hyväksymispäivä arvosana

arvostelija

Testauksen historiaa

Sirpa Paakki

Helsinki 9.3.2004

Tietojenkäsittelytieteen historia -seminaari

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

Perinteisen elinkaarimallin mukaisessa ohjelmistokehitysprojektissa voidaan jokaiseen prosessin vaiheeseen liittää *määrittely* (specification) ja sitä vastaava *toteutus* (implementation). Korkeimmalla tasolla määrittely vastaa koko tietojärjestelmän vaatimusmäärittelyä ja toteutus vastaa koko tietojärjestelmän lopullista toteutusta. Alemmalla tasolla määrittely voi olla esimerkiksi yksittäinen modulikuvaus ja sitä vastaava toteutus kyseisen modulin lähdekoodi [Bei90 s. 2-11].

Testauksen (testing) tavoitteena on varmistaa, että toteutus vastaa järjestelmän tai ohjelman määrittelyä. Täydellistä vastaavuutta eli ekvivalenssia ei toteutuksen ja määrittelyn välillä kuitenkaan voida osoittaa, koska tällaisen menetelmän avulla voitaisiin ratkaista myös Turingin pysähtymisongelma, joka on muun muassa lähteessä [How87 luku 4] todistettu ratkeamattomaksi ongelmaksi.

Toinen ongelma liittyy itse määrittelyn oikeellisuuteen. On epärealistista olettaa, että järjestelmän vaatimusmäärittely tyhjentävästi, yksiselitteisesti ja ristiriidattomasti selittäisi järjestelmän toiminnan, koska tällaiset määrittelyt on pääsääntöisesti annettu jollakin epäformaalilla kuvausmenetelmällä (tavallisesti luonnollisella kielellä) ja verraten yleisellä tasolla.

Jos määrittely kuitenkin on annettu formaalisti voidaan ohjelmalle rakentaa todistus, joka voidaan muodostaa joko käsin tai automaattisesti. Molemmilla tavoilla jää kuitenkin jäljelle epävarmuus siitä, onko itse todistus muodostettu oikein ja onko käytetty aksioomien järjestelmä validi ja automaattisen todistuksen yhteydessä lisäksi se, toimiiko kyseinen automaattinen todistusjärjestelmä oikein [GeH88]. Vaatimusmäärittelyn lisäksi olisi myös ohjelman ympäristö fyysiselle laitteistotasolle asti kuvattava ja todistettava.

Edellä mainittujen ongelmien takia ohjelmistojen testauksen voidaan yleisesti ajatella tarkoittavan vain ohjelmiston oikeellisuuden tutkimista. Testauksen tulos kertoo tällöin miten hyvin tai huonosti ohjelma vastaa määrittelyä. Jos ohjel-

mistoa ei voida todistaa oikeaksi millään formaalilla menetelmällä, on ohjelman oikeellisuutta tutkittava ohjelmistoa suorittamalla eli etsimällä ohjelmistosta virheitä [Bei90 s. 2-11]. Tässä esityksessä keskitytään testauksen historiaan nimenomaan jälkimmäisestä eli käytännönläheisemmästä näkökulmasta.

Testauksen tavoitteet sekä se, mitä ohjelmistokehitysprosessin toimintoja termin alle katsotaan kuuluvaksi, ovat vaihdelleet aikojen kuluessa. Gelperin ja Hetzel kuvaavat artikkelissaan ”The growth of software testing” testauksen neljä historiallista kehitysvaiheitta, joista muodostettu yhteenveto nähdään taulukossa 1 [GeH88]. Viimeinen eli ehkäisysuuntautunut aikakausi on useissa artikkeleissa katsottu alkaneen edellä mainitusta Gelperinin ja Hetzelin artikkelista. Eri aikakausien päättyminen tai alkaminen ei ole todellisuudessa täysin selvärajainen, mutta esityksen jäsentämisen helpottamiseksi käytetään tässä suoraan Gelperinin ja Hetzelin jaottelua.

- 1956	Jäljityksen aikakausi (Debugging-Oriented period)
1957 - 1978	Demonstraation aikakausi (Demonstration-Oriented period)
1979 - 1982	Hajoittamisen aikakausi (Destruction-Oriented period)
1983 - 1987	Arviointisuuntautunut aikakausi (Evaluation-Oriented period)
1988 -	Ehkäisysuuntautunut aikakausi (Prevention-Oriented period)

Taulukko 1: Ohjelmistojen testauksen aikakaudet.

Tässä esityksessä kuvataan luvuissa 2–6 taulukon 1 mukaiset testauksen aikakaudet sekä testauksen kehittymiseen vaikuttaneet artikkelit tai muuten testauksen tutkimusalueeseen vaikuttaneet henkilöt. Luvussa 6 käsitellään lisäksi lyhyesti testausprosessien ja testausmenetelmien kehittymistä viimeisen eli ehkäisysuuntautuneen aikakauden aikana.

2 Jäljityksen aikakausi

Ensimmäinen von Neumann -arkkitehtuurin mukainen eli tietokoneen muistissa olevaa ohjelmaa suorittava tietokone oli EDSAC (Electronic Delay Storage Automatic Calculator), joka valmistui vuonna 1949 [Cam92]. Ensimmäiset EDSAC-koneella näytösluontoisesti suoritetut ohjelmat olivat täysin virheettömiä. Näitä melko yksinkertaisia ohjelmia oli tosin suunniteltu ja paranneltu jo kauan ennen EDSAC-koneen valmistumista, joten niitä voidaan pitää tässä mielessä erityistapauksina.

Ensimmäisiä käytännön tietokoneohjelmia oli koodaamassa muun muassa Maurice Wilkes, joka vuonna 1949 teki sen merkittävän havainnon, että suurimman osan ajastaan ohjelmien koodaajat käyttivät virheiden etsimiseen ohjelmistaan [Cam92]. Tästä tapahtumasta katsotaan *virheenjäljityksen* (debugging) historian alkaneen. Tosin jo vuonna 1945 Grace Hopper oli löytänyt Mark I -tietokoneesta historian ensimmäisen *virheen* (bug), mutta tämä virhe eli ”bugi” oli todellakin vain hyönteinen, joka aiheutti koneen pysähtymisen. Tästä tapauksesta termi bugi on kuitenkin saanut alkunsa.

Digitaalisen tietojenkäsittelyn alussa *manuaalista tarkistamista* (manual checking), virheenjäljitystä ja testausta ei juurikaan eroteltu toisistaan. Aikakauden artikkeleissa oli tyypillistä pitää termejä jäljitys ja testaus joko toistensa synonyymeinä tai vaihtoehtoisesti luokitella sanan jäljitys alle kaikki mahdollinen virheiden etsimiseen ja poistamiseen liittyvä toiminta. Lisäksi testitapausten muodostaminen tapahtui joko täysin sattuman varaisesti (ad hoc) tai ohjelmoijan näppituntumaa käyttäen [GeH88]. Mitään apuvälineitä jäljityksen helpottamiseksi ei ollut. EDSAC:in ajalta säilyneissä raporteissa mainitaan tosin termi *kurkistelu* (peeping), joka tarkoittaa rekistereiden tilan tarkkailemista jokaisen suoritetun käskyn jälkeen, mutta tämän menetelmän käyttö oli erittäin epäsuotavaa, koska koneen käyttöaika oli rajoitettu [Cam92].

Ensimmäisiä tietokoneohjelmien testausta käsitteleviä artikkeleita on Alan Turingin kirjoittama artikkeli ”Computing Machinery and Intelligence” vuodelta 1950 [Tur50b]. Tätä ennen Turing oli tosin jo käsitellyt ohjelmien oikeaksi todistamista toisessa artikkelissaan samana vuonna [Tur50a]. Turingin näissä artikkeleissa esittämiin kysymyksiin on etsitty vastauksia lähinnä keinoälyyn liittyvän tutkimuksen piirissä, mutta myös ohjelmistojen testauksen perustava kysymys ”Kuinka tiedämmme, että ohjelma täyttää vaatimuksensa?” voidaan näistä artikkeleista löytää.

3 Demonstraation aikakausi

Virheenjäljityksen aikakaudella tietokoneohjelmat kirjoitettiin suoraan konekoodilla ja myöhemmin assemblerkielillä. Kuitenkin vasta korkean tason ohjelmointikielien ja kääntäjien valmistumisen myötä tuli mahdolliseksi kehittää huomattavasti suurempia ohjelmia entistä lyhyemmässä ajassa. Ohjelmien määrän lisääntymisen sekä niiden kompleksisuuden ja tuotantokustannusten kasvun myötä ohjelmistotuotantoon liittyvä taloudellinen riski kasvoi, mikä puolestaan lisäsi testauksen merkitystä. Oli selvää, että ohjelmat sisälsivät paljon virheitä ja puutteita, ja näiden virheiden korjaaminen tai häiriötilanteista toipuminen puolestaan aiheutti merkittäviä lisäkustannuksia. Demonstraation aikakaudella alkoivat sekä ohjelmistojen käyttäjät että projektien vetäjät kiinnittää entistä enemmän huomiota testauksen laatuun. Useiden epäonnistuneiden ohjelmistohankkeiden takia 60-luvun lopulla puhuttiin myös usein *ohjelmistokriisistä* (software crisis). [GeH88]

Alan kirjallisuudessa käänteen tekevänä artikkelina voidaan pitää Charle Bakerin vuonna 1957 kirjoittamaa artikkelia, jossa ensimmäisen kerran tehtiin selvä ero jäljityksen ja testauksen välille [Bak57]. Näille termeille annetut määritelmät eivät kuitenkaan vastanneet vielä lopullisia eli nykypäivänä käytettäviä määritel-

miä. Jäljityksen tavoitteena oli ainoastaan varmistaa, että ohjelma ylipäänsä toimii (eli että ohjelma ei kaadu). Testauksen tavoitteena puolestaan oli varmistaa, että ohjelma ratkaisee ongelman eli täyttää vaatimuksensa. Molempien termien alle sisältyi siis edelleenkin virheiden paljastamiseen, paikallistamiseen ja korjaamiseen liittyviä toimintoja.

Testauksen teoria kehittyi paljon 70-luvulla, jolloin ilmestyi useita artikkeleita, jotka liittyivät testiaineiston valitsemiseen ohjelman ohjaus- tai tietovuon perusteella [GoG75, How75]. Tavoitteena oli siis kehittää testaukseen menetelmiä, joiden avulla pystyttäisiin muodostamaan sellainen testitapausten joukko, jonka avulla ohjelma voitaisiin testata yhtä kattavasti kuin kaikkien mahdollisten testitapausten avulla. Näitä ajatuksia ei tosin sovellettu kaupallisissa ohjelmistohankkeissa.

4 Hajoittamisen aikakausi

Demonstraation aikakaudella suhtauduttiin melko optimistisesti ohjelman oikeellisuuden osoittamiseen lähinnä ad-hoc -menetelmän avulla muodostettujen testitapausten perusteella. Tähän asennoitumiseen toi kuitenkin muutoksen Meyersin klassikkoteos ”The Art of Software Testing” vuodelta 1979. Tässä teoksessa Meyers antoi testaukselle sen edelleenkin käytössä olevan määritelmän: testaus on ohjelman suorittamista siten, että tarkoitus on löytää ohjelmasta virheitä [Mey79 luku 1].

Meyers siis painotti nimenomaan aktiivista virheiden etsintää eikä ohjelman virheettömyyden osoittamista. Jos testaajan tai ohjelmoijan tavoitteena olisi virheettömyyden osoittaminen, saattaisi tämä tavoitteen asettelu aiheuttaa alintajuntaisen vääristymisen testitapausten valinnassa. Toisaalta silloin, kun päätavoitteena on virheiden löytäminen, on todennäköisempää, että testitapaukset ja testiaineisto ovat tämän tavoitteen mukaiset. Lisäksi testitapauksen odotettu tulos on kir-

jattava ylös etukäteen, koska muuten saatu tulos on aina myös odotettu tulos. [Mey79 s. 5-10]

Hajoittamisen aikakaudella myös jäljitys sai lopullisen merkityksensä, jonka mukaan jäljitys on vain ja ainoastaan virheen paikallistamiseen ja korjaamiseeen liittyvää toimintaa. Lisäksi testaus erillisenä vaiheena ohjelmiston *elinkaaressa* (life cycle) sai huomattavasti aikaisempaa enemmän painoarvoa. Muun muassa Howdenin artikkeli [How80] sisältää tällaisen virheiden etsintään painottuvan lähestymistavan kuvauksen.

Testauksen teorian tutkimuksessa oli muodstunut tässä vaiheessa jo kaksi erilaista lähtökohtaa. *Toiminnallinen* (functional, black box) testaus vakiintui tarkoittamaan ainoastaan ohjelman toiminnallisuuteen perustuvaa testausta, jossa testitapausten muodostaminen tapahtuu ohjelman määrittelyssä annetun toiminnallisuuden perusteella. *Rakenteellinen* (structural, white box) testaus puolestaan perustuu tietoon ohjelman kooditason rakenteesta ja riippuvuussuhteista. Rakenteellisessa testauksessa voidaan esimerkiksi muodostaa sellainen joukko testitapauksia, että modulin jokainen lause tulee ainakin kerran suoritettua tai että modulin jokainen ehtolauseke tulee ainakin kerran evaluoitua [FeP96 s. 300]. Tällaisia rakenteelliseen kattavuuteen perustuvia kriteereitä alettiin käyttää mm. yksikkötestauksessa.

5 Arviointisuuntautunut aikakausi

Arvioitiin painottuvalla aikakaudella testaus nähtiin ensimmäisen kerran osana metodologiaa, johon liitetään mukaan ohjelman analysointi, *katselmointi* (review) sekä muut testaustoimenpiteet ja jonka pyrkimyksenä on tuotteen arviointi sen elinkaaren jokaisen vaiheen lopussa [GeH88]. Tämä aikakausi katsotaan alkaneeksi Yhdysvaltojen kansallisen standardoimisliiton ohjeistuksesta vuodelta 1983, jonka keskeinen filosofia on seuraava:

> Mikään yksittäinen verifiointi, validointi tai testaustekniikka ei voi taata oikein toimivaa, virheetöntä ohjelmaa. Kuitenkin tiettyä projektia varten huolella valittu joukko näitä tekniikoita voi auttaa varmistamaan, että projekti tuottaa ja ylläpitää laadukkaita ohjelmia [NBS83].

Elinkaaren jokaisen vaiheen loppuun siis määriteltiin joukko toimintoja, joiden avulla tuli arvioida kuinka hyvin tuote vastaa määrittelyä. Lisäksi aikakaudella realisoitui ensimmäisen kerran ajatus siitä, että mitä myöhemmässä vaiheessa virhe löytyy, sitä enemmän se aiheuttaa lisäkustannuksia.

Yksi systemaattisen ja kurinalaisen testausprosessin tärkeimpiä puolestapuhujia 80-luvulta alkaen on ollut Boris Beizer, jonka kirjoittamaa teosta ”Software Testing Techniques” on käytetty myös oppikirjana yliopistoissa (huom: ensimmäinen painos ilmestyi vuonna 1983). Boris Beizer painotti lisäksi löytyneiden ja korjattujen virheiden luokittelun ja tilastoinnin hyötyjä. Organisaation ylläpitämien historiatietojen perusteella oli mahdollista päätellä esimerkiksi riskialttiit modulit tai ennustaa jossain määrin virhetiheyttä tai virheiden löytymisvauhtia [Bei90 s. 213-242].

6 Ehkäisysuuntautunut aikakausi

Viimeinen selkeästi erottuva aikakausi testauksen historiassa katsotaan alkaneeksi Gelperinin ja Hetzelin artikkelista vuodelta 1988 [GeH88]. Sen lisäksi että tuote arvioidaan jokaisen elinkaaren vaiheen lopussa, on jokaisen vaiheen edetessä käytettävä testausta tai muita arviontimenetelmiä estämään uusien virheiden syntymistä. Tämän uuden ajattelumallin mukaan arviointimenetelmien avulla pyritään paikallistamaan sellaiset ohjelman tai suunnittelu- ja määrittelydokumenttien osat, joissa virheitä saattaisi syntyä. Esimerkkinä tällaisista arviointimenetelmistä voidaan mainita *tarkastukset* (inspection) ja katselmoinnit. Gelperin ja

Hetzel määrittelivät lisäksi ensimmäisen kerran kattavan metodologian testauksen ja testausprosessin hallintaan.

Uuden ajattelumallin mukaan testaus tuli integroida mukaan ohjelmistotuotantoprosessiin siten, että prosessin jokaisessa vaiheessa testiaineiston suunnittelu tuli ottaa huomioon mahdollisimman aikaisin. Testiaineiston suunnittelu jo vaatimusmäärittelyn ja arkkitehtuurimäärittelyjen perusteella todennäköisesti nostaa esille sellaisia kysymyksiä, jotka voivat paljastaa lähdeaineistosta riittämättömyyttä, ristiriitaisuutta, tulkinnanvaraisuutta tai suoranaisia virheitä. Tällöin näiden puutteiden korjaaminen on vielä yksinkertaista ja kustannuksiltaan kohtuullista verrattuna myöhempiin kehitysvaiheisiin. Testauksen huomioiminen siis parantaa määrittelyiden ja ohjelmakoodin laatua jo ennen varsinaisten testitapausten suorittamista. Lisäksi testausprosessi tunnistettin yhdeksi riskienhallinnan välineeksi, jossa testausstrategia ja testitapausten suunnittelu perustetaan ohjelman suorituksessa ilmenevien häiriöiden aiheuttamien seurausten ja näiden häiriöiden todennäköisyyden arvioinnille [GeH88].

Viimeisimpiä suuntauksia sekä testauksen että koko ohjelmistoprosessin tehostamisessa on itse prosessin *kypsyystason* (maturity level) määrittely ja jatkuva kehittäminen [BSC96]. Testausprosessin kypsyystasoa määriteltäessä arvioidaan mm. testaukseen liittyvän ohjeistuksen tasoa, testaussyklin vaihejakoa, testauksen suunnittelumenetelmien tasoa, testauksessa käytettyä organisaatiomallia, testauksen etenemisen seurannassa käytettyjä menetelmiä, testauksen apuvälineitä ja itse testausmenetelmiä ja tekniikoita.

Oleellista on lisäksi, että korkealla kypsyystasolla olevalla testausprosessilla on selkeät ja dokumentoidut menetelmät joiden avulla mitataan ja seurataan sekä testauksen etenemistä että koko prosessin laatua [BSC96]. Testauksen mittaaminen kohdistuu siis sekä tuotteeseen että prosessiin ja tavoitteena on saavuttaa prosessi, joka jatkuvasti kehittää itseään.

Testausta ei voida kuitenkaan loputtomasti tehostaa vain prosessia parantamalla,

vaan oleellista on myös testauksessa käytetyt menetelmät, joiden avulla voitaisiin mahdollisimman tehokkaasti löytää ohjelmistossa olevat virheet. Tähän ongelmaan ei kuitenkaan voida löytää hopealuotia, vaan testattava ohjelmisto on aina testattava mahdollisimman kattavasti. Rakenteellisessa testauksessa voidaan tosin kehittää apuvälineitä, joiden avulla voidaan mitata testitapauksilla saavutettua kattavuutta ja jopa automaattisesti luoda uusia testitapauksia, mutta toiminnallisessa testauksessa testitapaukset on luotava aina ohjelmiston määrittelyn perusteella [CrJ02 s. 159].

7 Yhteenveto

Käsitykset siitä, mitä ohjelmistojen testaus tarkoittaa, mikä on testauksen tavoite ja mitä toimintoja testaukseen katsotaan kuuluvaksi, ovat vaihdelleet suuresti eri aikakausina. Sen jälkeen kun testauksen määritelmä lopuulta vakiintui, oli pääasiallisena tavoitteena edelleen kuitenkin vain virheiden etsintä toteutetusta ohjelmistosta. Aivan viimeaikoihin asti on tämän jälkeen jatkunut vaihe, jossa testaus nähdään systemaattisena prosessina, jonka tavoitteena on varmistaa ohjelmiston laatu erilaisten toimenpiteiden avulla ohjelmiston elinkaaren jokaisessa vaiheessa rinnakkain toteutuksen kanssa.

Lähteet

Bak57 Baker C., Review of D.D. McCracken's "Digital Computer Programming". *Mathematical Tables and Other Aids to Computation*, Volume 11, Number 60, October 1957, 298–305.

Bei90 Beizer B., *Software Testing Techniques*. Second Edition, Van Nostrand Reinhold, 1990.

BSC96 Burnstein I., Suwannasart T., Carlson C. R., Developing a Testing Maturity Model for Software Test Process Evaluation and Improvement. *Proceedings of the IEEE International Test Conference*, Washington D.C., USA, October 1996, 581–589.

Cam92 Campbell-Kelly M., The Airy tape: An Early Chapter in the History of Debugging. *IEEE Annals of the History of Computing*, Volume 14, Number 4, 1992, 16–26.

CrJ02 Craig R. D., Jaskiel S. P., *Systematic Software Testing*. Artech House Publishers, May 2002.

FeP96 Fenton N. E., Pfleeger S. L., *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, 1996.

GeH88 Gelperin D., Hetzel B., The Growth of Software Testing. *Communications of the ACM*, Volume 31, Number 6, June 1988, 687–695.

GoG75 Goodenough J. B., Gerhart S. L., Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, Volume 1, Number 2, June 1975.

How75 Howden W., Methodology for the Generation of Program Test Data. *IEEE Transactions on Computers*, May 1975, 554–560.

How80 Howden W., Functional Program Testing. *IEEE Transactions on Software Engineering*, Volume 6, Number 2, March 1980, 162–169.

How87 Howden W., *Functional Program Testing and Analysis*. McGraw-Hill, New York, 1987.

Mey79 Meyers G. J., *The Art of Software Testing*. John Wiley & Sons, New York, 1979.

NBS83 Guideline for Lifecycle Validation, Verification and Testing of Computer Software. National Bureau of Standards Report NBS FIPS 101, Washington D.C., USA, 1983.

Tur50a Turing A., Checking a Large Routine. *Report of a Conference on High Speed Automatic Calculating-Machines*, January 1950, 67–69.

Tur50b Turing A., Computing Machinery and Intelligence, *Mind: A Quarterly Review of Psychology and Philosophy*, October 1950, 433–460.