skenaarioita suoritetaan suhteellisen kontrolloiduissa oloissa, tehdään havaintoja koehenkilön ”pärjäämisestä” ja niitä analysoidaan.

83. A/B-testauksen lähtökohtana on jokin käyttäjätarina, jonka puitteissa tutkitaan eri toteutusvaihtoehtojen paremmuutta. Keskeistä on siis muutaman vaihtoehtojen suunnittelu ja mielekkään testiasetelman laatiminen – missä tärkeää on miettiminen, millaisilla mittauksilla paremmuutta mitataan. Eri vaihtoehtojen tarjoaminen eri populaatioille ohjelmiston julkaisulla on testin valmistelun ydintä. Data hankitaan mittaamalla versioiden käyttöä ja sitä analysoidaan testin aikana ja jälkeen.
84. Näissä kaikissa näkyy kuitenkin samanlainen jatkumo: testattavan tilanteen skouppaus, miettiminen mitä arvoa systeemiltä odotetaan, fokusoituminen keskeiseen, olosuhteiden luominen, jonkinlainen tilanteen toteuttaminen ja havaintojen tekeminen sekä havaintojen arviointi.

Johtopäätöksiä

85. Testitapaus on erinomainen käsitteellinen ja käytännöllinen väline varsinkin ”tekniseen” testaukseen, jossa kohdetta ”pommitetaan” erilaisilla syötteiden variaatioilla.
86. Sillä on myös keskeinen asema kulttuurisena konstruktiona, mihin liittyy riskejä organisaatioissa, joissa laatukulttuuri ei ole kypsää.
87. Liiallinen asioiden näkeminen testitapauksina on vaarallista – erilaiset asiat tarvitsevat erilaisia käsitteitä.
88. Testauksen maailmassa on paljon testityyppejä, joissa testitapauksen käsite ei ole mielekäs.
89. Ja aivan sama haaste on muussakin testauksen kehittämisessä... toiminnallisuustestauksen kulttuurihistoria ei riitä enää toiminnan ohjenuoraksi.
90. Kannattaa miettiä ja kehitellä käsitteistöä edelleen, jotta se muuttuvassa maailmassa pysyy mahdollisimman ilmaisuvoimaisen.

Matti Vuori on toiminut laatuun ja toiminnan kehittämiseen liittyvien asioiden parissa jo neljällä vuosikymmenellä tutkijana, konsulttina, kouluttajana ja opettajana ja ehtinyt nähdä ja miettiä monia erilaisia vastakkainasettelun lajeja. Nykyisin hän on töissä Tampereen teknillisellä yliopistolla.

Versionvaihtoa, videoita ja virheilmoituksia

Tommi Räihä & Riku Rytkönen, Sogeti Finland

Asiakkaalla ei ole dokumentoituja testitapauksia tai testitapaukset on dokumentoitu puutteellisesti tai epäjohdonmukaisesti. Edessä on kriittisen ERP-järjestelmän versionpäivitys ja pitää olla varma heti Go Live -päivästä lähtien, että keskeiset prosessit toimivat moitteettomasti.

Aikaa on annettu nelisen kuukautta varmistaa asiakkaamme, keskisuuren yrityksen ydintoiminnot toimintavarmoiksi. Budjettimme on sidottu aikatauluun ja laatuun: ei virheitä tuotannossa kriittisissä prosesseissa. Toimijoina on asiakkaan henkilökunnan lisäksi päivitettävän järjestelmän toimittaja, asiakkaan järjestelmiin integroidut asiakkaat ja heidän järjestelmien toimittajansa.

Ai niin, kesälomatkin osuvat keskelle projektia jokaisella toimijalla – ja sitten olisi tarkoitus testata ERP:in suorituskyky päivitysten jälkeen...

Alusta alkaen oli erittäin ilmeistä, että jotain pitää tehdä testausprojektin nopeuttamiseksi.

Aloitus myöhästyy jonkin verran – aika tavallinen tilanne

Kun projektia päästään aloittamaan ollaan jo myöhässä suunnitellusta aloituksesta jonkin verran: järjestelmän toimittajalla on joitakin aikatauluhaasteita. Siitä huolimatta suunnitellusta Go Live -päivästä päätetään pitää kiinni, koska muussa tapauksessa koko projektin toteutusta olisi lykättävä yli puolella vuodella. Sama aikataulurajoitus koskee testaustakin.

Tehtävän annon aikoihin oli arvioitu, että yksikkötestitapauksia tulisi tallennettavaksi n. 100 kpl. Näistä sitten koostetaan varsinaiset testitapaukset versionvaihdon mukanaan tuomien mahdollisten poikkeamien havaitsemiseksi. Yksikkötestien määrä nousi lopulta yli kahdensadan ja niistä hyödynnettiin kriittisiä end-to-end -prosesseja testatessa noin 60 %.

Asiakkaan henkilökunnan rajallinen osallistuminen projektiin oli jo alun perin sopimuksessamme mainittu asia. Sogetin piti tehdä testien suunnittelu, regressiotestisetin määrittely ja huolehtia testauksen toteutuksesta, pistää pystyyn SAP Solution Manager (SolMan).

Pääkäyttäjien osallistuminen oli tarkoitus pitääkin minimissään, koska liiketoiminta eli se ”oikea työ” piti myös tehdä ja lisäksi kesäloma järven rannalla oli jo jokaisella mielessä. Siis aloitetaan tyhjästä ja paljon on tekemistä, eikä asiakasta haluta sitoa tekemiseen mukaan liikaa. Aikaa käytettävissä neljä kuukautta, suunnitellaan miten luodaan testitapaukset, etsitään virheet ja testataan niiden korjaukset.

Ottelutilanne:

Projekti – Sogeti 1:0

Aikaa on vähän – otetaan avuksi nauhoitus

Aikataulun kireys sai aikaan ajatuksen prosessien dokumentoinnista nauhoittamalla tapahtumat käyttäjän ruudulta. Näin toimien asiakkaan käyttäjien ei tarvitse antaa ajastaan kuin sen verran kuin kuluu tavanomaisten työtehtävien tekemiseen.

Lähes kaikki yksikkötestitapaukset videoitiin asiakkaan henkilökunnan tehdessä normaaleja töitään. Näin päästiin aikaa ja vaivaa vaativasta ”vierihoidosta”, eli asiakkaan pääkäyttäjien ei tarvinnut olla vieressä neuvomassa sogetilaisia mitä nappia painetaan missäkin prosessin kohdassa. Prosessit ajettiin pääosin tuotannossa, jolloin saatiin todenmu-

kaista dataa ja nähtiin miten kunkin prosessin todella kuuluu pyörähtää.

Nauhoitettujen prosessikuvausten perusteella Sogetin testitiimi kirjoitti testitapaukset alan parhaiden standardien mukaiseen muotoon. Tällä tavalla rahaa ja aikaa säästyi niin asiakkaalta kuin meiltäkin: WIN-WIN.

Näillä työkaluilla tultiin toimeen:

- SAP ERP, asiakkaan liiketoimintaprosessit
- Wink 2.0, Open Source -sovellus prosessien nauhoitusta varten
- Ulkoinen kovalevy
- SAP ger 7.1, testitapausten ja videoiden tallennus
- MS Word, testitapausten kirjoittaminen
- Älypuhelimia 3G-tukiasemiksi, asiakkaalla ei nimittäin ollut vierasverkkoa

Työkalu Wink itsessään on helppo ja nopea ottaa käyttöön. Sen konfigurointi esim. SAP-prosesseja tallentamaan on yksinkertaista ja johdonmukaista jopa yritys ja erehdys -periaatteellakin.

Wink on siis avoimeen lähdekoodiin perustuva työkalu, joka kaappaa ruutukuvia halutun määrän sekunnissa ja muodostaa niistä videon. Näistä nauhoituksista kirjoitettiin yksikkötestitapauksia, jotka tallennettiin SolManiin. Samoin tallennettiin itse nauhoitukset tulevaa tarvetta varten SolManiin Business Blueprint -hakemiston kansioihin.

Regressiotestitapauksien koostaminen oli lopulta kohtuullisen helppoa, koska jo nauhoitusvaiheessa kriittiset prosessit oli käyty pääkäyttäjien kanssa läpi. Kun testitapauksia kirjoitti nauhoitusten perusteella templaten mukaiseen muotoon, vain kirjoitustyön lopputuloksen tarkistusvaiheessa tarvittiin apua tai kommentteja pääkäyttäjiltä.

Ottelutilanne:

Projekti – Sogeti 2:1

Määritellään SolMan – alusta alkaen

Sitten kun kaikki alkoi rullata nauhoitusten ja kirjoittamisen osalta, oli myös aika aloittaa SolManin tarvitsevat toimenpiteet. SolMan piti määritellä asiakkaalle käytettäväksi testaustyökaluna ja testeissä havaittujen poikkeamien raportointia ja hallintaa varten.

Asiakasorganisaatiossa ei aiemmin ollut käytetty SolMania eikä muutakaan testien hallintatyökalua, joten koulutusmateriaalit ja käyttäjäkoulutukset oli myös hoidettava, ennen kuin testaus voisi alkaa. Tässäkin vaiheessa tuli haasteita vastaan, joista mielenkiintoisin taisi olla Internet Explorerin versio asiakkaan koneilla. SolMan ei tukenut kuin versiota 9 tai vanhempia, joten SolMan käyttäjiltä piti alentaa versiota asiakkaan koneissa.

Testiosion käyttöön oton muut vaiheet sujuivat joustavasti yhteistyössä järjestelmän toimittajan kanssa. Käyttöön ei saatu tällä pikaisella aikataululla viestien välityspalvelua, vaan SolManiin kirjatut testien virheilmoitukset välitettiin sähköpostilla korjaustiimille.

Kirjoitetaan testitapauksia

Aikataulun mukaisesti testitapaukset piti olla tunnistettuina ja dokumentoituina kuukautta ennen testien alkamista. Se tarkoitti, että kesä kului kirjoittaen testitapauksia. Asiakkaan ei tarvinnut osallistua paljoakaan kirjoittamiseen, ellei prosessi ollut niin haastava, että nauhoitteiden pohjalta prosessin kulun hahmottaminen ei onnistunut. Tavanomaiset liiketoimintaprosessit kuten osto, myynti, varastonhallinta ja lähetystoiminnot oli mahdollista kirjata nauhoitusten perusteella yksikkötesteiksi.

Kun yksikkötestit olivat valmiina ja tallennettuina SolManiin ne katselmoitiin asiakaan pääkäyttäjien kanssa ja havaitut epäselvyydet korjattiin. Tässä vaiheessa asiakkaalle alkoi muodostua selväksi, mitä SAP AG tarkoittaa sloganillaan SolMan on ”Single Source of Truth”.

Seuraavaksi oli end-to-end tapausten määrittelyn vuoro ja myös tässä vaiheessa toimivaksi avuksi olivat nauhoitetut prosessit: muistin virkistäminen, kokonaisuuksien hahmottaminen, testidatan datan valikoiminen, jne.

Ottelutilanne:

Projekti – Sogeti 2:2

Ja lopuksi päästään testaamaan – ”EXECUTE, F8”

Aikataulussa oli annettu kolme viikkoa liiketoimintojen testaukseen ja viikko sitä ennen perustietojen toimintojen ja niihin liittyvien integraatioiden varmistamiselle. Virheilmoituksien kirjanpito haluttiin SolManiin, koska asiakkaamme aikaisemmissa projekteissa oli havaittu tarvetta sellaiselle toimintatavalle, sähköpostilla ja excelillä hallitut virheilmoi-

tukset olivat hukkuneet tai jääneet muiden postien sekaan.

Testauksessakin palattiin katsomaan toisinaan nauhoituksia, eli miten prosessin oikeastaan kuuluisi mennä. Ulkopuolinen testaaja ei nimittäin ollut sittenkään onnistunut painamaan mieleen kaikkia detaljeja asiakkaan prosessien koukeroista kolmen kuukauden aikana. Siis kun muistin virkistystä tarvittiin aika ajoin, silloin ladattiin SolManiin tallennettu video ja taas päästiin eteenpäin.

Kun ilmoituksia poikkeamista alkoi tulla ja järjestelmäntoimittaja ratkoi niitä ahkerasti, alkoi valjeta asiakkaalle järjestelmästään piirteitä, joita ei siinä oikeastaan pitänyt ollakaan. Järjestelmän todelliset ominaisuudet ilmenivät systemaattisella testauksella. Koska kirjoitetut testitapaukset perustuivat nauhoituksiin tuotantojärjestelmästä, pystyi niihin luottamaan. Testitapausten todistusarvo oli siten huomattavasti tukevammalla pohjalla, kuin jos ne olisivat perustuneet ulkopuolisen testiammattilaisen tekemiin haastattelumuistiinpanoihin muutaman kuukauden takaa.

Ilmoitetuista poikkeamista paljastui yllättävästi yli puolet sellaisiksi, jotka lopulta päätyisivät jatkossa muutospyynnöiksi järjestelmäntoimittajalle. Asiakkaan tietoisuus järjestelmästään kohosi ja järjestelmän toimittaja saa todennäköisesti muutospyynnöistä lisää toimeksiantoja.

Versionvaihto sujui kuulemamme mukaan ongelmitta, nauhoitukset ovat edelleen tallessa ja ”aiheettomia” virheilmoituksia muokataan muutospyynnöiksi.

Ottelutilanne:

Projekti – Sogeti 2:3

Tommi Räihä on testipäällikkö ja Riku Rytkönen on testauskonsultti Sogeti Finland Oy:ssä

Jotta Tšernobyl ei unohtuisi

Heikki Virtanen

Parikymmentä vuotta sitten tekoälystä tarinoinut dosentti sen sanoi. Ongelma ei ole tekoäly vaan sen oikea käyttö. Sama pätee testitapauksiin. Niitä voi käyttää väärin ja vääriin tarkoituksiin siinä missä mitä tahansa teknologiaa. Oikein käytettynä testitapaukset ovat korvaamaton apu, ja ilman niitä olisi hyvin vaikea kehittää ohjelmistoja.

Lauantai 26. huhtikuuta 1986, Prypjat, Ukraina

Läheisessä Tšernobylin ydinvoimalassa testattiin. Haluttiin kokeilla, pystyykö sammumassa olevan ydinreaktorin hidastuva generaattori tuottamaan sähköä jäähdytysjärjestelmälle niin kauan, että varajärjestelmät saadaan päälle.

Valitettavasti sen paremmin kokeen suunnittelijat kuin sitä suorittavat operaattoritkaan eivät tunteneet testikohteen eli RBMK reaktorin oikkuja. Koetta varten osa reaktorin turvajärjestelmistä piti kytkeä pois päältä ja ajaa sitä aliteholla. Jo pelkkä reaktorin käyttö aliteholla oli ehdottomasti kielletty reaktorityypin epävakauden vuoksi. Niinpä alle minuutti kokeen alkamisen jälkeen reaktorin teho oli noussut kymmenkertaiseksi normaaliin verrattuna tunnetuin seurauksin. (Lähde: fi.wikipedia.org:Tšernobylin_ydinvoimalaonnettomuus)

Länsimainen ihminen toki ihmettelee, miten reaktorin operaattorit suostuivat aloittamaan tuollaisen kokeen. Tšernobylin ydinvoimala tuotti sähkön lisäksi raaka-aineita ydinaseisiin ja oli siten ainakin osittain sotilaallisen toimintakulttuurin alainen. Silloisessa Neuvostoliitossa se tarkoitti, että oli terveellisempää tehdä turhia kyselemättä ja ihmettelemättä mitä käskettiin.

Testitapausten helmi

Yksi tyypillinen keino löytää virhe ohjelmasta on tehdä sen suoritus tavalla tai toisella näkyväksi ja seurata, mitä polkuja suoritus etenee ja mitä ovat parametrien ja muuttujien arvot matkan varrella.

Piisami on järjestelmällinen simpukoiden kerääjä, mutta sen aterian tähteistä on turha helmiä etsiä. Järvisimpukkaan niitä ei muodostu. (Kuva: Heikki Virtanen)

Tämä tekniikka vaatii tuekseen sellaisen testitapauksen, joka on riittävän yksinkertainen suorittaa ja joka pystyy luotettavasti ja toistettavasti osoittamaan, että ohjelmassa on virhe.

Tällainen testitapaus on kaunis kuin matemaattinen teoreema. Se sisältää järjestelmän elinkaaren kannalta olennaisen väitteen ja todistuksen sille. Jos testitapaus on vielä asiallisesti dokumentoitu, sen yhteydessä on kerrottu, miksi testitapauksen väite on tärkeä ja mitä seurannaisvaikutuksia sillä on.

Ei ole itsestään selvää, miten hyviä testitapauksia löytäisi tehokkaimmin. Järjestelmän käyttäjältä tulleen virhetoimintaraportin pohjalta pystyy melkoisen varmasti johtamaan yhden tai useamman virheen tunnistavan testitapauksen. Tosin järjestelmän tekijöiden kannalta on melkoisen noloa, että viallinen tuote on päätynyt käyttäjälle asti.

Järjestelmällisesti muodostettujen testitapausten pitäisi puolestaan olla, vaan ei käytännössä aina ole, tehottomia, koska ne ovat järjestelmän tekijöiden ennakoitavissa. Oletettavasti siis testaus on jonkin muun kuin käyttäjän tehtävä, sillä pitää olla

tarveperustainen yhteys järjestelmän tulevaan elinkaareen ja käyttöön ja testaajien tarvitsee tuntea, kuten Tšernobyl osoitti, järjestelmää myös sisältä, sen rakennetta ja toiminnan filosofiaa, joskus melkoisen yksityiskohtaisestikin.

Testitapaukset ja testaaminen

Tšernobylissa tehty koe on ääriesimerkki testitapauksesta, missä on kaikki katastrofin ainekset. Samantyylisiä virheitä tehdään testattaessa kuitenkin päivittäin, mutta testausympäristöt ovat onneksi sen verran kesyjä, ettei seuraukset ole yhtä huomiota herättäviä.

Kuinka usein taitamattomat teettävät haluttomilla testausta, jolla ei ole tuotteen laadun kannalta merkitystä? Kuinka usein testausta tehdään sellaisella tavalla ja sellaisessa ympäristössä, etteivät tulokset ole päteviä aidossa toimintaympäristössä? Kuinka usein tuijotetaan pelkkiä testauksen tunnuslukuja miettimättä, mitä oikeastaan testataan tai mitä testissä tulee tapahtumaan?

Aika usein, jos testaus tehdään pelkästään testitapausten kautta. Testitapauksia on helppo muodostaa systemaattisesti ajattelematta, mitä sen suoritus reaalimaailmassa tarkoittaa. Testitapaus kattaa hyvin rajallisen kokonaisuuden ja liittyy johonkin kontekstiin, ja aika harvoin on mahdollista ottaa huomioon edes muutamaa tapaa, joilla kyseinen konteksti normaalissa käyttötilanteessa saavutetaan. Lisäksi suuri osa testauksen tunnusluvuista on sellaisia, että ne saadaan näyttämään hyvältä lisäämällä testisettiin sopivia testitapauksia, mutta vaikeampaa on löytää asia, joka tunnuslukujen vuoksi lisätyn testitapauksen pitäisi tarkistaa.

Testitapaus on siis hyvä dokumentti kuvaamaan löydettyä virhettä. Testitapausten suunnittelu ennalta taas on melkoista hakuammuntaa. Se on kuin elintarviketurvallisuusviraston tekemät pistokokeet, joilla yritetään saada kiinni ruokaväärennöksiä markkinoille tuuppaavat koijarit. Mutta softan virheet eivät ole tarkoituksellisia vaan johtuvat suurimmaksi osaksi väärinkäsityksistä, joten ne eivät ole samalla tavalla satunnaisiakaan kuin virheet tuotteen valmistusprosessissa, jonka jäljiltä jokainen valmis tuoteyksilö pitää ja voidaan tarkastaa erikseen.

Haaveilua

Todennäköisesti Tšernobylin tapahtumien seurauksena toiminta- ja turvallisuuskulttuuri on kehittynyt merkittävästi entisen Neuvostoliiton ja sen liittolaisten alueilla. Kaikki ohjelmistokehittäjät ja managerit eivät kuitenkaan reagoi osoitettuun virheeseen yhtä rakentavasti. Virhe ohjelmassa merkitsee heille turhauttavaa lisätyötä ja hidastetta ohjelman julkaisun tiellä eikä merkkiä tarpeesta muuttaa omia ja organisaation työtapoja. Tämä näkyy esimerkiksi siinä, että tuleville testausammattilaisille kannattaa opettaa ne samat tekniikat testitapausten laatimiseen ja valintaan, mitkä on edellisillekin testaajien sukupolville opetettu.

Sen verran ohjelmistotekniikan opetuksessa ollaan kuitenkin menossa eteenpäin, että testaamisen alkeet ja yksikkötestauksen testitapausten etsiminen on tuotu osaksi ohjelmoinnin kursseja (Esimerkiksi TTY, Tietotekniikka: "Ohjelmoinnin tekniikat"). Näin varsinaisessa testauksen opetuksessa jää enemmän tilaa kehittyneempien testausmenetelmien ja testauksen organisoinnin opettamiselle.

Testauksen tuominen lähemmäksi ohjelmistokehittäjiä on kuitenkin vasta hyvä alku, koska laadun tekeminen testauksen kautta tuntuu kömpelöltä ja työläältä. Jos testaaja löytää järjestelmästä korjaamista vaativan ongelman, hänellä on sellaista tietoa, jonka avulla olisi pystynyt löytämään oikean ratkaisun jo ensimmäisellä yrittämällä. Vaadittua, käytäntöön soveltuvaa tekniikkaa tämän tiedon kokoamiseen, esittämiseen ja käsittelemiseen ei kuitenkaan ole vielä olemassa, joten vaaditaan paremmasta huomisesta haaveilevia ihmisiä ja paljon tutkimusta, ennenkuin perinteiset systemaattisen keinot etsiä ja rakentaa testitapauksia ovat muuttuneet tehottomiksi.

Keskustelua aiheesta voi jatkaa osoitteessa https://gist.github.com/hessuvi/0231b0f731077e6f7f99.

Heikki Virtanen on sielultaan laatua rakastava ohjelmistokehittäjä ja -tieteilijä, joka tuli testaajayhteisön jäseneksi keittiön kautta. Erääseen tutkimusryhmään tarvittiin testimallien teoriaa, työkaluohjelmistojen rakentamista ja Linux-palvelimen ylläpitoa taitava vahvistus.

Testien automaattinen generointi – kun aika ja mielikuvitus eivät riitä

Antti Kervinen, Intel

Laitteita ja ohjelmistoja voi käyttää hyvin monin eri tavoin, mutta automaattiset testit kattavat niistä vain murto-osan. Tästä syystä monet virheet livahtavat testien ohi käyttäjien kiusaksi.

Automaattisten testien kattavuuden parantaminen tarkoittaa monen testaajan ja managerin mielessä enemmän ja monimutkaisempia testiskriptejä. Toisin sanoen lisää kehitys- ja ylläpitotyötä. Mutta voisiko suuri resurssien tarve johtuakin siitä, että varsinaista ongelmaa – testien alhaista kattavuutta – yritetään ratkaista väärällä tavalla?

Esittelen tässä artikkelissa ratkaisun, jossa testiautomaatio huolehtii erilaisten käyttötapojen löytämisestä ja kattamisesta. Se on testien automaattinen generointi. Jotta lukijat voisivat kokeilla sitä käytännössä, annan muutamia fMBT-työkalulla toimivia esimerkkejä. fMBT on monipuolinen avoimen lähdekoodin testigeneraattori.

Yksinkertaiset testiskriptit eivät riitä pitkälle

Otan esimerkiksi matkalipunvarausjärjestelmän. Käyttäjä valitsee verkkopalvelusta sellaisen matkustuspäivän ja vuoron, johon on vielä paikkoja vapaana, valitsee mieleisen paikan ja maksaa lipun verkkopankissa. Tällaisen testiskriptin kirjoittaminen on yksinkertaista - ja niin on sellaisenkin, missä käyttäjä päättää peruuttaa tilauksen siinä vaiheessa, kun pitäisi siirtyä verkkomaksuun.

Testiskriptin kirjoittaminen vaikeutuu huomattavasti, jos halutaan testata peruuttamista mistä tahansa ostosketjun vaiheesta muuttamaan aiempia valintoja. Skriptistä on vaikea edes puhua, jos järjestelmää halutaan testata monella samanaikaisella käyttäjällä, joista kukin voi tehdä edellä mainittuja asioita omassa tahdissaan.

Monen samanaikaisen käyttäjän järjestelmän automaattinen testaaminen ei ole vaikeaa vain siksi, että testien skriptaaminen on työlästä. Se on vaikeaa jo siksi, että ihmisen mielikuvituksen rajat tulevat nopeasti vastaan. Testaamisen arvoisia vaihtoehtoja on musertavan paljon.

Enemmän kuin testiaskelten arpomista

Testien automaattisen generoinnin ero skriptattuun testiautomaatioon on selvä: testiskriptin laatijan täytyy keksiä testitapaukset ja ohjelmoida ne skriptin muotoon. Tässä vaiheessa hän tulee määritelleeksi järjestyksen, missä testiaskeleet tullaan aina suorittamaan. Jos halutaan vähemmän ennalta-arvattavia testejä, kirjoitetaan tavallisesti sen verran logiikkaa, että tietyt testiaskeleet suoritetaan satunnaisessa järjestyksessä.

Automaattisessa testien generoinnissa testauksen suunnittelija määrittelee testiaskeleet ja tilanteet, joissa kukin askel voidaan suorittaa. Työkalun, tarkemmin sanottuna testigeneraattorin, tehtäväksi jää päättää testiaskelten lopulliset suoritusjärjestykset.

Testigeneraattorit pystyvät käymään läpi erilaisia käyttötapoja systemaattisesti. Siksi ne löytävät virheitä nopeammin ja toistettavammin kuin satunnaisesti arvotut testit. Hyvä testigeneraattori tuottaa samasta testiaskelten kirjastosta monenlaisia testejä: esimerkiksi optimoituja smoke-testejä, valittuja asioita toistuvasti läpikäyviä stressitestejä, sekä erittäin pitkiä ja monipuolisia luotettavuustestejä.

Testiaskelkirjaston monikäyttöisyydestä seuraa myös toinen merkittävä etu: testien ylläpidettävyys helpottuu. Kun testattava tuote kehittyy, valtavan testitapausjoukon ylläpidon sijaan riittää kirjaston testiaskelten ylläpitäminen. Jos jokin ominaisuus ei tuotteen kehitysvaiheessa toimi, estetään sitä testaavien testiaskelten päätyminen generoituihin testeihin. Kun tuote saa uusia ominaisuuksia, lisätään niitä testaavat testiaskeleet

kirjastoon. Uusien ominaisuuksien yhteentoimivuus vanhojen kanssa tulee testattua automaattisesti. Varsinaisia testejä ei tarvitse ylläpitää, koska uudet testit generoidaan aina uudelleen.

Milloin testiaskel voidaan suorittaa?

Testien automaattinen generoiminen vaatii, että testigeneraattori pystyy selvittämään, mitkä testiaskeleet se voi suorittaa missäkin tilanteessa. fMBT:ssä jokaisella kirjaston testiaskeleella on oma vahti, joka määrää milloin askel on suoritettavissa.

Joissakin tapauksissa vahdin vastaus perustuu puhtaasti jo aiemmin suoritettuihin testiaskeliin. Esimerkiksi "play"-testiaskeleen jälkeen tulee mahdolliseksi suorittaa "stop". fMBT:n testigeneraattori voi generoida testit tällaisissa tapauksissa jo ennen varsinaista testiajoa, ja se osaa optimoida testiajon aikanakin generoitua testiä etsimällä jatkuvasti lyhintä tapaa kasvattaa kattavuutta.

Joskus taas vahdin vastaus voi riippua ajoaikana tehdyistä havainnoista. Jos testattavan tuotteen ruudulla on "stop"-painike, seuraavaksi on mahdollista suorittaa "stop"-testiaskel. Tässä vaihtoehdossa testi täytyy generoida testiajon aikana, ja sen optimointi näiden testiaskelten osalta on mahdotonta.

Esimerkiksi musiikkisoitinta testaavat testiaskeleet "play" ja "pause" fMBT:n syötekielellä, AAL/Pythonilla näyttävät tältä:

```
variables {
    playing
}
initial_state {
    playing = False
}
input "play" {
    guard { return not playing }
    body { playing = True }
}
input "pause" {
    guard { return playing }
    body { playing = False }
}
```

AAL:ssä guard-lohko toimii vahtina ja body-lohko määrittelee, kuinka tilanne muuttuu, kun testiaskel on suoritettu onnistuneesti. Näiden lohkojen sisään kirjoitettu ohjelmakoodi on tavallista Pythonia.

Tässä esimerkissä vahtien arvot riippuvat täysin aiemmin suoritetuista testiaskelista. Näin ollen fMBT voi etsiä optimaalista testiä simuloimalla testiaskelia testaamatta niitä oikeasti. Mutta koska guard-lohkoon voi kirjoittaa mitä tahansa Python-koodia, niihin voi kirjoittaa myös ajonaikaisia tarkastuksia, kuten onko tuotteen ruudulla painikkeita, paljonko kello on tai paljonko levytilaa on käytettävissä.

Yksi plus yksi on paljon enemmän kuin kaksi

Edellisestä yksinkertaisesta esimerkistä testigeneraattori voi tuottaa vain testejä tyyliin "play", "pause", "play", "pause", jne. Toisaalta, jos saman musiikkisoittimen kuulokkeiden liittämistä ja irroittamista testattaisiin, uudet testiaskeleet voitaisiin kuvata näin:

```
variables {
    playing, headphones
}
initial_state {
    headphones = False
}
input "connect" {
    guard { return not headphones }
    body { headphones = True }
}
input "disconnect" {
    guard { return headphones }
    body {
        headphones = False
        playing = False
    }
}
```

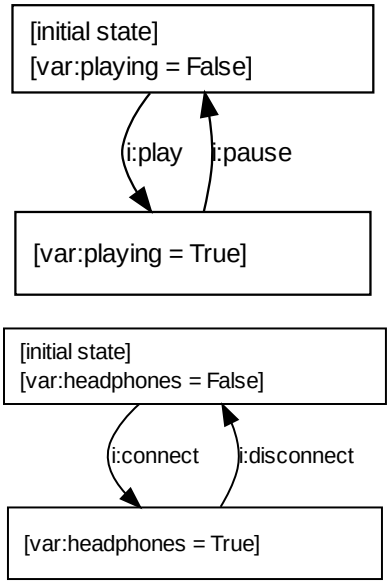
Jälkimmäisestä esimerkistä testigeneraattori tuottaa myös hyvin yksinkertaisia testejä: "connect", "disconnect", "connect", "disconnect", jne. Vaikka "playing"-muuttuja ei vaikuta juuri näiden testiaskelten testattavuuteen, jatkon kannalta on tärkeää, että sen arvoa muutetaan "disconnect"-askelen "body"-lohkossa. Se tarkoittaa, että musiikkisoittimen pitäisi lopettaa soittaminen, kun kuulokkeet irroitetaan.

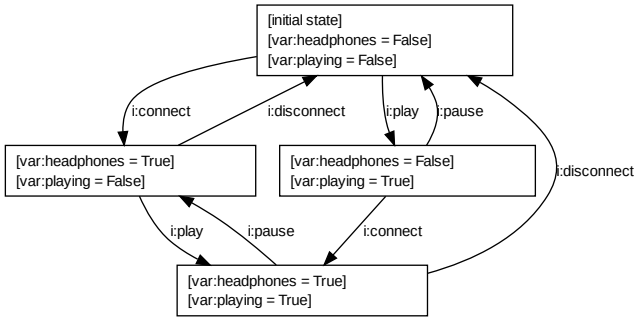
fMBT mahdollistaa toisistaan erillään määriteltyjen testiaskelten yhdistämisen. Tiedostoja voi sisällyttää toisiinsa ja niitä voi liimata peräkkäin. Jos testigeneraattorille annetaan tiedosto, joka saadaan yhdistämällä edelliset esimerkit, se alkaa tuottaa erittäin mielenkiintoisia testejä.

Esimerkkien yhdistelmästä generoidut testit testaisivat kuulokkeiden liittämisen ja irroittamisen silloin, kun musiikki soi, kun musiikki ei soi, sekä musiikin soittamisen kuulokkeiden kanssa ja ilman. Lisäksi ne testaisivat näiden erilaiset lomitukset, esimerkiksi kuulokkeiden liittämisen musiikin soidessa ja irroittamisen soiton päätyttyä. Yhdistettyinä nämä testiaskeleet testaavat paljon

sellaisia käyttötapoja, joita ne eivät olisi erillään testanneet.

Ilmiön voi ymmärtää helpommin, kun fMBT esittää yllä olevat AAL-koodit tila-avaruuksina (katso kuvat). Erikseen piirrettyinä esimerkit tuottavat vain kaksi tilaa (neliöitä; testin generointi alkaa tilasta, jossa lukee ”[initial state]”). Kummassakin tilassa voidaan testata vain yksi testiaskel (tilasta lähtevä nuoli). Kun testigeneraattori saa yhdistetyn AAL:n, se näkee neljä tilaa, joista kussakin se voi valita seuraavaksi testattavan askeleen kahdesta vaihtoehdosta.





Miten generoitua testiä pääsee ajamaan?

fMBT generoi ja suorittaa testiä samanaikaisesti. Jotta tämä olisi mahdollista, täytyy testiaskelille kirjoittaa myös toteutukset. Ne ovat koodia, joka on yhteydessä testattavaan järjestelmään: antaa sille tarvittavat syötteet ja tarkistaa, että järjestelmä reagoi syötteisiin odotetusti. Tämän koodin voi kirjoittaa Pythonilla testiaskelen sisään omaan "adapter"-lohkoonsa, "guard" ja "body" -lohkojen rinnalle. Verkosta löytyy valmiita esimerkkejä, missä näin on tehty mm. Linuxin MPlayerin ja Androidin Phone-sovelluksen testauksessa.

Testiaskelten ja niiden toteutusten lisäksi fMBT:n testigeneraattori tarvitsee konfiguraatiotiedoston. Siinä kerrotaan mistä testiaskeleet ladataan, millä tavalla testejä generoidaan, miten kattavuutta mitataan ja missä tilanteessa testin generointi ja suoritus lopetetaan. Jos testiaskeleita määritellään fmbt-editor:lla, se tuottaa automaattisesti konfiguraatiotiedoston, jota voi muokata käyttötarpeiden mukaan.

Milloin testejä kannattaa generoida?

Jos tarvitset suoraviivaisen automaattisen testin, joka suorittaa testiaskeleet aina samassa järjestyksessä, perinteinen testiskripti on edelleen nopein tapa päästä maaliin. Mutta jos haluat testien kattavan monia eri tapoja käyttää testattavaa järjestelmää, testien automaattinen generointi helpottaa työtä merkittävästi. Kattavassa rinnakkaisten käyttötapausten testauksessa testien generoinnille ei ole järkeviä vaihtoehtoja.

Viitteet

- fMBT-työkalun kotisivu: http://01.org/fmbt.
- Tutoriaali fMBT:n käyttöön ja testien generointiin: https://github.com/01org/fMBT/wiki/Tutorial
- Blogi "Testing and understanding concurrency" sisältää esimerkin rinnakkaisten käyttötapausten testaamisesta.

Antti Kervinen on tutkinut ja rakentanut automaattisia testigeneraattoreita yli kymmenen vuoden ajan. Parhaillaan hän kehittää fMBT-työkalua Intelillä.

TestausOSY

Finnish Association of Software Testing

Kaikenlaisten softatestausihmisten yhteisö.
Yli 900 jäsentä ja kasvussa. Laadun ja sen tekijöiden asialla moniarvoisesti ja -äänisesti. Ajan tasalla ja edellä.

Jäsenetuina hyvän yhteisön ajatustenvaihtoa ja tekemistä, ilmaispääsy moniin tilaisuuksiin ja alennuksia kaupallisiin koulutuksiin.

Kaikki tämä ilman jäsenmaksua.

Liity jäseneksi osoitteessa
www.testausosy.fi

Tule LinkedIn-ryhmäämme keskustelemaan

LinkedIn

Matti Vuori 2014

TestausOSY
Finnish Association of Software Testing