Ohjelmiston testaus

Antti Enqvist
Antti Krats
Kai Lahti
Mikko Luukkonen
Asmo Maanselkä
Juho Yli-Honkola

Ohjelmistotuotanto-kurssin harjoitustyö

19.11.2001

JYVÄSKYLÄN YLIOPISTO

1. Johdanto

Ohjelmistotestaus kuuluu olennaisena osana ohjelmistoprosessiin. Testauksella pyritään toteamaan ohjelmiston toimivuus, eli että se täyttää sille annetut vaatimukset, sekä systemaattisesti löytämään tuotteessa olevat virheet. Kaikkia virheitä on toki mahdoton löytää (ainakin suuremmissa ohjelmistoissa), mutta hyvällä testauksen suunnittelulla saadaan karsittua tuotteesta pois ainakin usein toistuvat virheet. Näin saadaan myös yrityksen luotettavuutta parannettua, ainakin asiakkaiden silmissä. Toisaalta ohjelmistotestausta tukee myös se tosiasia, että noin 40% kaikista ohjelmistoprojekteista epäonnistuu täydellisesti.

2. Ohjelmistotestauksen lähtökohdat

2.1. Ohjelman laatu

Ohjelman laadulle asetetut vaatimuksen vaihtelevat usein laajastikin ohjelman käyttötarkoituksen mukaan. Onhan selvää että ohjelmistoilta joilta edellytetään turvallisuutta ja toimivuutta, ovat myös laatuvaatimukset ja testauksen tarve huomattavasti korkeammat, kuin ohjelmistoilla joiden ”vääränlainen toimita” ei vaaranna ihmisten henkiä. Äärimmäistä turvallisuutta vaativaa ohjelmistoa voisi edustaa esim. ydinvoimalan valvontaohjelmiston, jonka vääränlaisesta toiminnasta seuraisi laaja katastrofi.

Seuraavassa on listattu kohtia, joita hyvänlaatuisen ohjelmiston tulee täyttää:

Ohjelman tulisi vastata tarkoitustaan ja määrittelyään
Olla helppokäyttöinen ja riittävän yksinkertainen
Olla helposti ylläpidettävä ja päivitettävä
Olla turvallinen
Olla mahdollisimman suorituskykyinen

2.2. Testauksen laatu

Testauksen laatua voidaan mitataan sillä, miten kattavasti testaus on suoritettu. Täydellisellä kattavuudella takoitetaan sitä, että ohjelman kaikki mahdolliset reitit olisi tutkittava kaikilla mahdollisilla syötteillä. Tästä voi hyvin päätellä, että täydellisesti kattava testaus on mahdoton toteuttaa. Testauksen määrä on sidottu käytettävissä oleviin resursseihin(aika, raha, henkilöstöresurssit ym.), sekä siihen millaiseen tarkoitukseen ohjelmisto tulee. Myös voisi ajatella, onko järkevää toteuttaa täysin kattava testaus, vaikka siihen olisi käytettävissä tarvittavat resurssit. Jos taas resursseja eli syystä tai toisesta ole käytettävissä, niin usein testaus suoritetaan ohjelmoijan toimesta, jolloin omien virheiden löytäminen on vaikeaa. Tai ainakin niitä onnistuu hyvin välttämään, sillä itse kyllä tietää millaisia syötteitä ohjelmaan tulisi saada, jotta se toimisi oikein.

3. Testausprosessi

3.1. Testauksen suunnittelusta

Ennen varsinaisen testaamisen aloittamista on erittäin tärkeää suunnitella, mitä ja miten ohjelmaan aiotaan testata - pelkkä sattumanvarainen kokeileminen ei ole testausta. Systemaattinen ja hyvin suunniteltu testaus antaa luotettavimman tuloksen ohjelman todellisesta laadusta, lisäksi se säästää aikaa ja antaa selkeän kuvan ongelma-alueista. Testausmenetelmien arviointi ja kattavuuden selvittäminen on todella vaikea tehtävä. On tärkeää huomioida, että menetelmien käytettävyydessä ja soveltuvuudessa on suurta vaihtuvuutta. Se johtuu usein sovellusteknisistä yksityiskohdista.

Dokumentointi ja raportointi on tehtävä suurelta osin prosessinaikana, se helpottaa ihmisten välistä kommunikaatiota ja nopeuttaa kokonaistestaukseen tarvittavaa aikaa. Testausprosessista laaditaan omat testisuunnitelmansa. Testisuunnitelma pitää sisällään tarkkoja yksittäisiä suunitelmia, kuten Testausprosessin suunnitelma, jossa määritellään testauksen vaiheet. Vaiheita voi olla erinlaisia ohjelmistosta riippuen. Tässä eräänlainen jako[1]:

> Vaatimusten seurannan suunnitelma, jossa määrätään ne järjestelmälle asetetut vaatimukset, joiden toteutumista testataan. Testit kohdistetaan ennalta suunniteltuihin vaatimuksiin. Siten saadaan systemaattista tietoa ongelma-alueiden luotettavuudesta ja ongelmien kriittisyydestä.
> Testikohteiden määrityksen suunnitelma, jossa määrätään ne ohjelmistoprosessin tuottamat tulokset, joita otetaan tarkempaan testaukseen. Merkitään ja asetetaan ongelma-alueet ja annetaan niille suurempi prioriteetti, lisäksi niiden käyttäytymistä seurataan kiinteämmin.
> Testausten ajoituksen suunnitelma, jossa testien vaiheistus määrätään suhteessa muuhun ohjelmistoprosessiin. Yleensä testaus etenee sitä mukaan kun valmiita palikoita syntyy. Testausta ei saa jättää viimeiseksi toimenpiteeksi ,vaan se on kiinteästi yhteydessä ohjelmistoprojektin etenemiseen.
> Testausten dokumentointisuunnitelma ,jossa on olennaista määritellä tarkasti millaisia tietoja tarvitaan testauksesta kerätään ja tallennetaan järjestelmän validointia ja luotettavuuden kehittämistä varten. Dokumentointisuunnitelma vaikuttaa suurelta osin koko testauksen onnistumiseen. Dokumentoinnin ja raportoinnin olennainen osa ovat ns. testilokit (test log). Ne sisältävät raportin suoritettujen testien yksityiskohdista. Monet testausohjelmistot tekevät itse lokit, lokitiedostoihin.
> Laitteisto- ja ohjelmistovaatimusten testaussuunnitelma, jossa määritellään laitteistovaatimusten toteutumista selvittävät testit. On tärkeää testata ohjelmistoa erilaisissa verkkotyypeissä, työympäristöissä ja käyttöjärjestelmissä.
> Uhat, jotka saattavat vaikuttaa testauksen läpiviemiseen, esimerkiksi aikataulun tai henkilöstön johdosta. Yleensä testaukseen käytetään liian vähän aikaa ja projekti on tiukkaan aikataulutettu, joten testaukseen on vaadittava tarvittavaa aikaa jo projektin aloitusvaiheessa. Uhkia kannatta taulukoida ja asettaa tietyille uhille prioriteetti korkeammalle.

3.2. Testauksen dokumentointi

Testausdokumentaatiota voi syntyä varsin paljon: järjestelmäsuunnitelma, integrointitestaussuunnitelma jokaisesta integrointitestistä, moduulitestaussuunnitelma jokaisesta moduulitestistä, sekä vastaavasti raportti jokaisesta suoritetusta testistä. Pienehköissä projekteissa riittää yleensä yksi testaussuunnitelma, joka kattaa sekä integrointitestauksen että järjestelmätestauksen. Alustava suunnitelma laaditaan määrittelyvaiheessa ja sitä täydennetään myöhemmin suunnitteluvaiheessa. Testaussuunnitelmat voidaan myös sisällyttää projektisuunnitelmaan, toiminnalliseen määrittelyyn ja tekniseen määrittelyyn. [1]

Testaussuunnitelmasta selviää mm. mitä testejä tehdään ja milloin, miten ne järjestetään ja millaisia lopputuloksia odotetaan. Oleellisen tärkeää on määritellä testauksen lopettamiskriteerit. [1]

---

[1] Jako perustuu osittain http://erin.mit.jyu.fi/pako/kurssit/ot2000/l12/lect12/lect12.html

Testausdokumentin pitäisi vakuuttaa käyttäjä ohjelman toimivuudesta. Dokumentissa kuvataan testitapaukset ja olosuhteet, annetut syötteet ja odotetut tulokset. Koska ohjelma toimii, ei ole syytä erikseen jokaisen testikuvauksen jälkeen todeta, että "Testi onnistui" tms.

Dokumentissa on syytä kuvata seuraavat asiat:

Ensin lyhyt alustus, minkä osien testaus on dokumentoitu ja miksi, käytetyt testausmenetelmät.

> Tärkeimpien komponenttien testaus. Tämä tarkoittaa ohjelman toiminnallisuuden kannalta keskeisten osien (aliohjelmien) testausta. Tästä pitäisi ilmetä, että ohjelma toimii oikeellisesti ja toteuttaa tehtävänsä.
> Käyttöliittymän testaus. Tämä tarkoittaa käyttäjän syötteiden käsittelyä sekä ohjelman käytettävyyden testausta ja käyttöliittymän toimintaa. Tässä vaiheessa ei tarvitse puuttua käyttöliittymän toteutukseen, erilaisten testitapausten kuvaus riittää. Testausmateriaalin voi laittaa myös dokumentin liitteeksi, mikäli sitä on paljon. [2]
> Eri testaustasoilla löytyneet virheet tulisi raportoida ja analysoida. Kirjattavia tietoja ovat mm. virheen kuvaus, miten vakavasta virheestä on kysymys, milloin se löydettiin, miten se olisi voitu löytää aikaisemmin, milloin virhe oli tehty ja miten se olisi voitu estää. Ilman raportteja on mahdotonta saada selville testaukseen käytettyä aikaa ja laatujärjestelmän kehityskohteita. [1]

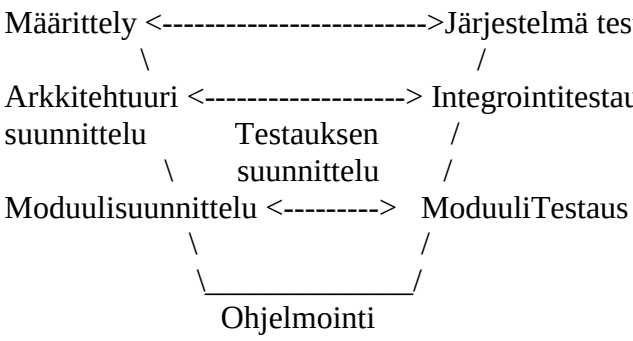
3.3. Virheenjäljitysprosessista

Virhe on poikkeama määritellystä ohjelman suorituksesta. Toisaalta se voi olla jokin iso looginen kokonaisuus esim. ohjelma laskee väärin tai pieni huolimattomuus esim. muuttuja alustamatta. Virheiden luonne ja syntymekanismi riippuu paljon ohjelmiston tekijöiden osaamistasosta, työkaluista ja tehtävän luonteesta.

Tyypillinen empiirisesti kerätty arvio virheiden määrästä koodausvaiheessa on n. 30 - 90 virhettä 1000 ohjelmakoodin riviä kohden. Lisäksi tyypilliseen hyvin testattuun kaupalliseen koodiin jää eri arvioiden mukaan n. 0.5-3 virhettä 1000 riviä kohden. Monet näistä virheistä ovat koodissa sen koko elinkaaren ajan. On huomattavaa, että runsas testausmenetelmien käyttö ei välttämättä vähennä lopputuotteeseen jäävien virheiden määrää. Sen sijaan hyvä suunnittelu vähentää niitä huomattavasti.[3]

Virheenjäljitys prosessissa yritetään etsiä, korjata ja paikantaa testitapahtumien paljastamia virheitä. Halutaan paikantaa ongelmia aiheuttavat virheet sekä korjata ne. Virheen paikantaminen ei ole helppoa. Virheen korjaus voi aiheuttaa lisää virheitä tai korjataan tyytyväisenä jokin yksittäinen virhe, mutta todellinen aiheuttaja jää korjaamatta. Myöskin yhteensopivuus toisten toimittavien komponenttien, ohjelmistojen ja protokollien kanssa voi aiheuttaa ongelmia. Debuggauksen tarkoituksena on etsiä virhettä aiheuttava ohjelmanosa ja eristää se. Debuggauksen lähestymistapoina ovat backtracking- tai cause elimination-metodit. Backtracking-metodin mukaan koodia lähdetään käymään virheen esiintymiskohdasta taaksepäin, kunnes virhe löydetään. Cause elimination-metodin mukaan virheen aiheuttajaa jäljitetään sulkemalla pois "varmat tapaukset" (joiden ei uskota aiheuttaneen virhettä).

4. Ohjelmistotuotannon testaustasot



Kuva1. Testauksen V-malli[1]

Erilaisia testaustasoja ovat ns. V-mallin mukaisesti moduulitestaus, yksikkötestaus ja järjestelmätestaus. Järjestelmätestausta voi joskus seurata erillinen kenttätestaus ja hyväksymistestaus.

4.1. Moduulitestaus

Moduulitestauksessa on testattavana yksittäinen moduuli. Moduuli koostuu yleensä noin 100-1000 ohjelmarivistä. Moduulin toimintaa verrataan moduulin suunnitteluun ja arkkitehtuurisuunnittelun tuloksiin, tavallisimmin tekniseen määrittelydokumenttiin. Testauksen suorittaa yleensä moduulin toteuttaja.Testauksen suorittamiseksi voidaan joutua toteuttamaan testipelejä.[1]

Testipeliin voi kuulua ohjelman ympäristöä simuloivia osia, tavallisimmin testiajureita ja tynkämoduuleita.

-Testiajurit mahdollistavat moduulin toteuttamien palveluiden kutsumisen ja tulosten tarkastelun.

-Tynkämoduulit puolestaan korvaavat testattavan moduulin tarvitsemat muut moduulit, jos niitä ei ole vielä olemassa.

4.2. Integrointitestaus

Integrointitestauksessa yhdistellään yhteen moduuleita tai moduuliryhmiä siis (Osajärjestelmiä). Painopiste on moduulien välisten rajapintojen toimivuuden tutkimisessa. Testauksen tuloksia verrataan tavallisemmin tekniseen määrittelyyn. Integrointitestaus etenee usein rintarinnan moduulitestauksen kanssa ja sitä onkin usein tarpeetonta tarkastella erillään moduulitestauksesta.

Integrointi etenee tavallisesti kokoavasti (Buttom up) alimman tason moduuleista ylöspäin. Jäsentävässä eli osittavassa (top down) integroinnissa etenemissuunta on päinvastainen. Järjestelmä testauksessa tarkastelun kohteena on koko järjestelmä ja tuloksia verrataan määrittelydokumentaatioon (ohjelmiston toiminnalliseen määrittelyyn) ja asiakas dokumentaatioon. Järjestelmätestaukseen voi lisäksi liittyä kenttätestaus ja hyväksymistestaus. Järjestelmätestauksen suorittajina pitää olla kehitystyöstä mahdollisimman riippumattomia testaajia. Järjestelmä testauksessa testataan myös järjestelmän ei-toiminnalliset ominaisuudet:kuormitustestit,luotettavuustestit,asennustestit,käytettävyystestit.

Huom. Mitä korkeammalla V-mallin testaustasolla ollaan, sen kalliimmaksi virheiden korjaus tulee. Virheiden korjaus voi myös aiheuttaa uusia virheitä. Kun esimerkiksi järjestelmätestauksessa havaittu virhe korjataan, voi korjaus aiheuttaa muutoksia useisiin moduuleihin ja erilaisiin kompleksisiin toiminnallisiin järjestelmän osiin. Siksipä moduulit pitää vielä testata moduulitestauksella, ja lopuksi järjestelmäntestauksella eli regressiotestauksella. Sen suorittaminen on erittäin kallista ellei testausta saada automatisoitua.[1]

4.3. Alfa- ja Beta-testaus

Alfatestauksella tarkoitetaan yleensä asiakkaan suorittamaa testiä toimittajan tiloissa, ja beta-testauksella asiakkaan omissa tiloissaan itsenäisesti suorittamaa testausta. Asiakaskunta otetaan huomioon parhaiten Käytettävyys testauksella. Sillä pyritään varmistamaan, että käyttäjä selviytyy mahdollisimman hyvin tehtävistä, joiden suorittamiseksi järjestelmää ollaan rakentamassa. Käytettävyys testaus on yleensä käyttöliittymän testausta. [1]

5. Testauksen riittävyyden arviointi

Riittävän testauksen määrää on varsin vaikea arvioida, sillä varsinkin järjestelmä testauksessa voidaan testata "loputtomasti". Yleensä raha ja aika ovat rajoittava tekijä, sillä aikataulun venyminen voi tulla kalliiksi, jos tuote myöhästyy markkinoilta liikaa. Toisaalta vikojen määrä valmiissa tuotteessa ei saisi olla liian korkea, ne vaikuttavat yrityksen maineeseen, ja vikojen korjaaminen jälkikäteen on kallista. Nykyään tietenkin on internet, jossa on kohtuullisen halpaa jaella paikkauksia ainakin pahimpiin virheisiin. Nykyään erittäin harva ohjelmisto (valmisohjelmisto) julkaistaan kerralla täysin valmiina.

Testaussuunnitelmassa tulisi aina asettaa hyväksymiskriteerit testauksen lopettamiseksi. Järjestelmätestauksessa kriteeri voi olla kumulatiiviseen löydettyjen virheiden määrään, eli kun virhekäyrä tasaantuu, testaus voidaan lopettaa.

Moduulitestauksessa tarvittavan testauksen määrää voidaan yrittää arvioida mutkikkuus mitoilla, ja riittävyyttä kattavuusmitoilla ja virheitä kylvämällä. Mutkikkuusmittojen avulla pyritään päättelemään moduulin monimutkaisuutta, ja näin löytämään paljon testausta vaativat moduulit. Tunnetuimmat mutkikkuusmitat ovat Halsteadin mitta ja McCaben syklomaattinen numero. Halsteadin mitta lasketaan laskemalla ohjelmassa käytettyjen varattujen sanojen ja +,-,yms. lukumäärä N, ja tämän jälkeen eri operaattorien lukumäärä n. Mittaluku ohjelmalle on N(log2)n. McCaben numero lasketaan lisäämällä funktion kontrolliverkon haarautumiskohtien lukumäärään ykkönen ja näin saatu luku kuvaa funktion kontrolliverkon monimutkaisuutta. Lukua voidaan pitää minimimääränä testitapauksia, joilla funktioa on testattavana.

Mutkikkuusmittojen hyödyistä on ristiriitaisia tietoja, ja asiantuntijat esittävät niistä eriäviä mielipiteitä. McCaben syklomaattinen numero on yleisemmin käytössä, Halsteidin mitalla on lähinnä historiallista merkitystä.Kattavuusmitoilla yritetään varmistaa, että testiaineistolla saadaan käytyä ohjelman kaikki osat läpi. Mittoja ovat lause,päätös ja ehtokattavuus ja näiden yhdistelmät. Lausekattavuudella tarkoitetaan, että jokainen lause suoritetaan vähintään kerran. Se saattaa vaikuttaa yksinkertaiselta, mutta se on käytännössä varsin vaikeaa ja todellinen kattavuus nousee harvoi yli 80%.

Päätöskattavuudessa edellytetään että jokainen ehto saa testattaessa molemmat arvonsa. Ehtokattavuudessa edellytetään kaikkien ehtojen saavan kaikki arvonsa. Jos vaaditaan ehto ja päätöskattavuutta, mittaa kutsutaan päätös/ehtokattavuudeksi. Moniehto kattavuus edellyttää testausta kaikkien ehtojen kaikilla yhdistelmillä.Polkutestauksessa ohjelmaa tarkastellaan verkkona jossa solmuina ovat ohjelman haarautumat ja testattaessa pyritään käymään kaikki polut ohjelman läpi. Se onnistuu yleensä vain pienen osan sisällä, koska polkujen määrä kasvaa räjähdysmäisesti ohjelmakoon kasvaessa.

Testauksen kattavuus saadan varsin korkeaksi moduulitasolla, mutta valmista versiota testattaessa lausekattavuus nousee harvoin yli 50 %. Virheiden kylvämisellä tarkoitetaan tahallisten virheiden kylvämistä koodiin. Testauksessa voidaan löydettyjen oikeiden virheiden ja kylvettyjen virheiden suhteesta yrittää päätellä montako virhettä on löytämättä.Esim. jos kylvetyistä on löydetty puolet,

luultavasti oikeistakin löydetty puolet. Käytännössä menetelmää ei juurikan käytetä, koska se lisää kustannuksia lisätyön muodossa ja välttämättä kaikkia virheitä ei muisteta poistaa.

6. Staattinen analyysi

Etsitään virheitä ohjelmakoodista sitä tutkimalla joko käsin tai ohjelmalla. (Esim. kääntäjä). Näin voidaan löytää esim. alustamattomien muuttujien ja väärien indeksien käyttö ja näistä johtuvat virheet. Staattisessa analyysissä käytetään myös ohjelmamittoja, joilla arvioidaan koodin monimutkaisuutta. McCaben syklomaattisen numeron = ehdollisten ohjelmahaarojen lkm + 1, arvon tulisi olla alle 15, koska käytännössä on havaittu funktion olevan vaikea hahmottaa ja testata muuten.

6.1. Koodin tarkastelu, läpikäynti ja pöytätestaus

Ohjelmakoodin tarkastelu käsin on havaittu varsin tehokkaaksi testausmenetelmäksi varsinkin, kun tiedetään, mitä etsiä. Useimmiten tällä tavoin voidaan paljastaa tyypillisiä ja tunnettuja virheitä, joita automaattisesti ei välttämättä havaita. Suuremmissa projekteissa koodin tarkastelu tapahtuu ryhmätyönä. Jokainen testaaja tutkii ja käy läpi koodin ensin itsekseen tehden samalla muistiinpanoja. Tämän jälkeen pidetään palaveri, jossa koodista sitten keskustellaan yhdessä. Koodin analysoinnissa ei keskitytä pelkästään tyypillisiin virheisiin, vaan usein kiinnitetään huomiota mm. ohjelmointityyliin. Samalla voidaan jo esittää korjausehdotuksia. Kaikista kommenteista ja päätöksistä pidetään kirjaa. Lopuksi tuotokset jo hyväksytään, hyväksytään korjauksin tai hylätään. Hylätyt tuotokset korjataan ja hyväksytetään uudelleen johtoryhmällä.

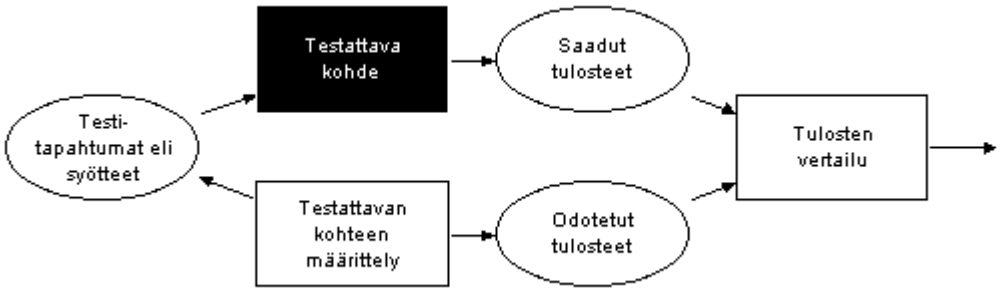
Kokemusten perusteella on havaittu, että nämä metodit ovat tehokkaampia kuin perinteisemmät pöytätestausmenetelmät, joissa usein itse ohjelmoija kävi koodia läpi. Käytännössä näillä metodeilla voidaan löytää 30 - 70 % loogisista suunnittelu- ja koodausvirheistä.[4]

Pöytätestauksessa tarkistetaan, että algoritmi toimii kuten sen uskotaan toimivan. Tarkistuksen suorittaa yleensä yksi henkilö, joten monet virheistä jäävät usein huomaamatta.

7. Dynaaminen analyysi

7.1 Black box-testaus

Black box-testaus perustuu testattavan systeemin (esim. ohjelma) tai komponentin (esim. funktio) input-output käyttäytymiseen. Black box-testauksessa ei välitetä testattavan kohteen rakenteesta tai sisällöstä, vaan tutkittavana ovat kohteen tulosteet (output) erilaisilla syötearvoilla (input). Testaajalle kohde on siis jokin tuntematon "musta laatikko", black-box. Testattavan kohteen oikeellisuutta tarkastellaan vertaamalla saatuja tulosteita haluttuihin tai odotettuihin tulosteisiin. Testitapaukset johdetaan aina kohteen määrittelyn perusteella.



Kuva 2. Black box testaus.

Seuraavassa on lueteltu muutamia black-box-metodeja, joiden mukaan testiaineisto (syötteet) voidaan valita.

Testitapausten jako ekvivalenssiluokkiin (Equivalence partitioning)

Jotta testitapausten lukumäärä saataisiin mahdollisimman pieneksi, mutta silti mahdollisimman kattaviksi, on yksi mahdollisuus jakaa syötejoukko ns. ekvivalenssiluokkiin. Testattavan kohteen määrittelyn perusteella voidaan nähdä, mitkä syötteet vaikuttavat kohteeseen samalla tavalla (ja tuottavat samanlaisia tulosteita). Nämä syötteet muodostavat oman ekvivalenssiluokkansa. Ekvivalenssiluokat voidaan vielä jakaa "oikeisiin" (valid) sekä "vääriin" (invalid). Tämän jälkeen syötteet eli testitapaukset tulisi valita kattavasti sekä oikeista että vääristä ekvivalenssiluokista. Esimerkkinä oikeista ja vääristä ekvivalenssiluokista voidaan ajatella ohjelmaa, joka kysyy: "Oletko varma (K/E) ?". Tällöin oikeat ekvivalenssiluokat muodostuisivat kirjaimista K ja k, sekä kirjaimista E ja e. Väärän ekvivalenssiluokan puolestaan muodostaisivat kaikki muut mahdolliset syötteet.

Reuna-arvoanalyysi (Boundary value analysis, BVA)

Reuna-arvoanalyysissä testitapauksiksi valitaan nimensä mukaan rajoilla (reunoilla) olevia (äärimmäisiä) arvoja. Nämä arvot voivat olla esimerkiksi mahdollisimman suuria tai pieniä, pitkiä tai lyhyitä ja niin edelleen. Reuna-arvoanalyysi perustuu olettamukseen, että jos testattava kohde epäonnistuu joillakin rajojen arvoilla, se yleensä epäonnistuu myös arvoilla, jotka kuuluvat rajojen sisään. Tällä tavoin saadaan taas testitapausten lukumäärää pienennettyä.

Virheen arvaus (Error guessing)

Testaajalla on usein jo kokemusperäistä tietoa erityyppisistä ongelmia aiheuttavista syötteistä, jotka eivät sisälly esimerkiksi edellä mainittujen metodien antamiin tai ehdottamiin syötteisiin. Esimerkiksi 0 on syöte, joka ei välttämättä kuulu BVA:n rajoihin, mutta on aiheuttanut ja aiheuttaa virheellisiä käsittelyjä useissa ohjelmissa. Virheen arvaus-metodi perustuukin siihen, että listataan mahdolliset mieleen tulevat testitapaukset, jolla testattava kohde saadaan epäonnistumaan. Tämän jälkeen suoritetaan testiajot näillä syötteillä. Tämä ei itsessään ole kovinkaan kattava black box-testausmetodi, mutta se on hyvä lisä muille metodeille.

Sattumanvarainen testaus (Random testing)

Yksi tapa (joskin usein huono sellainen) on valita syötejoukosta umpimähkään tietty määrä syötteitä ja suorittaa testiajot näillä. Tällä tavoin, varsinkin valitun syötejoukon ollessa suhteellisen pienen, ei voida saada varmuutta siitä, että kohde tuli edes kohtuullisesti testattua.

7.1.1. Mustalaatikko testaamisen hyviä puolia [5]

Testit voidaan yleensä helposti uusia (Melzer 1996). Ensimmäisellä testikerralla käytetty syöte voidaan sellaisenaan syöttää ohjelmalle uudestaan, esimerkiksi ohjelman korjauksen jälkeen ja vertailla saatuja tuloksia toisiinsa ja ohjelman omiin määrityksiin.

Ympäristö jossa ohjelma ajetaan tulee myös testatuksi (Melzer 1996). Ajettaessa ohjelmaa tietyssä ympäristössä, voidaan varmistaa, ettei ko. ympäristö rajoita tai estä ohjelman toimintaa. Tämä voi olla myös haittana itse ohjelman testaamiselle: Testaaja voi erehtyä luulemaan ympäristön aiheuttamaa virhettä testattavasta ohjelmasta johtuvaksi.

Ohjelman testaajan ei välttämättä tarvitse olla ohjelmoinnin asiantuntija. Kuka tahansa voi suorittaa jonkinlaista mustalaatikko testausta, jos hänellä on käsitys siitä, mitä mustalaatikko testaaminen on. Käyttäjäläheisimmillään mustalaatikko testaus voi olla puhdasta käyttäjätestausta, jossa käyttäjä yrittää saada oikeaksi arvelemillaan syötteillä aikaan haluamansa tulosteet.

7.1.2. Mustalaatikko testaamisen huonoja puolia [6]

Kaikkia ohjelman ominaisuuksia ei kyetä testaamaan (Melzer 1996). Ohjelmaa ei voi muuttaa tietyn rakenteen, yksinkertaisimmillaan muuttujan testaamista varten. Huomaamatta saattaa jäädä esim. tietyllä arvoalueella tapahtuva vika, jonka muussa ohjelman kohdassa tapahtuva päinvastainen vika korjaa.

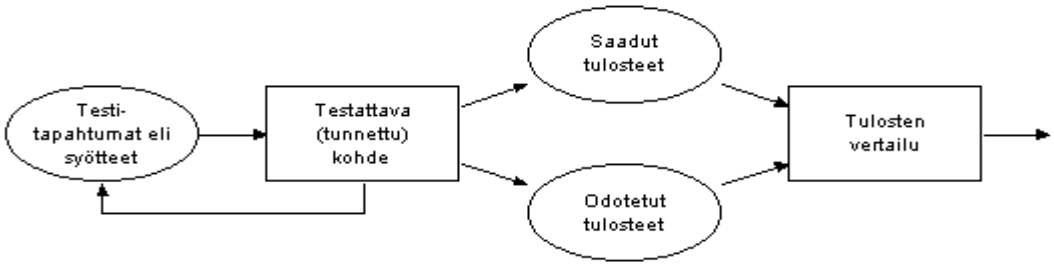
Virhetoiminnon syytä ei löydetä. Testaaja tietää vain ohjelman määritykset, antamansa syötteen ja saamansa tulosteen. Hänellä on mahdollisuus vain havaita virhetoiminto.

Testauksen uusimisessa on ongelmansa. Montako uutta virhetoimintaa löydetään ajamalla jo aikaisemmin ajettu syöte? Jos uusien löydettyjen virheiden määrä jää vähäiseksi uusimalla jo suoritettu testitapaus, koko testi muuttuu tehottomammaksi (Kaner 1997).

Mustalaatikko testaaminen aiheuttaa lisäkustannusriskin (Kaner 1997). Mustalaatikko testaaminen voidaan suorittaa vasta valmiille tuotteelle tai sen osalle. Tällöin testaamisajankohta on lähellä ohjelman aiottua julkaisupäivää. Julkaisupäivän lähestyessä ohjelman parissa työskentelee yleensä useampia ihmisiä, osan viimeistellessä koodia, osan testatessa, joidenkin keskittyessä dokumentointiin. Vasta tässä vaiheessa löydetty virhe voi aiheuttaa sen, että muuta jo valmista materiaalia joudutaan jopa hävittämään.

7.2. White box-testaus

White box-testauksessa testitapaukset johdetaan kohteen, esimerkiksi koko ohjelman sisäisestä rakenteesta ja logiikasta. Tavoitteena on valita testitapaukset siten, että kaikki kohteen (esimerkiksi ohjelman tai funktion) haarat ja ohjelmapolut tulisi käytyä läpi.



Kuva 3. White box-testaus

Lasilaatikkotestauksessa testidatan valinnalla pyritään siihen, että testattava ohjelma tulee testattua mahdollisimman kattavasti. Testauksen kattavuutta (coverage) voidaan kuitenkin arvioida eri tavoilla, jotka samalla määrittelevät kuinka testidata valitaan. Seuraavassa on lueteltu muutamia white box-metodeja, joiden kriteerien mukaan testiaineisto eli syötteet voitaisiin valita.

Lauseiden kattavuus (Statement coverage)

Lauseiden kattavuus-metodin mukaan testitapausten tulisi kattaa (suorittaa ainakin kerran) kaikki ohjelman lauseet. Käytännössä tämä tapa ei ole paras mahdollinen, koska testitapausten määrä voi ohjelman koon ja monimutkaisuuden vuoksi kasvaa liian suureksi. [1]

Haarojen kattavuus (Branch / Path coverage)

Haarojen kattavuus-metodin mukaan kaikki ohjelman haarojen vaihtoehdot tulisi pystyä suorittamaan valituilla testitapauksilla. Haaralla (branch) tarkoitetaan tässä ohjelman kohtaa, jossa on kaksi tai useampia vaihtoehtoisesti suoritettavia lauseita (if-else, switch-case jne.). Käytännössä tämä on mahdollista vain yksittäisten funktioiden tapauksessa, jolloin eri polkuvaihtoehtoja on vielä testauksen kannalta järkevä määrä. Suurempien ohjelmarakenteiden ollessa kyseessä kasvaa mahdollisten suorituspolkujen määrä kohti ääretöntä.

Ehtokattavuus (Condition coverage)

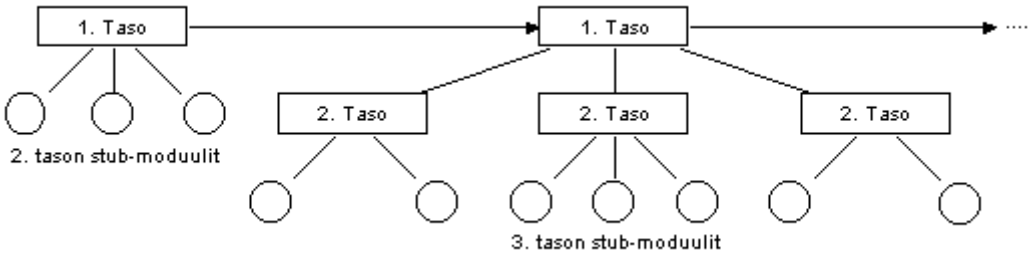
Ehtokattavuudessa annetaan päätösten tekemiseen vaadittaville ehdoille niiden kaikki eri arvot (TRUE ja FALSE). [1]

Päätösten ja ehtokombinaatioiden kattavuus (Decision coverage)

Päätösten kattavuus (Decision coverage) -metodin mukaan testitapaukset tulisi valita siten, että kaikki ehdon sisältävät lauseet (esim. if-lause) saisivat kaikki mahdolliset tulokset (esim. TRUE ja FALSE). Useasti ehdolliset lauseet sisältävät tai peittävät toisia ehdollisia lauseita ja näin ollen kaikki mahdolliset kombinaatiot eivät tule välttämättä testatuiksi. Ehtokombinaatioiden kattavuus (Multiple-condition coverage) -metodi paikkaakin tämän puutteen, sillä sen mukaan testitapausten tulisi kattaa kaikki mahdolliset tuloskombinaatiot. Käytännössä kaikkien tuloskombinaatioiden läpikäynti on kuitenkin usein mahdotonta.

7.3. Top-down

Top-down -strategiassa testaus aloitetaan ohjelman korkeimman tason moduuleista ja vasta sitten edetään alempien tasojen moduulien ja komponenttien testaukseen. Jotta testattavaa moduulia alempien tasojen moduuleja voitaisiin simuloida, tarvitaan ns. "tyhmiä" moduuleita tai moduulin pätkiä (**module stubs, stubs**). Nämä **stub-moduulit** ovat hyvin yksinkertaisia, eivätkä sisällä mitään monimutkaisempaa toiminnallisuutta. Kun testaus etenee alemmalle tasolle, stub-moduulit korvataan oikeilla moduuleilla.



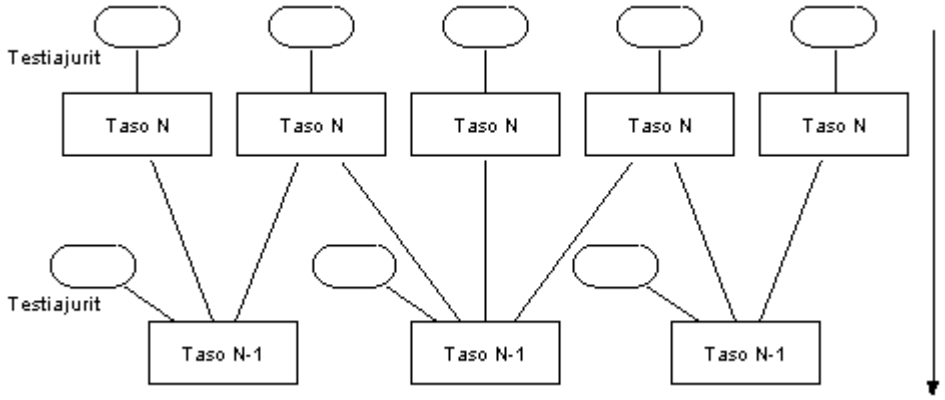
Kuva 4. Top-down-testaus

Top-down-testauksen etuja ovat varhaisessa vaiheessa saatava alustava, toimiva ohjelma (jo psykologisestikin kannustavaa suunnittelussa mukana oleville) sekä kahden mahdollisesti erillisen testivaiheen, integraatio- ja systeemitestauksen, yhdistyminen.

Suurin haittatekijä on "ylimääräisten" stub-moduulien luominen, joka vie aikaa. Lisäksi testiaineiston valinta voi tuottaa vaikeuksia (esim. monimutkaisten moduulien stub-versioissa, jotka eivät välttämättä kuvaa täysin oikean moduulin käytöstä) ennen kuin oikeat moduulit on lisätty.

7.4. Bottom-up

Bottom-up -strategiassa testaus aloitetaan alimman tason moduuleista (moduuleista, jotka eivät käytä tai kutsu muita moduuleja). Testaus etenee ylemmille tasoille, kunnes koko ohjelma on testattu. Kuten top-down-testauksessa jouduttiin luomaan stub-moduuleita, bottom-up testauksessa joudutaan luomaan ns. **testiajureita** (**test drivers**), joilla alemman tason komponentteja voidaan suorittaa ja testata. Käytännössä useamman testiajurin sijaan, luodaan yleensä testauksen mahdollistava ympäristö, joka toimii kaikkien moduulien yhteisenä testiajurina.



Kuva 5. Bottom-up -testaus

Bottom-up -strategian etuna on testiaineiston kohtalaisen helppo valinta. Myös siinä tapauksessa, että kriittisimmät tai virhealtteimmat moduulit sijoittuvat alemmille tasoille, on bottom-up testaus hyvä valinta.

Suurin haitta bottom-up-testauksessa on se, että mitään valmista ohjelmaa ei ole olemassa ennenkuin viimeinenkin ylimmän tason moduuli on testattu. Testausta ei myöskään voida aloittaa ennen kuin alimmankin tason moduulit on suunniteltu ja määritelty.

8. Ohjelmistotestauksen työkaluja

Kuten ohjelmistotestaus, myös ohjelmistotestauksessa käytettävät työkalutkin jaetaan usein karkeasti sen mukaan, suoritetaanko testattavaa kohdetta testauksen aikana vai ei. Testauksessa käytetyt automatisoidut apuvälineet luokitellaan usein siis staattisen analyysin ja dynaamisen analyysin ja testauksen työkaluihin. Kolmannen luokan muodostaa testauksen hallinnan työkalut.

8.1. Staattisen analyysin työkalut

Staattisen analyysin työkalujen tarkoitus on löytää virheitä ohjelman koodista ilman minkäänlaista ohjelman suoritusta. Triviaali esimerkki staattisen analyysin apuvälineestä on kääntäjä, joka koodin kääntämisen yhteydessä ilmoittaa useista koodista havaituista virheistä.

Staattisia analysoijia käytetään yleisimmin ohjelmamittojen laskemiseen ja ohjelman rakenteen tarkempaan analysointiin. Ohjelmamitoista voidaan päätellä ja ennustaa koodin mahdollisia ongelmakohtia sekä analysoida ohjelman kompleksisuutta.

8.2. Dynaamisen analyysin / testauksen työkalut

Jotta suoritusaikainen testaus ja sen analysointi olisi mahdollista, tarvitaan ohjelman pohjalle jonkinlainen ympäristö, joka mahdollistaa tulosten havainnoinnin. Tällainen ympäristö muodostetaan yleensä testiajureiden avulla. Testiajuri voi olla joko itsenäinen ohjelma tai testattavaan ohjelmaan yhdistetty (itsenäinen) osa.

Usein testiajureiden avulla voidaan myös nauhoittaa erityyppistä informaatiota testausta suoritettaessa. Nauhoitukset suoritetaan yleensä erillisiin lokitiedostoihin, joista haluttua informaatiota sitten voidaan tarkastella. Yhä useammin, varsinkin graafisten käyttöliittymien tapauksessa, testiajureilla saadaan myös tietoa testattavan ohjelman vaikutuksesta ympäristöönsä, esimerkiksi tietoa ohjelman muistin ja resurssien käytöstä.

Itse testaus suoritetaan yleensä joko käsin (koko ajan syöttäen testitapahtumia) tai luodulla testiskriptillä, joka suorittaa tai syöttää testattavalle ohjelmalle siinä luetellut testitapahtumat. Testiskriptien etuna on yhtäläisen testauksen nopeus ja helppous toistettaessa jo aikaisemmin suoritettuja testejä esimerkiksi regressiotestauksessa.

Seuraavassa on lueteltu joitakin tyypillisiä, tiettyyn tehtävään erikoistuneita dynaamisen testauksen työkaluja. Simulaattoreilla tarkoitetaan työkaluja, joilla simuloidaan (väliaikainen) suoritusympäristö esimerkiksi ohjelman osille ja funktioille, joita sinällään ei voida suorittaa.

Testauksen kattavuutta ja testausta analysoivat työkalut antavat tietoa testauksen laadusta (kattavuudesta). Niiden avulla pidetään kirjaa jo testatuista ja testaamattomista ohjelman osista.

Suorituskykyä ja rasitussietoisuutta analysoivilla työkaluilla voidaan tarkastella esimerkiksi ohjelman tehokkuutta ja simuloida useiden käyttäjien ympäristöjä.

Dynaamisen analyysin työkaluihin luetaan kuuluvaksi usein myös testiaineistogeneraattorit. Näiden työkalujen tarkoituksena on siis automatisoida testiaineiston luontia. Testitapahtumat johdetaan usein suoraan ohjelman koodista saadun informaation perusteella tai käyttämällä hyväksi valmiita tietokantoja.

8.3. Testauksen hallinnan työkalut

Testauksen hallinnan työkalut ovat kehitetty automatisoimaan testauksen suunnittelua, analysointia, dokumentointia ja raportointia. Markkinoilla on tarjolla ohjelmia ja ohjelmistoja, joilla luodaan oma ympäristökokonaisuus testauksen hallintaan. Tyypilliset hallintaympäristöt helpottavat testauksen organisointia, analysoivat testituloksia, tekevät yhteenvetoraportteja, lähettävät yhteenvetoja prosessissa mukanaoleville.

8.4. Testausohjelmistot ja -järjestelmät

Ohjelmistotestaukseen on kasvavassa määrin tarjolla suurempia testausohjelmistokokonaisuuksia. Testausjärjestelmät kattavat usein lähes kaiken toiminnan aina suunnittelusta projektin loppuun saakka. Suurempien testausjärjestelmien etuna on mm. eri testaus- ja prosessin vaiheiden saumattomuus, prosessin hallinnan helppous, dokumentoinnin yhtenäisyys ja useimmin monen käyttäjän (verkko)tuki.

Lähteet

[1] Haikala Ilkka, Marijärvi Jukka, Ohjemistotuotanto, Suomen ATK-kustannus Oy, 1997

[2] Jaakko Kyrö http://www.cs.helsinki.fi/u/jkyro/Testaus.html

[3] http://erin.mit.jyu.fi/pako/kurssit/ot2000/l12/lect12/lect12.html

[4] http://www.soberit.hut.fi/tik-76.115/00-01/palautukset/groups/Vym/ps/laatusuunnitelma.pdf

[5] Melzer, Ingo. Black Box or Functional Testing.
http://www.mathematik.uni-ulm.de/~melzer/thesis/node21.html

[6] Kaner, Cem. Improving the Maintainbility of Automated Test Suites.
http://www.kaner.com/lawst1.htm