SAIMAAN AMMATTIKORKEAKOULU
Tekniikka Lappeenranta
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

Laura Sainio

OHJELMISTOTESTAUKSEN MENETELMÄT JA TYÖVÄLINEET

Lukumateriaali 2009

Saatteeksi

Tämä dokumentti on osa opinnäytetyötäni, jonka tehtävänä oli tuottaa opetusmateriaalia Saimaan ammattikorkeakoulun ohjelmistotestausta käsittelevälle opintojaksolle. Dokumentti sisältää ohjelmistotestauksen teoriaa, ja toimii lukumateriaalina täydentäen allekirjoittaneen laatimaa diasarjaa testauksesta.

Laura Sainio

VERSIOHISTORIA

Pvm	Versio	Muutokset
21.9.2009	0.1	Muutettu opinnäytetyöraportista opintomateriaaliksi, uusi johdantokappale
22.9.2009	0.11	Lisätty termistöä
28.9.2009	0.12	Lisäyksiä lukuun 5.3 ja 5.3.4
29.9.2009	0.13	Muutoksia lukuun 6.3
1.10.2009	0.14	Lisäyksiä lukuun 6.3
6.10.2009	0.15	Kuvien siistintää, luku 8
8.10.2009	0.16	Aloitettu luku 9
13.10.2009	0.17	Muutoksia lukuihin 8.1 ja 8.2
19.10.2009	0.18	Kuvia lukuun 8.2, lisätty perintä ja kooste
23.10.2009	0.19	Luku 9, 9.1 ja 9.2 alkua
27.10.2009	0.20	Lukua 9.2
28.10.2009	0.21	Lukuja 9.2.3 – 9.2.5
2.11.2009	0.22	Lisäyksiä lukuun 4.6.5
3.11.2009	0.23	Lisäyksiä lukuun 4.6.5
5.11.2009	0.23	Luku 9.2 valmiiksi
13.11.2009	0.24	Muutoksia lukuun 9, luvut 9.1–9.3
15.11.2009	0.25	Kuvaluettelo, kuvien numeroinnin päivitys
16.11.2009	0.26	Muotoilun tarkistamista
15.3.2010	1.0	Viimeiset tarkistukset

SISÄLTÖ

1 JOHDANTO 11
1.1 Ohjelmistotuotannon erityispiirteet 12
1.2 Ohjelmistotuotanto ja laatu 13
2 HISTORIA 15
2.1 Testauksen historia 15
2.2 Ohjelmistotestauksen ammattimaistuminen 17
3 NÄKÖKULMIA TESTAUKSEEN 18
3.1 Testauksen tarkoitus 18
3.2 Virheet 19
3.3 Testauksen määritelmät 20
3.4 Testauksen koulukunnat 22
3.4.1 Analyyttinen koulukunta 23
3.4.2 Standardilähtöinen koulukunta 24
3.4.3 Laatulähtöinen koulukunta 26
3.4.4 Kontekstiohjattu koulukunta 26
3.4.5 Ketterän testauksen koulukunta 27
4 TESTAUS OSANA OHJELMISTOTUOTANTOA 29
4.1 Vesiputousmalli ja V-malli 29
4.2 Virheiden luokittelu 31
4.3 Verifiointi ja validointi 32
4.4 Yksikkötestaus 33
4.4.1 Testipeti, ajuri ja tynkä 35
4.4.2 Test Driven Development 35
4. 5 Integrointitestaus 37
4.5.1 Jäsentävä integrointi 39
4.5.2 Kokoava integrointi 40
4.5.3 Jatkuva integrointi 41
4.6 Järjestelmätestaus 41
4.6.1 Toiminnallisuustestaus 42
4.6.2 Volyymitestaus 43
4.6.5 Tietoturvatestaus 44
4.6.6 Suorituskykytestaus 46
4.6.7 Resurssien käytön testaus 46
4.6.8 Kokoonpanon testaus 46
4.6.9 Yhteensopivuustestaus 47
4.6.10 Asennettavuustestaus 47
4.6.11 Luotettavuustestaus 47
4.6.12 Toipuvuustestaus 48
4.6.13 Ylläpidettävyystestaus 48
4.6.14 Dokumentoinnin testaus 49
4.6.15 Prosessin testaus 49
4.6.16 Järjestelmäintegrointitestaus 49
4.7 Hyväksymistestaus 49
4.7.1 Alfatestaus 50
4.7.2 Betatestaus 51
4.8 Regressiotestaus 51
4.9 Ylläpitotestaus 52
5 TESTAUKSEN KÄYTÄNNÖT 54

5.2 Staattiset testausmenetelmät 54
5.2.1 Katselmoinnit 55
5.2.2 Katselmointityyppejä 57
5.2.3 Staattinen analyysi 59
5.3 Dynaamiset testausmenetelmät 60
5.3.1 Rakenteen testaus ja lasilaatikkotekniikat 61
5.3.3 Käyttäytymisen testaus ja mustalaatikkotekniikat 64
5.3.4 Harmaalaatikkotestaus 79
5.3.5 Tutkiva testaus 80
6 TESTAUKSEN HALLINTA 83
6.1 Organisointi 83
6.2 Suunnittelu 85
6.3Testaussuunnitelman laatiminen 86
6.3.1 Testauksen dokumentointi 86
6.3.2 Testauksen dokumentit standardin 829 mukaan 87
6.4 Havaintojen raportointi 91
6.5 Testauksen päättäminen 93
7 KETTERÄT MENETELMÄT JA TESTAUS 96
7.1 Ketterät periaatteet ja menetelmät 96
7.2 Ketteryys testauksessa 97
7.3 Ketteryyden vahvuudet ja ongelmat 98
8 OLIOTESTAUS 100
8.1 Ohjelmistojen rakenteen kehitys 100
8.2 Olio-ohjelmoinnin käsitteitä 102
8.3 Oliotestauksen erityispiirteitä 109
8.3.1 Oliojärjestelmien yksikkötestaus 110
8.3.2 Perinnän vaikutukset testaukseen 110
8.3.3 Polymorfismin ja dynaamisen sidonnan vaikutukset testaukseen 111
8.3.4 Tilariippuvuus 111
8.3.5 Tiedon kapseloinnin vaikutus testaukseen 112
8.3.6 Abstraktien luokkien vaikutus testaukseen 112
8.3.7 Poikkeuskäsittelyn vaikutus testaukseen 112
8.3.8 Rinnakkaisuuden vaikutus testaukseen 113
8.4 Oliojärjestelmän testaus 113
8.4.1 UML-mallit testitapauksien pohjana 114
8.4.2 Olioiden testaus 117
8.4.2 TDD ja Mock-objektit 119
9 TESTITYÖKALUT JA AUTOMAATIO 120
9.1 Automaation vaikutus testauksen suunnitteluun 121
9.1.1 Automaation vaikutus ohjelmiston kehitysprosessiin 123
9.1.2 Automaation kohteet 124
9.2 Työkalujen valinta ja käyttöönotto 126
9.2.1 Ennen automaatiota 126
9.2.2 Projektiryhmän muodostaminen 127
9.2.2 Automatisoidun testauksen vaiheet 128
9.3Testaustyökalujen tyypit 142
9.3.1 Hallinnan työkalut 142
9.3.2 Suunnittelun ja tarkastuksen työkalut 147
9.3.3 Staattisen analyysin työkalut 148
9.3.4 Dynaamisen analyysin ja virheiden etsinnän työkalut 150

9.3.5 Kattavuustyökalut 152
9.3.6 GUI-ajurit 154
9.3.7 Kuormituksen, suorituskyvyn ja simuloinnin työkalut 155
9.3.8 Suorituksen ja vertailun työkalut 157
9.3.9 IDE:t ja koostamistyökalut 164
KUVAT 170
LÄHTEET 173

TERMIT JA LYHENTEET

Bugi Virhettä kuvaava puhekielinen ilmaus, ”bug”.

CASE Computer-Aided Software Engineering. Tietokoneavusteinen tietojärjestelmien kehittäminen. CASE-välineellä tarkoitetaan ohjelmistopohjaista apuvälinettä, joka tukee jotain systeemityön vaihetta.

Debug Debuggaus, virheenjäljitys ja korjaus.

Dynaaminen testaus
Ohjelmiston testaaminen ohjelmakoodia suorittamalla.

Ei-toiminnallinen testaus
Testausta, joka kohdistuu ei-toiminnallisiin ominaisuuksiin, jotka vaikuttavat ohjelmistoon laatuun, mutta eivät ole liitettävissä suoraan toimintoon tai toimintoryhmään ohjelmistossa, esim. suorituskyky- ja luotettavuustestaus.

FiSTQB ISTQB:n Suomen aluejärjestö.

GUI Graphic User Interface. Graafinen käyttöliittymä.

Harmaalaatikkotestaus
Musta- ja valkolaatikkotestauksen välimuoto, jossa käytetään hyväksi tietoa ohjelman toteutusperiaatteista.

Hyväksymistestaus
Loppukäyttäjän suorittama testaus jossa arvioidaan tuotteen kykyä vastata asiakasvaatimuksiin.

Häiriö Failure. Vikaantuminen, ohjelmiston poikkeama odotetusta toimituksesta, palvelusta tai tuloksesta.

IDE Integrated Development Environment. Ohjelmointiympäristö.

IEEE Institute of Electrical and Electronics Engineers. Maailman suurin ja merkittävin tekniikan alan järjestö, jonka toimintaan kuuluu muun muassa tekniikan alan julkaisutoiminta, standardien luominen sekä konferenssi- ja koulutustoiminta

IEEE 829 IEEE Standard for Software Test Documentation, versiot vuosilta 1983, 1998 ja 2008. Testausdokumentaation standardi.

IEEE 1008 IEEE Standard for Software Unit Testing vuodelta 1987. Yksikkötestauksen standardi.

IEEE 1028 IEEE Standard for Software Reviews vuodelta1997. Katselmointistandardi.

Integrointitestaus
: Testaus jonka tarkoituksena on testata moduulien välisten rajapintojen toimintaa.

ISEB Information Systems Examinations Board. British Computer Societyn tietojenkäsittelyn tutkintolautakunta.

ISTQB International Software Testing Qualifications Board. Tutkintolautakunta, jonka tehtävänä on kehittää kansainvälinen testaajien sertifiointijärjestelmä.

Järjestelmätestaus
: Testaus jonka tarkoituksena on testata koko järjestelmän toimintaa.

Katselmointi
: Ohjelmiston osien tai projektin tilan arviointi, jonka tarkoitus on tunnistaa tuotosten eroavuudet suunnitelmiin nähden sekä tuottaa kehitysehdotuksia.

Ketterä kehitys
: Agile development. Nopeita ja muutoksiin nopeasti vastaavia ohjelmistokehitysmenetelmiä tarkoittava termi.

Lasilaatikkotestaus
: Testausmenetelmä jossa huomioidaan testauksen kohteen toteutus ja rakenne. Käytetään myös nimeä valkolaatikkotestaus.

Mustalaatikkotestaus
: Testausmenetelmä joka perustuu testauksen kohteen ulkoiseen toimintaan, niin, ettei järjestelmän sisäinen toteutus ole selvillä. Ohjelma on ”musta laatikko”, jonka sisään ei voi nähdä. Ks. toiminnallinen testaus.

Mock-objekti
: Korvikeolio, jota voidaan käyttää olioiden rajapintoja testatessa oikean olion sijasta.

Progressiotestaus
: Testausmenetelmä, jossa testataan ohjelmistoon jälkeenpäin lisättyä uutta toiminnallisuutta.

Regressiotestaus
: Testausmenetelmä jolla todennetaan ohjelmistoon tehdyn korjauksen toimivuus ja tutkitaan, onko korjaus aiheuttanut uusia vikoja ohjelmistoon.

Savutesti ”Smoke test”, eli ohjelmistoversion verifiointitestaus. Ilmaisu tulee laitteiston testauksesta: Jos virran kytkentä saa aikaiseksi savun nousemisen laitteesta, ei enempiä testejä tarvitse tehdä. Savutesti tarkoittaa (automatisoitua) testijoukkoa, joka ajetaan säännöllisesti sen testaamiseksi, toimiiko ohjelmistoversio muutosten jäljiltä.

Scrum Yksi ketterän ohjelmistokehityksen menetelmistä.

Skedulointi prosessien suoritusjärjestyksen aikatauluttaminen.

SOA Service Oriented Achitechture. Palvelukeskeinen ohjelmistoarkkitehtuuri

Staattinen testaus
Ohjelmiston testaaminen ohjelmakoodia suorittamatta.

Sulautettu järjestelmä
Tiettyyn tarkoitukseen tehty, suljettu tietokonejärjestelmä, esimerkiksi viihde-elektroniikan tai kodinkoneiden ohjausjärjestelmät.

Sytyke Systeemityön yhdistys, Tietotekniikan liiton (TTL ry) suurin valtakunnallinen teemayhdistys, jonka alaisuudessa toimii yhdeksän osaamisyhteisöä.

Tarkastus Tarkasti määritelty kokouskäytäntö katselmoinnin toteuttamiseen.

Toiminnallinen testaus
Funktionaalinen eli ulkoinen eli mustalaatikkotestaus.

TDD Test Driven Development t. Test Driven Design. Yksikkötestauksen menetelmä, jossa testit kirjoitetaan ennen ohjelmakoodia.

TestausOSY
Sytyke:n alainen testauksen osaamisyhteisö.

UML Unified Modeling Language,. Graafinen mallinnuskieli järjestelmä- ja ohjelmistokehityksen kuvaamiseen.

V&V Verifiointi ja validointi, ohjelmistotuotannon työvaihe, jossa varmistetaan, että ohjelmisto täyttää sille asetetut vaatimukset ja tarpeet.

V-malli Ohjelmistoprosessin elinkaarimalli, jossa testaus on integroitu kaikkiin prosessin osavaiheisiin.

Vesiputousmalli
Perinteinen ohjelmistoprosessin elinkaarimalli.

Vika Fault, defect, ”bug”. Komponentissa tai järjestelmässä oleva virhe, joka aiheuttaa sen, ettei se pysty suorittamaan siltä edellytettävää toimintoa.

Virhe Error, mistake. Ihmisen toimintaa, joka tuottaa väärän tuloksen.

XP Extreme Programming. Eräs ketterän ohjelmistokehityksen menetelmistä.

Yksikkötestaus
Testausvaihe, jossa kohteena on vain yksittäinen moduuli tai komponentti.

Ylläpitotestaus
Asiakkaalle toimitetun järjestelmän muutosten jälkeinen testaus.

Äärellinen automaatti
Käsite, joka viittaa esimerkiksi sovelluksen mallintamiseen eräänlaisena tilojen ja niiden välisten relaatioiden tai siirtymien joukkona. Voidaan puhua myös tilakoneesta.

Ääritestaus
XP-perustainen yksikkötestausmenetelmä, jossa testit laaditaan TDD:n avulla.

1 JOHDANTO

Ohjelmistotekniikka on tekniikan ala, joka muistuttaa matematiikkaa sillä tavoin, ettei sillä ole mitään erityistä sovellusaluetta, vaan sitä voidaan soveltaa lähes mihin tahansa tehtävien automatisointiin liittyvää ongelmaa ratkottaessa. Ohjelmistotyyppejä on lukuisia:

- varus- ja työkaluohjelmistot, kuten esimerkiksi käyttöjärjestelmät, tietokantajärjestelmät, tietoliikenneohjelmistot ja kääntäjät
- teknis-tieteelliset ohjelmistot, esimerkiksi sääanimaatio-ohjelmistot.
- asiantuntijajärjestelmät, eli järjestelmät, joihin on koottu jonkin alan asiantuntemusta, esimerkiksi lääkäreiden diagnostiikkaohjelmat
- kaupallis-hallinnolliset ohjelmistot, esimerkiksi yritysten toiminnanohjausjärjestelmät
- prosessinohjausjärjestelmät, esimerkiksi paperikoneen ohjausjärjestelmät
- sulautetut järjestelmät, esim. televisiota tai pesukonetta ohjaavat ohjelmistot.

Sen lisäksi, että ohjelmistotekniikan sovellusalue on laaja, on se alana hyvin nuori muuhun teolliseen tuotantoon verrattuna: teollisiksi tuotteiksi luokiteltavia ohjelmistoja on tuotettu vasta muutama vuosikymmen. Alan teknologian nopeasta kehityksestä johtuu, että ohjelmistojen koko on kaksinkertaistunut muutamien vuosien välein. Suurimmat ohjelmistokokonaisuudet on toteutettu avaruusteknologian alueella, ja ne ovat kooltaan kymmenien miljoonien ohjelmarivien suuruisia.

Kirjassa Ohjelmistotuotanto annetaan esimerkki ohjelmistokokonaisuuden monimutkaisuudesta vertaamalla sitä salapoliisiromaaniin, jossa on noin 10000 lausetta: *”Miljoonan rivin ohjelmistokokonaisuus vastaisi siis 100-osaista romaania, jossa henkilöt, johtolangat, murhaajat ja muut yksityiskohdat ovat keskenään ristiriidattomia. Kirjasarjaa käytettäisiin kymmenkunta vuotta. Tänä aikana lukijat vaatisivat siihen muutoksia: kirjan virheitä korjataan, murhaajaksi vaihdetaankin hovimestarin sijaan herra X, sivu poistetaan ja korvataan kokonaisella uudella luvulla jne. Mahdollisesti kokonaisia osia lisättäisiin tai korvat-*

taisiin uusilla. Muutosten myötä juonen johdonmukaisuuden säilyttäminen tulee lopulta mahdottomaksi ja järjestelmän täydellinen uusiminen tulee kannattavammaksi kuin vanhan ylläpito." Tämänkaltainen toiminta on jokapäiväistä ohjelmistotuotannossa; vain yksi kolmasosa alan töistä on uusien ohjelmistojen kehittämistä, loput kaksi kolmasosaa on vanhojen ohjelmien kehittämistä ja jatkokehitystä. Edellä mainituista syistä johtuen ei ole yllätys, että ohjelmistoista löytyy virheitä. Luvussa 3 kerrotaan, minkälaisia onnettomuuksia virheelliset ohjelmistot ovat aiheuttaneet. (Haikala, Märijärvi, 2004, s. 17–25.)

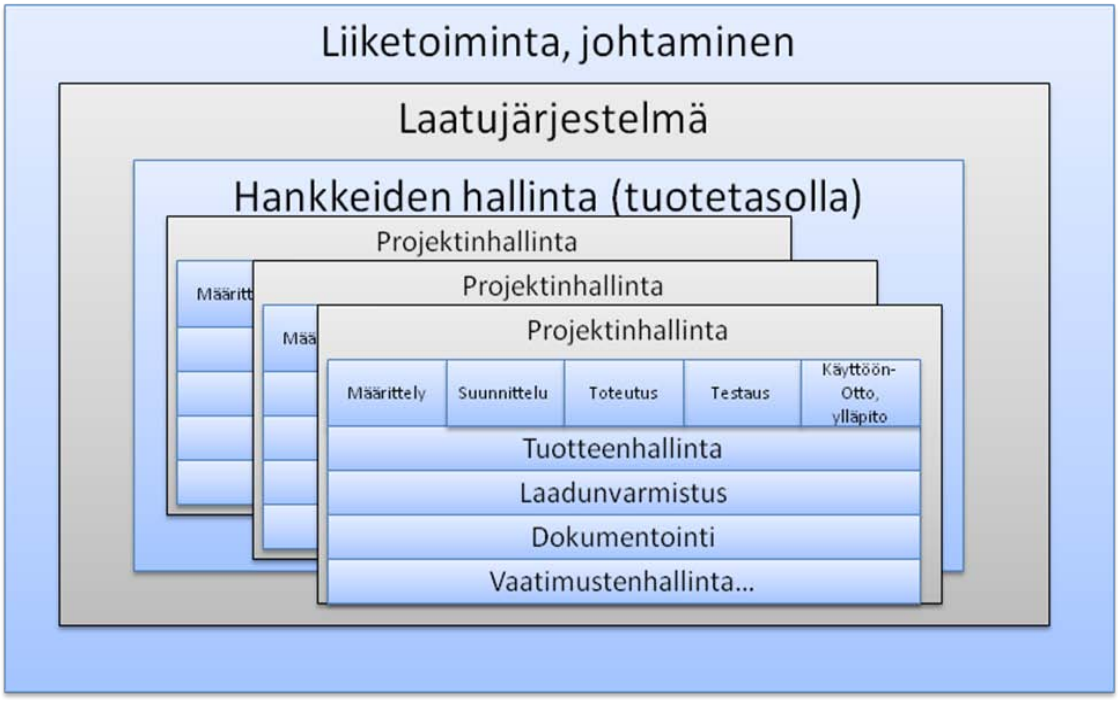
1.1 Ohjelmistotuotannon erityispiirteet

Ohjelmistoja ei voi toteuttaa rutiininomaisesti kuten talonrakennusta. Osaltaan tämä johtuu alan nuoruudesta, mutta on myös muita syitä, miksi ohjelmistotekniikka poikkeaa muusta tekniikasta. Ensimmäinen syy on monimutkaisuus: ohjelmistot on laadittu ratkomaan monimutkaisia ongelmia, joten on selvää, että ne ovat itsekin monimutkaisia. Toinen syy on työn näkymättömyys: on hyvin vaikea nähdä ohjelmistoprojektista, mikä sen valmiusaste todellisuudessa on. Muunnettavuus on myös yksi syy: ohjelmistoja on ”helppo” muuttaa, ja koska näin on, niitä myös muutetaan. Lisäksi erityistä on ohjelmistoprojektien ainutkertaisuus: talon on rakentanut joku ennenkin, mutta on mahdollista, ettei samankaltaista ohjelmistoa ole rakentanut kukaan, tai jos on, ei kukaan niistä, jotka nyt ohjelmistoa rakentavat. Ohjelmistotuotannossa ongelmia aiheuttaa edellä mainittujen lisäksi sen skaalautumattomuus: suuren projektin hallinta on hankalaa. Jos se ajautuu ongelmiin, on sen pelastaminen monesti toivotonta, eikä henkilökunnan lisääminen auta myöhässä olevan ohjelmistoprojektin ajan tasalle saattamista, päinvastoin. Viimeinen erityispiirre on ohjelmistojen epäjatkuvuus, joka tarkoittaa sitä, että ohjelmistoihin perustuvien järjestelmien käyttäytyminen virhetilanteessa on usein epäjatkuvaa. Vaikka silta ei yleensä romahda yhden pultin irrotessa, on aivan mahdollista, että yksi väärä bitti tekee tietojärjestelmästä käyttökelvottoman sekunnin murto-osassa. (Haikala ym. 2004, s. 28–31.)

Edellä luetellut piirteet ovat sellaisia, että ne johtuvat täysin käytettävän teknologian erityispiirteistä, eikä näihin ole odotettavissa muutosta tulevaisuudessakaan.

1.2 Ohjelmistotuotanto ja laatu

Ohjelmistotuotanto on monimutkainen ala, ja projektit usein ainutkertaisia. Ohjelmiston laadulla tarkoitetaan ohjelmistotuotteen kykyä täyttää käyttäjänsä toiveet ja odotukset, joten laatu on subjektiivinen, käyttäjästä ja ympäristöstä riippuva käsite. Laatua voidaan tarkastella sekä toiminnan, että tuotteen laadun kannalta, sillä tuotteen laadun lisäksi toiminnan laatu on tärkeä. Ajatellaan, että tuotteen laatuun vaikutetaan parhaiten toiminnan laadun kautta (kuva 1.1).



Kuva 1.1 Ohjelmistotuotannon osa-alueet (Haikala ym. 2004, s. 35)

Tuotteen tekoprosessia kuvaavaa toimintatapaa yrityksissä kutsutaan laatujärjestelmäksi. Sen tavoitteena on taata, että tuotantoprosessi tuottaa suunniteltua laatutasoa olevia tuotteita aikataulun ja budjetin mukaisesti. Laatujärjestelmää kuvataan laatukäsikirjalla ja muilla ohjeistuksilla. Laadunvarmistuksen tarkoitus on estää virheiden pääseminen tuotteeseen ja toisaalta löytää tehdyt virheet mahdollisimman aikaisessa vaiheessa projektia. Laadunvarmistusta tehdään testaamalla ja tarkastamalla. (Haikala ym. 2004, 48–51.)

Ohjelmistotuotannon erityispiirteistä johtuen laadunvarmistus on erityisen tärkeää, sillä huonolaatuinen ohjelmisto on yleensä käyttökelvoton. Tämä dokumentti

kertoo ohjelmistojen testauksesta, verifioinnista ja validoinnista. Dokumentin tarkoituksena on selvittää, minkälaisilla tekniikoilla voidaan vastata V&V:n kysymyksiin: rakennammeko me tuotetta oikein, ja rakennammeko me oikeaa tuotetta.

2 HISTORIA

Tässä luvussa kerrotaan ohjelmistotestauksen historiasta sekä siitä, mitä testaus on ja miksi se on tärkeää.

2.1 Testauksen historia

Automaattisen tietojenkäsittelyn historian alkuaikoina testaus keskittyi ohjelmistojen sijasta laitteistoihin. Tietokoneohjelmat olivat pääosin huolellisesti suunniteltuja, lyhyitä algoritmeja, mutta laitteistot olivat vain prototyyppejä. Ohjelmistoissa epäilemättä oli omat ongelmansa, mutta ne liittyivät tiiviisti laitteistojen luotettavuuteen. Ensimmäinen tietokoneohjelman tarkastamista käsittelevä artikkeli ilmestyi Alan Turingilta vuonna 1949, ja se käsitteli tarkastamista vakuutena ohjelman oikeasta toiminnasta. Seuraava, Turingilta vuonna 1950 ilmestynyt artikkeli käsitteli ohjelmiston toimivuuden testausta näkökulmasta, joka kysyi, täyttääkö ohjelmisto sille määritellyt vaatimukset. Tätä artikkelia voidaan pitää ensimmäisenä, joka käsittelee ohjelmiston testausta. Alkuaikoina testauksella tarkoitettiin virheiden etsintää ja korjausta (debug), ja sen suoritti yleensä ohjelmoija itse. Testaus suoritettiin valmiille ohjelmistolle ohjelmistoprojektin lopussa, ja sen tarkoitus oli vain varmistaa, että kaikki toimii niin kuin pitikin. Tätä aikakautta voidaan kuvailla virheenjäljityksen aikakaudeksi (debugging-oriented period) (kuva 2.1)

Vuosi	Aikakausi
– 1956	The debugging-oriented period Virheenjäljityksen aikakausi
1957–1978	The demostration-oriented period Havainnollistamisen aikakausi
1979–1982	The destruction-oriented period Hajottamisen aikakausi
1983–1987	The evaluation-oriented period Arvioinnin aikakausi
1988–	The prevention-oriented period, Ennaltaehkäisyn aikakausi

Kuva 2.1 Ohjelmistotestauksen kehitysvaiheet

1950-luvulla testauksen tavoite oli vain varmistaa, että ohjelman pystyi ajamaan, ja että se ratkaisi ongelman, jonka ratkaisemiseen se oli tarkoitettu. Tietokoneohjelmat monimutkaistuivat kuitenkin koko ajan, eikä tämänkaltainen varmistaminen enää ollut riittävää, vaan ohjelmien testaamista oli pakko alkaa tehdä suunnitellusti. Myös termeinä virheiden etsintä ja korjaus verrattuna testaamiseen erotettiin toisistaan, jolloin ensin mainittu piti huolta siitä että ohjelma toimi ja toiseksi mainittu siitä, että se vastasi vaatimuksia, jotka sille oli annettu. Tämä oli testauksessa havainnollistamisen aikakausi (demonstration-oriented period).

1970- ja 1980-lukujen taitteessa ohjelmistotestauksessa oli vallalla hajottamisen aikakausi (destruction-oriented period), jolloin ajateltiin, että testauksen tarkoitus on löytää mahdollisimman paljon virheitä testattavasta ohjelmistosta. Vuosia 1983–1987 pidetään arviointisuuntautuneena kautena (evaluation-oriented period), sillä tällöin alettiin testata, täyttääkö se määrittelynsä vaatimat ominaisuudet. Katselmoinneista ja tarkastuksista tuli osa testauskäytäntöjä, joilla tätä tutkittiin. Oli opittu, että mitä myöhäisemmässä vaiheessa virhe löytyy, sitä enemmän se maksaa. Vuodesta 1988 katsotaan alkavan ennaltaehkäisyn aikakausi

(prevention-oriented period), joka jatkuu edelleen. Nykyaikainen ohjelmiston testaus pyrkii havainnollistamaan, että ohjelmisto on määrittelyjen mukainen, sekä löytämään että ennaltaehkäisemään vikoja ja virhetilanteita aloittamalla testauksen suunnittelu jo ohjelmistoprojektin alkuvaiheissa. Voidaan todeta, että kehittyessään ohjelmistotestaus on säilyttänyt edellisten aikakausien painotukset ja tuonut niiden lisäksi uusia näkökulmia testaukseen.(Gelperin, Hezel, 1988.)

2.2 Ohjelmistotestauksen ammattimaistuminen

Ohjelmistotestaus on ammattimaistunut 1970-luvulta alkaen. Ensimmäinen ohjelmistotestausta käsittelevä konferenssi järjestettiin Pohjois-Carolinan yliopistossa Chapel Hillsissä vuonna 1972, ja ensimmäinen aihetta käsittelevä kirja, G.J. Myersin the Art of Software Testing, julkaistiin vuonna 1979, perustuen edellä mainitun konferenssin sisältöön. Kyseisestä kirjasta tehtiin sittemmin uusi versio vuonna 2004 (Myers, Sandler, Badgett 2004). Ensimmäistä ohjelmistotestausta käsittelevää ANSI/IEEE-standardia alettiin valmistella myös vuonna 1979, ja tuloksena oli vuonna 1983 julkaistu ”829-1983 IEEE Standard For Software Test Documentation”, joka määritteli kahdeksan dokumenttia sisältöineen ja muotoineen. Sittemmin standardista on tullut kaksi uutta versiota, vuosina 1998 ja 2008 (IEEE 2008,[Std.829-2008]). Myös yksikkötestaukselle on julkaistu standardi vuonna 1987, ”1008-1987 IEEE Standard For Software Unit Testing”. (Gelperin ym. 1988.)

Nykyään testaus onkin tunnustettu erikoisosaamisen alana ohjelmistotuotannossa ja mittava määrä aihetta käsitteleviä artikkeleita ja kirjoja on kirjoitettu ja erilaisia konferensseja ja seminaareja pidetty alkuaikojen jälkeen. Ohjelmistotestaus on vakiintunut osaksi ohjelmistotekniikan opetusta tekniikan alan oppilaitoksissa, ja yhdistyksiä on perustettu. Suomessa 1979 perustettu systeemityön yhdistys, Sytyke ry, joka on toiminut rekisteröityneenä yhdistyksenä vuodesta 1987 alkaen, sisältää testauksen osaamisyhteisön FAST eli Finnish Association of Software Testing (TestausOSY-FAST 2009).

Testaajille on pyritty luomaan myös sertifiointijärjestelmä, jotta saataisiin nostettua testaajien ammattitaitoa ja sen arvostusta sekä helpotettaisiin sen mittaamista kansainvälisesti. Vuonna 2002 Edinburghissa perustettiin International Software Testing Qualifications Board, ISTQB, ja sen tehtävänä oli kehittää kansainvälinen testaajien sertifiointijärjestelmä (Tietotekniikan liitto 2007). Vuonna 2006 ISTQB: n Foundation Level exam-sertifikaatti yhdistettiin ISEB:n Information Systems Examinations Board:n sertifikaatin kanssa (ISEB 2009a). ISEB on British Computer Society:n tietojenkäsittelyn tutkintolautakunta, ja sillä on kolmiportainen sertifiointijärjestelmä. Foundation-tason lisäksi järjestelmään kuuluu ISEB Intermediate Certificate in Software Testing sekä ISEB Practition Certificate, jossa on kaksi suuntautumisvaihtoehtoa, Test Analysis sekä Test Management (ISEB 2009b; ISEB 2009c).

3 NÄKÖKULMIA TESTAUKSEEN

Tässä luvussa kerrotaan, mikä on testauksen tarkoitus ja käydään läpi muutamia testauksen määritelmiä. Lisäksi kerrotaan eri koulukuntien näkemyksistä testauksen suhteen sekä siitä, minkälaisia seurauksia voi olla huonosti testatuilla ohjelmistoilla.

3.1 Testauksen tarkoitus

Kuten historiaa käsittelevässä luvussa todettiin, testauksen lähtökohta on ohjelmistotuotannon alkuaikoina ollut varmistaa ohjelmiston toimivuus. Nykyaikaisen testauksen lähtökohta on päinvastainen (Watkins. 2001, s. 9): testaamalla on tarkoitus paljastaa ohjelmiston toimimattomuus ja löytää niin paljon virheitä kuin mahdollista. Testauksella varmistetaan, että ohjelmisto vastaa määrittelyään ja että se toimii riittävän hyvin täyttäen sille asetetut laatuvaatimukset. Lisäksi halutaan varmistaa, ettei ohjelmisto tee mitään, mitä sen ei oleteta tekevän. Testaamalla voidaan selvittää myös se, kuinka paljon ohjelmistoa voi rasittaa, ennen kuin se alkaa toimia virheellisesti tai lopettaa toimintansa, sekä saada selville riskit, joita ohjelmiston julkaiseminen aiheuttaa sen tuleville käyttäjille.

3.2 Virheet

Kesäkuussa 1996 tapahtui kallein ohjelmointivirheen aiheuttama vahinko: Kouroussa sijaitsevasta Guayanan avaruuskeskuksesta lähetettiin neitsytmatkalleen Ariane 5 – kantoraketti, joka jouduttiin räjäyttämään maan ilmakehässä noin 40 sekuntia laukaisunsa jälkeen, koska kantoraketti ei ollut enää ohjattavissa. Ohjauksen pettäminen johtui siitä, että raketin edellisestä mallista, Ariane 4:sta, oli otettu komponentteja uudelleenkäyttöön huomioimatta sitä, että uusi raketti oli tehokkaampi kuin vanha. Raketin korkeutta ja nousukulmaa mittaava moduuli sai syötteekseen suuremman luvun kuin mitä se pystyi käsittelemään, ja tämä aiheutti muistivuodon, joka kaatoi järjestelmän ohjauksen. Myös varajärjestelmä kaatui, ja rakettia ei ohjannut enää mikään. Vikaraportissa (Lions, 1996) todetaan että:

"The failure of the Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence (30 seconds after lift- off). This loss of information was due to specification and design errors in the software of the inertial reference system.

The extensive reviews and tests carried out during the Ariane 5 Development Programme did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the potential failure."

Vaillinaisesti suoritettu testaus maksoi tässä tapauksessa 7 000 000 000 dollaria kantoraketin kehityskustannuksina sekä 500 000 000 dollaria itse raketin ja sen lastina olleiden satelliittien osalta. Jos tätä verrataan esimerkiksi Suomen valtion vuoden 2009 budjettiin, joka on 46 000 000 000 euroa, nähdään, että kyse on melkoisesta rahasummasta.

Myös ihmishenkiä on menetetty ohjelmointivirheiden takia: vuonna 1991 Persianlahden sodassa amerikkalaisten itse laukaisema Patriot Scud-ohjus aiheutti 28 oman sotilaan kuoleman ja sadan haavoittumisen osumalla amerikkalaisten parakkiin. Ohjausvirhe johtui siitä, että ajan laskemisessa käytettyjä lukuja pyöristettiin katkaisemalla luku, ja virhe kertautui ajan kuluessa. Onnettomuushet-

kellä virhe oli jo 0,35 sekuntia ja ohjuksen nopeus 1600 m/s. Tapaus oli ironinen siksi, että jos samanlaista pyöristystapaa olisi käytetty joka kohdassa ohjelmaa, ei vääränlainen ajanlaskenta olisi aiheuttanut ongelmia. (Blair, Obenski, Pridickas, 1992.)

Toinen tunnettu, kuolemantapauksia aiheuttanut ohjelmointivirhe oli Therac-25 säteilyhoitokone, joka antoi joissain tilanteissa potilaalle valtavan yliannostuksen säteilyä. Järjestelmä oli huonosti dokumentoitu ja huonosti testattu, eikä koneen toiminnasta saadusta palautteesta ei pidetty kirjaa, ja kun ongelmista raportoitiin, ei asiaa tutkittu riittävästi. Syynä onnettomuuksiin voi nähdä olemattomat laadunhallintaprosessit ja sokea luotto siihen, että ohjelmisto on testattu eikä se voi ajastaan rikkoutua. Virheellisesti toimiva kone tappoi kolme ja vammautti pysyvästi kolme ihmistä. (Levenson, Turner, 1993.)

Näistä esimerkeistä voi nähdä, että huonosti testatuilla ohjelmistoilla voi olla kohtalokkaat seuraukset sekä taloudellisesta että inhimillisestä näkökulmasta tarkasteltuina.

3.3 Testauksen määritelmät

Testausta on määritelty eri tavoin riippuen siitä, mikä aikakausi ja minkä koulukunnan edustaja määrittelyn on tehnyt.

Bill Hetzelin kirjasta The Complete Guide to Software Testing löytyy seuraava määritelmä, joka on kirjattu vuodelle 1973:

"Testing is the process of establishing confidence that a program or system does what it is supposed to do" – testaus on prosessi, jolla varmistetaan, että ohjelma tai järjestelmä tekee, mitä sen on tarkoitus tehdä. (Hetzel, 1988,s. 4.)

Edellä mainittu määritelmä on havainnollistamisen kauden ajattelun mukainen määritelmä, jonka mukaan testaamalla vain varmistetaan kaiken toimivan oikein. Tämänkaltaista testausta kutsutaan positiiviseksi testaukseksi.

Glenford J. Myersin (2004, 6) mukaan testauksen määritelmä oli:

"Testing is the process of executing a program with the intent of finding errors."
- Testaus on prosessi, jossa ohjelmaa ajetaan tarkoituksena löytää virheitä.

Tämä määritelmä löytyi jo Myersin kirjan vuoden 1979 painoksesta ja kertoo siitä, että hajottamisen aikakausi oli alkanut. Se ei ota kantaa siihen, onko ohjelmisto vaatimusten mukainen. Lähestymistapaa, jossa varmistetaan että ohjelmistossa on virheitä, kutsutaan negatiiviseksi testaukseksi. Käytännössä nämä määritelmät yhdistyvät, sillä testaamalla on tarkoitus varmistaa se, että ohjelmiston vaatimukset täyttyvät ja ohjelmistossa olevat virheet löytyvät, eli testaus on yleensä samalla sekä negatiivista että positiivista.

Bill Hetzelin kirjasta The Complete Guide to Software Testing löytyy myös seuraava määritelmä, joka on kirjattu vuodelle 1983:

"Testing is any activity aimed at evaluating an attribute or capability of a program or a system and determining that it meets its required results"- Testaus on mitä tahansa toimintaa, joka suoritetaan ohjelman tai ohjelmiston ominaisuuksien arvioimiseksi siten, että varmistetaan sen tuottavan vaaditut tulokset (Hetzel, 1998, s. 6).

Edellä mainittu määritelmä on arvioinnin aikakauden mukainen ajatus, että testaus on laadunhallintaa, jolla tutkitaan täyttääkö ohjelmistotuote määrittelynsä vaatimat ominaisuudet.

Seuraava määritelmä kiinnittää huomionsa siihen, että nykyaikaisessa ohjelmistotuotannossa suuri osa ohjelmistoprojekteja epäonnistuu täysin tai ainakin osittain, ja että huonolaatuisen ohjelmiston päästäminen markkinoille aiheuttaa ikävyyksiä sen loppukäyttäjille ja siitä johtuen huonoa mainosta ja tappiota ohjelmiston laatineelle yritykselle. Tämä määritelmä on James Bachin (1999) laatima:

"Testing is the process by which we explore and understand the status of the benefits and the risk associated with release of a software system"- Testaus on prosessi, jonka avulla pyritään tutkimaan ja ymmärtämään ne edut ja riskit, jotka ohjelmiston julkaiseminen aiheuttaa (Watkins, 2009, s. 9).

Tämä ennaltaehkäisevän testauksen mukainen määritelmä kertoo, että testaus on laadunvarmistuksen lisäksi myös riskienhallintaa.

3.4 Testauksen koulukunnat

Ohjelmistotestauksen asema suhteellisen nuorena erityisalanaan aiheuttaa sen, että vakiintuneita käytäntöjä on vähän. Alalla on asiantuntijoita, jotka näkevät testauksen hyvin eri tavoin, ja näitä mielipide-eroavuuksia selventääkseen on yritetty jaotella testaajat eri koulukuntiin. Koulukuntajaottelun on toivottu selventävän sitä, miksi eri asiantuntijat ovat asioista eri mieltä ja parantavan näin väittelyjen tasoa. Koulukunnat eivät ole vielä vakiintuneita, ja aiemmin on puhuttu neljästä koulukunnasta, nykypäivänä usein viidestäkin. Pettichord (2007) jaotteli vuonna 2003 Floridassa, testauksen työpajatapahtumassa ”Workshop on teaching software testing”, koulukunnat analyyttiseen, standardilähtöiseen, laatulähtöiseen, kontekstiohjattuun ja ketterän testauksen koulukuntiin (kuva 3.1).

Koulukunta	Periaate
Analytic school	Testaus on ohjelmistotiedettä
Standard school	Testaus on hallittu prosessi
Quality school	Testaus on laadun varmistamista
Context-driven school	Testaus on ohjelmistokehityksen yksi haara
Agile school	Testaus todistaa, että tuote on valmis

Kuva 3.1 Testauksen koulukunnat (Pettichord)

Cem Kaner taas toteaa blogikirjoituksessaan joulukuussa 2006, ettei vielä ole valmis julkaisemaan omaa näkemystään koulukunnista, koska kirja, jossa sen haluaisi julkaista, ei ole vielä valmis:

“At my time of writing, I think the best breakdown of the schools is:

- ***Factory school**: emphasis on reduction of testing tasks to routines that can be automated or delegated to cheap labor.*
- ***Control school**: emphasis on standards and processes that enforce or rely heavily on standards.*
- ***Test-driven school**: emphasis on code-focused testing by programmers.*
- ***Analytical school**: emphasis on analytical methods for assessing the quality of the software, including improvement of testability by improved precision of specifications and many types of modeling.*
- ***Context-drive school**: emphasis on adapting to the circumstances under which the product is developed and used.”*

Lainauksesta voidaan todeta, että tehdaskoulukukunta on synonyymi Pettichordin standardikoulukunnalle. Sen lisäksi mukaan on otettu testauslähtöinen koulukunta sekä kontrollikoulukunta, joka perustuu vahvasti varsinaisiin standardeihin, mutta on todennäköisesti nimetty tuolla tavalla sekaannusten välttämiseksi standardikoulukunnan ja kontrollikoulukunnan välillä. Koulukuntien määrittelyt elävät edelleen ja muitakin jaotteluja löytynee. Seuraavissa luvuissa esitellään Pettichordin mukaisten koulukuntien periaatteet.

3.4.1 Analyyttinen koulukunta

Analyyttisen koulukunnan ydinajatuksena on, että ohjelmisto on looginen kokonaisuus ja testaus matematiikan sivuhaara. Testaustekniikoilla pitää analyyttisen koulukunnan mukaan olla loogis-matemaattinen muoto ja testauksen tekninen suoritus. Koodin kattavuuden tutkiminen on analyyttisten testaajien mukaan objektiivinen tapa mitata testausta. Erilaisia koodikattavuusmittareita on lukuisia, muun muassa lausekattavuus, ehtokattavuus, haarakattavuus ja päätöskattavuus, näistä enemmän luvussa 4. Analyyttisten testaajien mielestä ohjelmiston tulee olla tarkasti määritelty ennen kuin sitä voidaan testata. Testauksella todennetaan, että ohjelmisto toimii määrittelyn mukaisesti. Koulukunta on vallalla yliopistomaailmassa ja yrityksissä, joissa tarvitaan korkeaa luotettavuutta ja turvallisuustasoa.

3.4.2 Standardilähtöinen koulukunta

Standardikoulukunta, jota myös tehdaskoulukunnaksi kutsutaan, näkee, että testaus on sarja työtehtäviä, joilla mitataan, missä vaiheessa ohjelmistoprojekti on. Testauksen tulee olla helposti hallittavaa. Tulosten pitää olla ennustettavissa ja testien hyvin suunniteltuja ja vaivattomasti toistettavissa. Standardikoulukunta kiinnittää huomionsa testauksen aiheuttamiin kustannuksiin ja pyrkii alentamaan niitä testauksen automaatiolla ja rutinoimalla testausta niin pitkälle että halpa, ammattitaidoton työvoimakin pystyy suorittamaan testit annetuilla ohjeilla. Tälle koulukunnalle ominainen testauksen apuväline on vaatimusten jäljitysmatriisit (requirements traceability matrix), eli taulukot, joilla pyritään jäljittämään jokainen vaatimus ja tutkimaan, onko vaatimukset testattu. Systeemejä voi olla erilaisia ja eritasoisia, mutta esimerkkinä annettakoon taulukot, joista ensimmäiseen (kuva 3.2) on listattu käyttötapaukset ja käyttötapausten sisältämät vaatimukset, ja merkitty, mitkä käyttötapaukset liittyvät kuhunkin vaatimukseen.

Käyttötapaus Vaatimus	UseCase 1	UseCase 2	UseCase 3	UseCase 4	UseCase 5	UseCase 6
Requirem. 1			x	x		
Requirem. 1.1					x	
Requirem. 1.2						x
Requirem. 2	x					
Requirem. 2.1				x		
Requirem. 3						
Requirem. 4						x

Kuva 3.2 Käyttötapaus-vaatimusmatriisi

Yhteen vaatimukseen voi liittyä useita käyttötapauksia. Taulukoiden otsikot voivat toimia linkkeinä toisiin kaavioihin tai vaatimusmäärittelydokumentteihin. Seuraavaan taulukkoon (kuva 3.3) listataan käyttötapaukset ja testitapaukset. Edelliseltä taulukkotasolta pystytään jäljittämään, mistä käyttötapauksiin kuuluvat vaatimukset löytyvät. Testaus-käyttötapausmatriisi täytetään niin, että saadaan jokaisen käyttötapauksen ja komponentin yhdistelmälle ainakin yksi testitapaus. Tällä pyritään varmistamaan, että tuote on testattu kattavasti (Myöhänen, 2002).

Testitapaus Käyttö- tapaus	TestCase 1	TestCase 2	TestCase 3	TestCase 4	TestCase 5	TestCase 6	TestCase 7
UseCase 1		x	x			x	
UseCase 2		x		x			
UseCase 3	x	x					
UseCase 4	x		x		x		
UseCase 5	x			x			x
UseCase 5				x			x

Kuva 3.3 Testaus-käyttötapausmatriisi

Tehdaskoulukunta vaatii, että testaus, kuten muutkin ohjelmistotuotannon aktiviteetit, ovat selkeärajaisesti erotettu toisistaan, ja että toiminnoilla on tarkat aloitus- ja lopetuskriteerit. Koska koulukunnan työtapaan kuuluu mittava dokumentointi jäljitysmahdollisuuksineen, on koulukunta muutosvastainen, koska suunnitelmiin tulevat muutokset vaikuttavat suureen määrään dokumentteja ja rutiineja. Tehdaskoulukunta kannustaa sertifioinnin, standardoinnin ja parhaiden käytäntöjen käyttöön. Koulukunta on yleinen yritysten ja julkishallinnon tietojärjestelmäprojekteissa.

3.4.3 Laatulähtöinen koulukunta

Laatulähtöinen koulukunta ajattelee, että ohjelmistotuotannon laadun varmistamiseksi tarvitaan kurinalaisuutta. Testaus arvioi, noudatetaanko sovittua kehitysprosessia, mikä se sitten onkaan. Testaajien tehtävä on vahtia, että kehittäjät noudattavat sääntöjä, ja suojella loppukäyttäjiä huonolaatuiselta ohjelmistolta. Laatulähtöisen testauksen periaatteena on että testausyksikkö toimii portinvartijana ja että ohjelmisto on valmis vasta sitten, kun laadunvarmistus näin sanoo. Laatulähtöisen koulukunnan mielestä testaus on ohjelmistokehityksen suhteen erillinen prosessi, joka nähdään osana laadunvarmistusta. Koulukunta on yleinen suurissa, byrokraattisissa organisaatioissa.

3.4.4 Kontekstiohjattu koulukunta

Kontekstiohjatun eli tilannelähtöisen koulukunnan mukaan ohjelmisto on ihmisen luoma, ja ihmiset luovat tilanteen. Testauksen tarkoituksena on löytää virheitä, ja virhe voi olla mitä tahansa ohjelmiston vääränlaista toimintaa, joka tuottaa harmia loppukäyttäjälle. Testaamalla pyritään oppimaan uusia asioita testattavasta aineistosta ja testausta pidetään vaativana, älyllisenä toimintona. Tämä koulukunta pitää ohjelmistoprojektiin matkan varrella tulevia muutoksia hyväksyttävinä ja odotettavina, ja pyrkii vastaamaan muutoksiin mukauttamalla testaussuunnitelmaa tarpeen mukaan. Seuraavaksi listataan koulukunnan seitsemän perusperiaatetta (Kaner, Bach, Pettichord, 2002, s. 261).

1. Minkä tahansa käytännön arvo riippuu sen kontekstista.
2. On olemassa hyviä käytäntöjä omassa kontekstissaan, mutta ei ”parhaita käytäntöjä”.
3. Yhteistyötä tekevät ihmiset ovat tärkein osa minkä tahansa projektin viitekehystä.
4. Projektit muuttuvat usein tavoilla, joita ei voi ennakoida.
5. Tuote on ratkaisu. Jos ongelmaa ei ole ratkaistu, tuote ei toimi.
6. Hyvä ohjelmistotestaus on haastava älyllinen prosessi.
7. Tehokas ja oikea-aikainen ohjelmiston testaus edellyttää testaajalta arvostelukykyä ja ammattitaitoa läpi koko projektin.

Koulukunnalle ominainen tapa testata on tutkiva testaus (exploratory testing, ET), jossa testataan tuotetta ilman ennakkosuunnitelmaa. Tutkivan testauksen määritelmä:

"Exploratory testing is simultaneous learning, test design, and test execution." – Tutkiva testaus on samanaikaista oppimista, testaussuunnittelua ja testien suorittamista.

Kontekstiohjattu koulukunta on nopean ja käytännöllisen testaustoiminnan kannalla, ja arvostaa enemmän testaajan kykyjä löytää virheitä kuin sitä, miten hän osaa seurata jotain ennalta määrättyjä käytäntöjä. Koulukunta on yleinen kaupallisessa ohjelmistotuotannossa.

3.4.5 Ketterän testauksen koulukunta

Ketterän testauksen koulukunta nojautuu nimensä mukaisesti ketteriin menetelmiin. Ketterässä kehityksessä ohjelmistoa ei määritellä etukäteen tarkasti, vaan selvitetään, miten sen pääpiirteittäin tulisi toimia. Kun tämä on selvitetty, jaetaan kokonaisuus iteraatioihin, joissa ohjelmistoa aletaan kehittää, ja järjestetään iteraatiot prioriteettijärjestykseen. Iteraatioista voidaan kirjoittaa käyttäjätarinoita. Käyttäjätarinat ovat lyhyitä, selväsanaisia tekstejä, joissa on jokaisessa yksi selkeä toiminnallinen vaatimus järjestelmälle. Tekstistä käy ilmi, kuka on toimija, joka pystyy tekemään toiminnon, ja minkälainen toiminto on ja miksi se pitää tehdä. Käyttäjätarinasta kirjoitetaan testisarja, jotka valmiin iteraation tulee läpäistä. Ketterässä kehityksessä pyritään toistamaan jatkuvaa määrittele-suunnittele-toteuta-testaa-julkaise-sykliä. Läpäisty testisarja kertoo että iteraatio on valmis julkaistavaksi. Ketterissä menetelmissä suositaan testien automatisointia, koska iteraatiot koostetaan kokonaiseksi ohjelmistoksi usein, parhaimmillaan joka päivä, ja kooste on kätevä testata joka kerta sille kirjoitetulla, automaattisesti suoritettavalla regressiotestisarjalla. Yksi erikseen mainittava ketterä testausmenetelmä on Test Driven Design, tai Test Driven Development, TDD, jossa testit kirjoitetaan ennen ohjelmakoodia (Ambler, 2009). Testauslähtöisessä kehityksessä etukäteen kirjoitetut testitapaukset voivat korvata varsinaisen vaatimusmäärittelyn, sillä testitapaukset kertovat yksiselitteisesti, kuinka ohjelman pitäisi toimia.

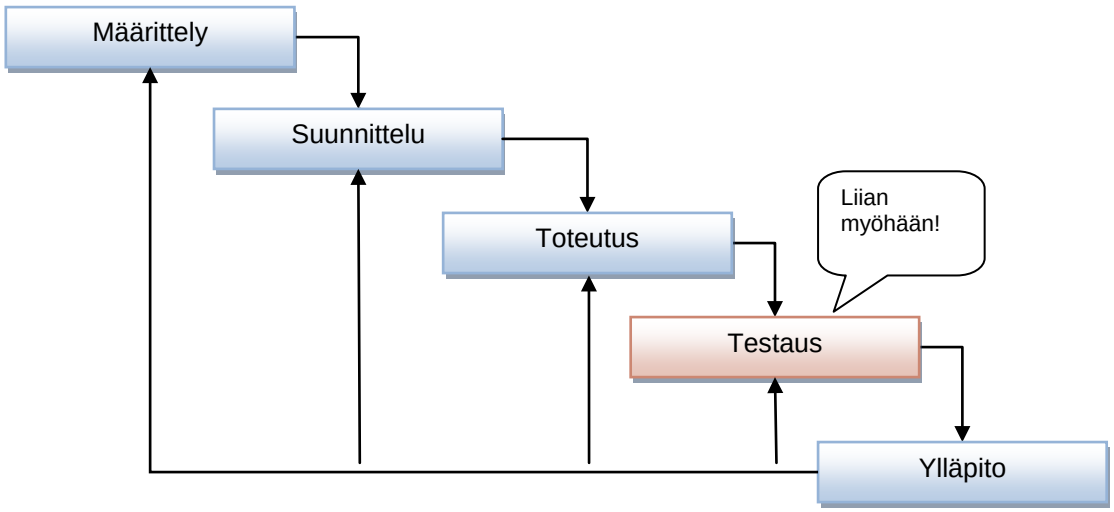
Ketterän testauksen koulukunta on yleinen kaupallisessa ohjelmistotuotannossa.

4 TESTAUS OSANA OHJELMISTOTUOTANTOA

Tässä luvussa selvitetään testauksen sijoittumista ohjelmistoprojektissa, määritellään virheet ja niiden vakavuusluokittelu sekä selitetään testauksen eri vaiheet V-mallin mukaisesti.

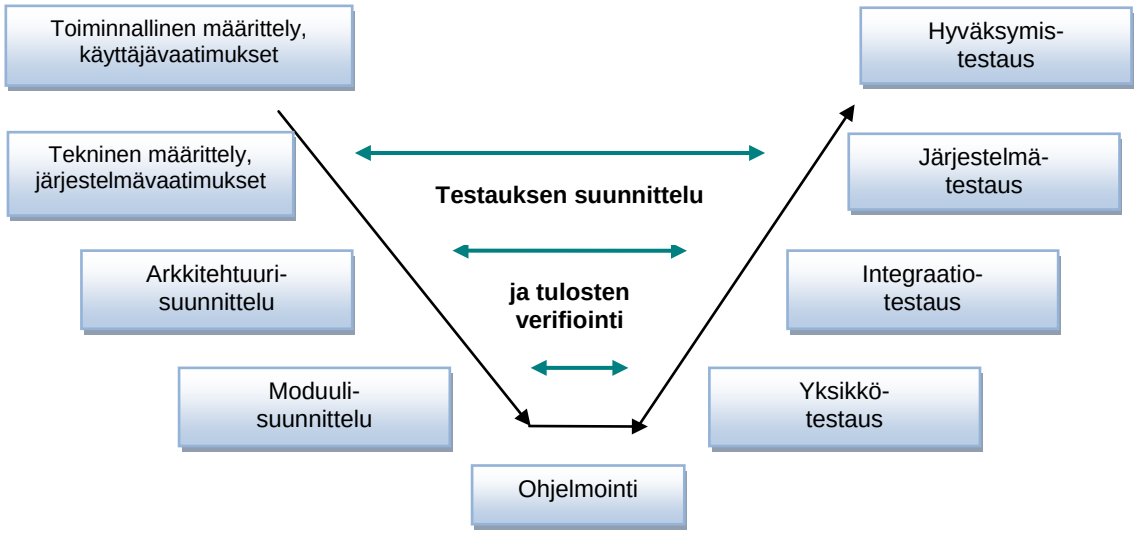
4.1 Vesiputousmalli ja V-malli

Perinteisesti testaus on nähty ohjelmistotuotannossa yhtenä vesiputousmallin osavaiheena. Vesiputousmallin (kuva 4.1) vaiheet ovat määrittely, eli ohjelmiston vaatimusten selvittäminen, suunnittelu, eli arkkitehtuurin ja ohjelmiston yksityiskohtien suunnittelu, toteutus, jolla on tarkoitettu varsinaisen ohjelmakoodin kirjoittamisen ohella yksikkötestausta ja toteutuksen verifiointia sekä viimeisenä testaus, joka on järjestelmätestausta sekä toteutuksen validointia. Tätä mallia käytetään edelleen pienissä projekteissa. Kun testaus on prosessin viimeinen vaihe, se jää usein liian vähälle huomiolle. Esimerkiksi projektin myöhästyessä karsitaan määrällisesti eniten testaukseen käytettävästä ajasta. Tämä vaikuttaa huonontavasti ohjelmiston laatuun. Jos projektin loppuvaiheessa aloitettu testaus paljastaa suunnitteluvirheistä tai puutteellisesta vaatimusmäärittelystä aiheutuneita vikoja, on näiden virheiden korjaus kallista ja aikaa vievää. Joissain tapauksissa virheiden korjaaminen on mahdotonta.



Kuva 4.1Ohjelmistotuotannon vesiputousmalli

Myöhäisessä vaiheessa aloitetun testauksen ongelmia korjaamaan on kehitetty testauksen V-malli (kuva 4.2). Tässä mallissa ohjelmiston testaamiseen aletaan ottaa kantaa jo projektin alkuvaiheissa. Mallissa testaus on integroitu prosessin kaikkiin vaiheisiin, jolloin jokaisen ohjelmistotuotantoprosessin työvaiheen yhteydessä suunnitellaan testipaketti. Näiden testipakettien sisältönä ovat hyväksymistestauksen yhteydessä käyttäjävaatimukset, järjestelmätestauksen yhteydessä järjestelmävaatimukset, integrointitestauksessa arkkitehtuurisuunnitelma sekä yksikkötestauksessa moduulisuunnitelma.



Kuva 4.2 Testauksen V-malli

V-malli on vesiputousmalliin nähden parempi siinä, että testauksen suunnittelu aloitetaan heti projektin alussa ja itse testauskin heti, kun testattavaa materiaalia kertyy. Tästä syystä projektin myöhästyessä siitä ei karsita enempää kuin muistakaan projektin vaiheista. Ohjelmiston laatu paranee, kun testaus valmistellaan huolellisesti ja tarkastetaan kunkin testipaketin vaatima vaihe. V-mallin mukaisilla tarkastuksilla karsiutuu myös suurin osa suunnittelu- ja määrittelyvirheistä eivätkä ne kulkeudu toteutukseen asti. Myös vaikeimmat validointivirheet saadaan karsittua minimiin tarkastuksilla.

4.2 Virheiden luokittelu

Ohjelmistotuotannossa ovat erilaiset virheet ja vikatilanteet on nimetty tarkasti, jotta projektien parissa työskentelevät ihmiset ymmärtäisivät, minkälaisesta virhetilanteesta milloinkin on kyse. Vian ja virheen määritelmiä käytetään joissain lähteissä siten, että virheellä (error, mistake, bug) tarkoitetaan poikkeamaa spesifikaatiosta, vialla (fault) virheellisen ohjelmakohdan suoritusta ja häiriöllä (failure) järjestelmän häiriötä, jonka vika aiheuttaa. Seuraavaksi esitellään virhetilanteiden määritelmät ISTQB:n testaussanaston mukaisesti. (ISTQB, 2007.)

Virhe (error, mistake) tarkoittaa *”ihmisen toimintaa, joka tuottaa väärän tuloksen.”* Virhe voi johtua väärästä toimintatavasta, virheellisestä ajattelusta tai epäonnistuneesta kommunikaatiosta.

Vian (fault, defect, ”bug”) määritelmä testaussanastossa on *” komponentissa tai järjestelmässä oleva virhe, joka voi aiheuttaa sen, että komponentti tai järjestelmä ei pysty suorittamaan siltä edellytettävää toimintoa; esim. virheellinen lauseke tai muuttujan määrittely.*” Vika voi tarkoittaa määrittelystä poikkeavaa toimintaa tai toimintaa, jota ei ole kirjattu määrittelyihin, mutta joka kuitenkin aiheuttaa virheellistä toimintaa ohjelmistossa.

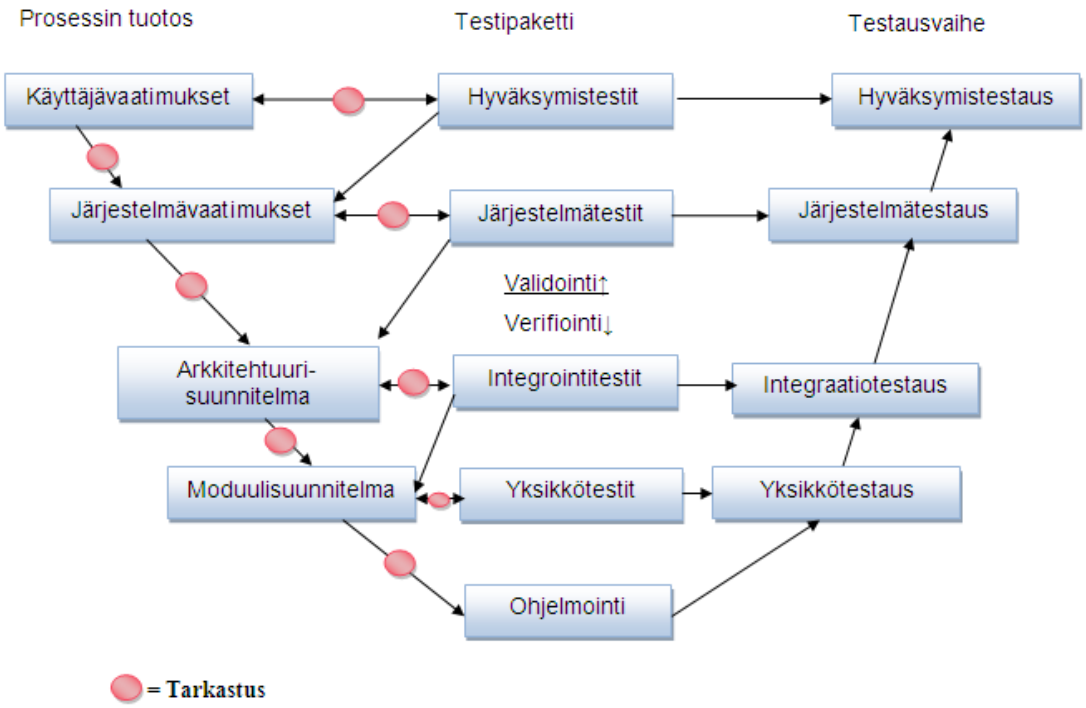
Häiriö eli vikaantuminen (failure) määritellään *”ohjelmiston poikkeamaksi odotetusta toimituksesta, palvelusta tai tuloksesta.”* Tällaisia poikkeamia voi olla se, että ohjelmisto tekee vaaditut asiat, mutta ei niin nopeasti kuin on vaadittu, tai että jotain odottamatonta tapahtuu, kuten laskennan väärä lopputulos, ohjelman jääminen silmukkaan tai sen kaatuminen.

Virheelle voidaan määritellä prioriteetti ja vakavuus. Virheen prioriteetti ilmaisee sen, kuinka nopeasti ohjelmistoprojektista vastuussa olevat henkilöt haluavat saada sen korjattua. Virheen prioriteetti muuttuu usein projektin edetessä, ja yleensä vakavilla ongelmilla on korkea korjausprioriteetti ja pienillä kosmeettisilla ongelmilla matala. Tämä ei kuitenkaan aina pidä paikkaansa, sillä jos kosmeettinen ongelma on esimerkiksi sellainen, että yrityksen logo on www-sivulla väärin päin, sen korjausprioriteetti on arvattavasti korkea. Virheen vakavuudes-

ta puhuttaessa tarkoitetaan sen vaikutusta tai seurauksia, ja se ei yleensä muutu, ellei projektin edetessä opita sen piiloseurauksista jotain uutta. Virheen vakavuudesta päättää tyypillisesti testaaja, mutta prioriteetti on projektitason tai yritystason päätös. (Kaner ym. 2002, s. 75.)

4.3 Verifiointi ja validointi

Testaussanasto (ISTQB, 2007) määrittelee verifioinnin näin: *”Määrättyjen vaatimusten täyttymisen vahvistaminen kokeellisesti ja objektiivisen todistusaineiston avulla.”* Validoinnin määritelmä on: *”Määrättyä käyttöä varten tai sovellukselle asetettujen vaatimusten täyttymisen vahvistaminen kokeellisesti ja objektiivisen todistusaineiston avulla.”* Verifioinnissa varmistetaan että ohjelmisto on määrittelynsä mukainen, eli se vastaa kysymykseen ”Rakennammeko me ohjelmistoa oikein?” Validoimalla taas varmistetaan, että täyttääkö ohjelmisto asiakkaan asettamat odotukset, ja kysymys, johon validointi vastaa, kuuluu ”Rakennammeko me oikeanlaista ohjelmistoa?” Ohjelmistotuotannossa nämä yhdistyvät termiin V&V (verification & validation). V&V:n tavoitteena on, että tarkastamalla ja testaamalla ohjelmistoa koko prosessin elinkaaren ajan säännöllisesti (kuva 4.3) varmistutaan siitä, että ohjelmisto täyttää sille asetetut tarpeet, että se on riittävän virheetön omassa kontekstissaan. (Watkins, 2001, s. 10.)



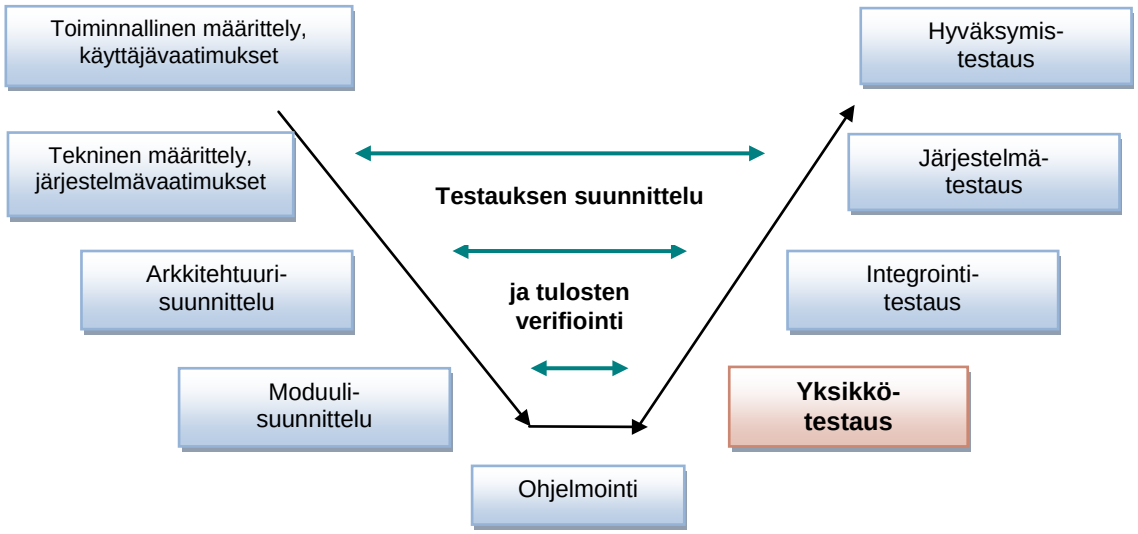
Kuva 4.3 Verifiointi ja validointi (Taina, 2004)

V&V-tekniikat jakautuvat testaukseen ja katselmointiin. Testaus on ohjelmiston dynaamista analysointia eri tekniikoin. Tarkastuksilla tarkoitetaan staattista ohjelmiston dokumentaation tai ohjelmakoodin läpilukemista, jolla pyritään löytämään virheitä. Katselmointikäytäntöjä on olemassa useita, niistä enemmän luvussa 5.

4.4 Yksikkötestaus

Yksikkötestauksella (unit testing) tarkoitetaan yksittäisten ohjelmistokomponenttien testausta (ISTQB, 2007). Sen synonyymeja ovat moduulitestaus (module testing) ja komponenttitestaus (component testing). Testattavan yksikön määritelmä vaihtelee sen mukaan, minkälaiset menetelmät ja minkälainen ohjelmistoprojekti on kyseessä (Watkins, 2001, s. 46). Proseduraalisessa ohjelmoinnissa yksikkö voi olla yksi funktio tai proseduuri tai toisiinsa tiukasti liittyvien funktioiden tai proseduurien ryhmä. Olio-ohjelmoinnissa yksiköllä voidaan tarkoittaa luokkaa, oliota tai yksittäisen metodin toteuttamaa funktiota. Visuaalisessa ohjelmoinnissa yksikkö voi olla ikkuna tai ikkunassa sijaitseva toisiinsa liittyvien

komponenttien ryhmä, kuten valintalista. Komponenttipohjaisessa kehitysympäristössä yksiköksi voidaan kutsua uudelleenkäytettävää komponenttia, josta on tiedettävä sen alkuperä ja testaushistoria, jotta testaaja voi päättää, paljonko komponenttia vielä pitää testata. Yksikkötestauksen suorittaa usein ohjelmoija itse omalle koodilleen, ja se sijaitsee testauksen V-mallissa alimmalla tasolla, moduulien suunnittelun vastinparina ja lähinnä ohjelmakoodin kirjoittamista (kuva 4.4).



Kuva 4.4 Yksikkötestauksen sijoittuminen V-mallissa

Yksikkötestaus on testauksen vaihe, jota pyritään usein automatisoimaan. Automatisoidun yksikkötestauksen etu on, että testit voi suorittaa kuka tahansa ja ne voidaan ajaa aina, kun on tarpeen, esimerkiksi tehtäessä muutoksia testattavaan yksikköön tai integroitaessa sitä kokonaisuuteen (Kaner ym. 2002, s. 35). Huolellinen yksikkötestaus on tärkeää. Jos yksikkötestauksessa on puutteita, se kostautuu testauksen myöhemmissä vaiheissa, kun joudutaan tekemään aikaa vievää ja monimutkaista vianetsintää suuresta määrästä yhdistettyjä komponentteja.

Yksikkötestaukseen on kehitetty erilaisia työkaluja, joilla voi luoda tarvittavan testausympäristön yksikkötestien suorittamiseksi. Näistä voidaan mainita xUnit-työkalut, joilla testit kirjoitetaan koodiksi, joka ajaa testattavia skriptejä ja sen

jälkeen tallentaa, tarkistaa ja esittää tulokset muodossa, jota testin tekijä voi tarkastella.

4.4.1 Testipeti, ajuri ja tynkä

Testipedillä tai testikehyksellä tarkoitetaan ohjelmistoa tai laitetta, jonka avulla voidaan korvata puuttuvaa laitteistoa tai ohjelmistoa jonkin osakokonaisuuden testaamisessa. Testipeti mahdollistaa ajureiden tai tynkämoduulien tekemisen, joiden avulla luodaan testausympäristö, jossa voidaan suorittaa yksikkötestejä. Testipetiä tynkineen ja ajureineen voidaan käyttää myös integraatiotestauksessa.

Testiajuri (driver) on ohjelmistokomponentti tai testaustyökalu, joka korvaa komponentin, jolla kontrolloidaan tai kutsutaan testattavaa komponenttia tai järjestelmää.

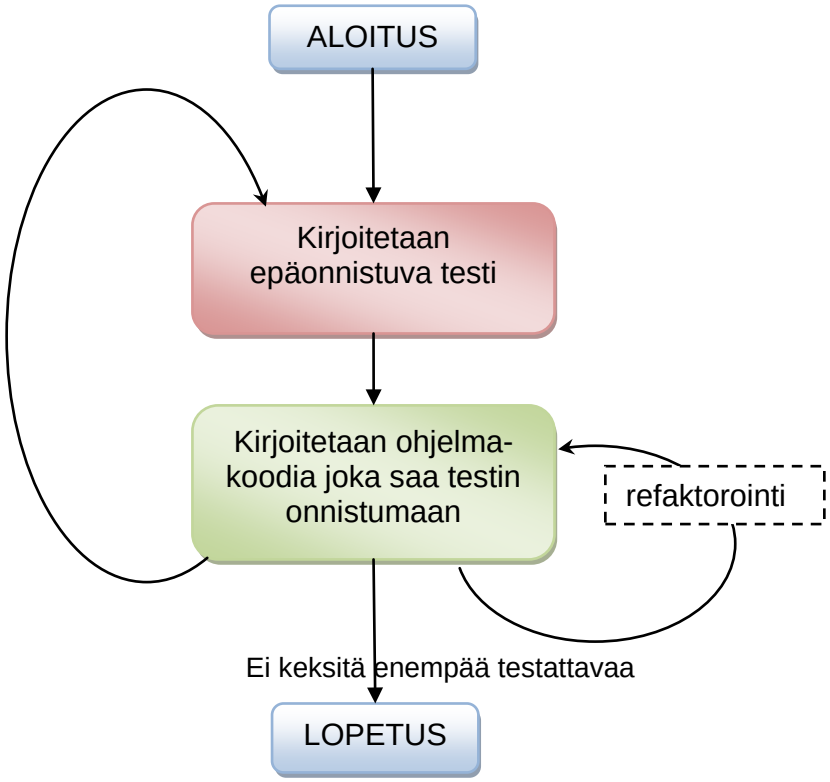
Tyngäksi (stub) kutsutaan ohjelmistokomponentin erikseen tehtyä toteutusta, jota käytetään kehitettäessä tai testattaessa komponenttia, joka kutsuu sitä tai on muuten riippuvainen siitä. Se korvaa kutsutun komponentin. (ISTQB, 2007.)

4.4.2 Test Driven Development

Testiohjattu kehitys eli TDD (test driven development) on ohjelmistokehityksen työtapa, jossa testitapaukset suunnitellaan, kirjoitetaan ja usein automatisoidaan ennen kuin aletaan kirjoittaa varsinaista ohjelmakoodia, jota testitapauksilla testataan (ISTQB, 2007). Kirjoitettavat testit toimivat formaalina määrittelynä toiminnallisuudelle, joka on tarkoitus toteuttaa, ja läpi menevät testit varmistavat, että toteutus on määrittelyn mukainen. TDD:tä käytettäessä projektiin muodostuu mittava joukko yksikkötestejä, joka voidaan automatisoida. Tätä testijoukkoa jatkuvasti ajettaessa löytyvät ohjelmakoodin virheet välittömästi ja ne ovat helposti paikallistettavissa ja korjattavissa. Näitä testejä voi käyttää hyväkseen regressiotestauksessa, ja TDD:tä käytetään ketterissä ohjelmistokehitysmenetelmissä kuten XP:ssä (eXtreme Programming), joissa regressiotestejä ajetaan jatkuvasti ohjelmistoa koostettaessa päivittäin.

TDD:tä toteutetaan syklillä, joka XP:ssä on kuvattu termein red-green-refactor (kuva 4.5), johtuen siitä, että testityökalujen testauksen etenemistä osoittava palkki on punainen tai vihreä sen mukaan, läpäiseekö koodi testitapauksen (Shore, 2005):

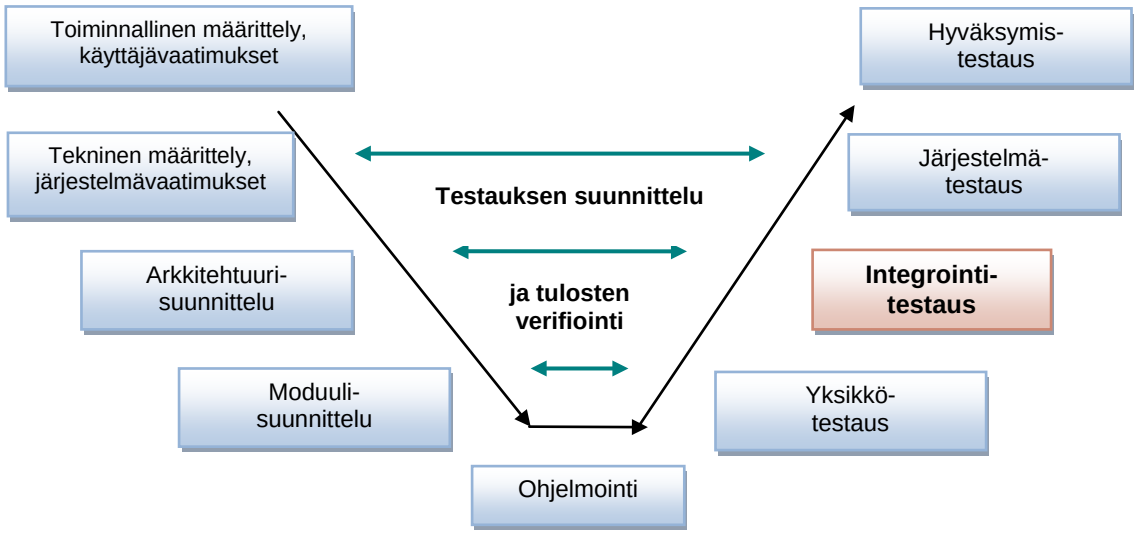
1. Think: Mietitään ja kirjoitetaan lista mahdollisista testeistä, joiden onnistunut läpäisy osoittaisi koodin olevan valmis.
2. Red: Kirjoitetaan lyhyesti testitapaus, yleensä alle viisi riviä koodia. Ajetaan testijoukko ja todetaan uuden testin epäonnistuvan ja testipalkin muuttuvan punaiseksi. Testi ei mene läpi, koska sen testaamaa toiminnallisuutta ei ole vielä kirjoitettu, ja testi toimii vain määrittelynä tulevalle toiminnallisuudelle ja rajapinnalle.
3. Green: Kirjoitetaan pieni määrä tuotantokoodia, ei yleensä yli viittä riviä. Koodin ei tarvitse olla tiivistettyä tai hyvin suunniteltua, sillä tarkoitus on koodata nopeasti sen verran, että testi menee läpi ja testipalkki muuttuu vihreäksi. Koodi refaktoroidaan kun se on läpäissyt testin.
4. Refactor: Siivotaan koodista päällekkäisyydet ynnä muu sellainen, ja ajetaan testit uudestaan.
5. Repeat: Sykliä toistetaan jatkuvasti lisätessä uusia testejä ja uutta ohjelmakoodia. Syklien tulisi olla nopeita, alle kymmenenminuuttisia, sillä pitempi sykli kertoo siitä, että ohjelmaa kehitetään liian isoissa osissa.



Kuva 4.5 Test Driven Development

4. 5 Integrointitestaus

Integrointitestaus (integration testing) on testauksen vaihe, jossa ohjelmiston osakokonaisuudet, yksiköt eli komponentit, yhdistetään toisiinsa ja testataan, toimivatko ne yhdistettyinä ohjelmiston toiminnallisten vaatimusten mukaisesti. Tässä vaiheessa on tarkoitus löytää yhteensopimattomuudet moduulien välillä, ei enää viallisia moduuleja. Integrointitestauksen testitapausten pitäisi demonstroida sitä, onko yhdistettyjen komponenttien rajapintojen vuorovaikutus sellaista, kuin sen pitäisi olla. Integrointitestejä suunnitellessa pitää huolehtia, etteivät ne sisällä päällekkäisyyksiä yksikkötestien kanssa, sillä se olisi vain resurssien ja ajan hukkaamista. Integrointitestaus sijoittuu testauksen V-mallissa toisiksi alimmalle testaustasolle järjestelmän arkkitehtuurisuunnittelun vastinpariksi (Kuva 4.6).



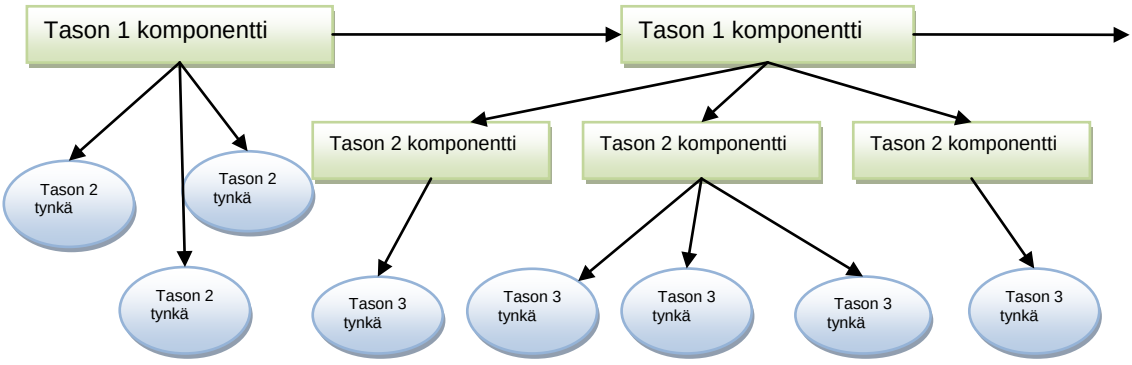
Kuva 4.6 Integrointitestauksen sijoittuminen V-mallissa

Komponenttien integrointitestausta ei tule sekoittaa järjestelmäintegrointitestaukseen, joka kohdistuu rajapintoihin, jotka ovat testattavan ohjelmiston ja siihen integroitavien, ulkopuolisten ohjelmistojen tai organisaatioiden välillä. (Watkins, 2001, s. 53.)

Komponentteja voidaan integroida eri järjestyksissä. Yksi, ainakin pienissä ohjelmistoprojekteissa yleinen integrointimenetelmä, on ns. Big Bang -testaus, jossa ohjelmiston tai laitteiston tai molempien elementit yhdistetään kaikki kerralla johonkin komponenttiin tai toisiinsa, sen sijaan, että se tehtäisiin vaiheittain (ISTQB, 2007). Tämä menetelmä aiheuttaa sen, että integrointi sijoittuu kokonaisuudessaan ohjelmistoprojektin loppuun, jolloin aikaa ei yleensä ole enää korjauksiin. Suuren kokonaisuuden koostaminen kerralla tekee myös vianetsinnästä vaikeaa. Suositeltavampia integrointitapoja ovat vaiheittaiset menetelmät, esimerkkeinä jäsentävä, kokoava ja jatkuva integrointi. Näissä uudet komponentit lisätään vähitellen, ja uusia komponentteja lisättäessä ajetaan regressiotestit, joilla testataan, ettei lisäys ole aiheuttanut vikoja aiemmin lisättyihin komponentteihin. Regressiotestit ovat usein automatisoituja, jolloin ne voidaan ajaa aina, kun testattava ohjelmisto koostetaan.

4.5.1 Jäsentävä integrointi

Jäsentävä integrointi (Top-down testing) tarkoittaa vaiheittaista testaustapaa, jossa testataan ensimmäiseksi ylimmän tason komponentit, korvaten alemman tason komponentit tyngillä. Ohjelmiston komponenttihierarkian haarat käydään läpi taso kerrallaan (kuva 4.7). Testattuja ylemmän tason komponentteja käytetään testattaessa alempia tasoja. Prosessia jatketaan, kunnes kaikki komponentit on testattu. (ISTQB, 2007.)



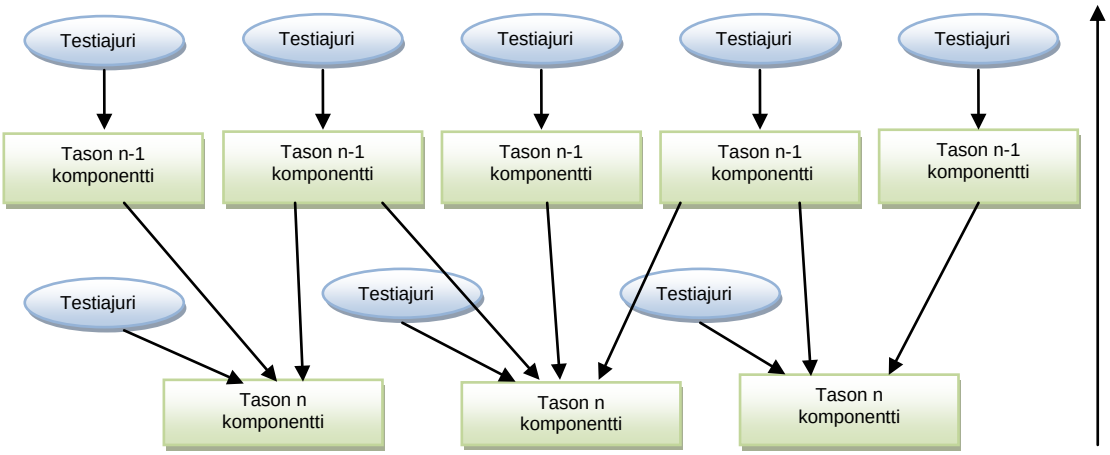
Kuva 4.7 Jäsentävä integrointi

Menetelmän etuna on se, että tynkien avulla saadaan aikaiseksi jo varhaisessa kehitysvaiheessa toimiva ohjelman osa, joka ottaa vastaan oikeita syötteitä ja josta tulee oikeita tulosteita, tynkien simuloidessa asioita, jotka tulevat myöhemmässä kehitysvaiheessa tapahtumaan ohjelman sisällä. Se auttaa löytämään käyttäjävirheitä ja ongelmia ohjelman hallintalogiikassa ja tarjoaa mahdollisuuden esitellä ohjelmaa havainnollisesti sen tuleville käyttäjille. Lisäksi menetelmä tarjoaa mahdollisuuden todentaa, että kaikki ohjelmiston vaaditut ominaisuudet ovat kehitteillä. Se toimii myös moraalisena kannustimena ollessaan konkreettinen todiste ohjelmiston rakentumisesta. (Myers ym, 2004, s. 114.) Menetelmää kannattaa suosia, jos testattavan ohjelmiston kriittisimmät virhemahdollisuudet sijaitsevat sen moduulihierarkian ylätasolla.

Menetelmän haittapuoli on se, että tynkien laatiminen vie aikaa, ja monimutkaisista komponenteista on usein vaikea suunnitella ja toteuttaa sellaista tynkää, joka kuvaisi oikean komponentin toimintaa riittävän täydellisesti, varsinkin kun tyngän simuloima tulostus voi olla ohjelmoinnillisesti hyvin etäällä syötteestä, jonka valmis, ylemmän tason moduuli tarjoaa. Ongelmia syntyy myös silloin, jos uusien komponenttien lisäyksen yhteydessä tehtävä regressiotestaus on puutteellista.

4.5.2 Kokoava integrointi

Kokoava integrointi, (Bottom-up testing) on myös vaiheittainen integrointimenetelmä. Kokoavassa menetelmässä testataan ensimmäiseksi alimman tason komponentit, korvaten puuttuvat ylemmän tason komponentit testiajureilla (ISTQB, 2007). Kuvassa 4.8 havainnollistetaan tätä, siinä alimman tason komponentit on merkitty n-tasoksi. Valmiita, alemman tason komponentteja käyte-



Kuva 4.8 Kokoava integrointi

tään hyödyksi testattaessa ylemmän tason komponentteja. Prosessia jatketaan, kunnes ylin taso on saavutettu. Kokoava integrointi hoidetaan käytännössä yleensä niin, että rakennetaan kokonainen testiympäristö yksittäisten testiajurien sijaan. Menetelmää kannattaa suosia silloin, kun testattavan ohjelmiston kriittisimmät virhemahdollisuudet sijaitsevat moduulihierarkian alatasolla.

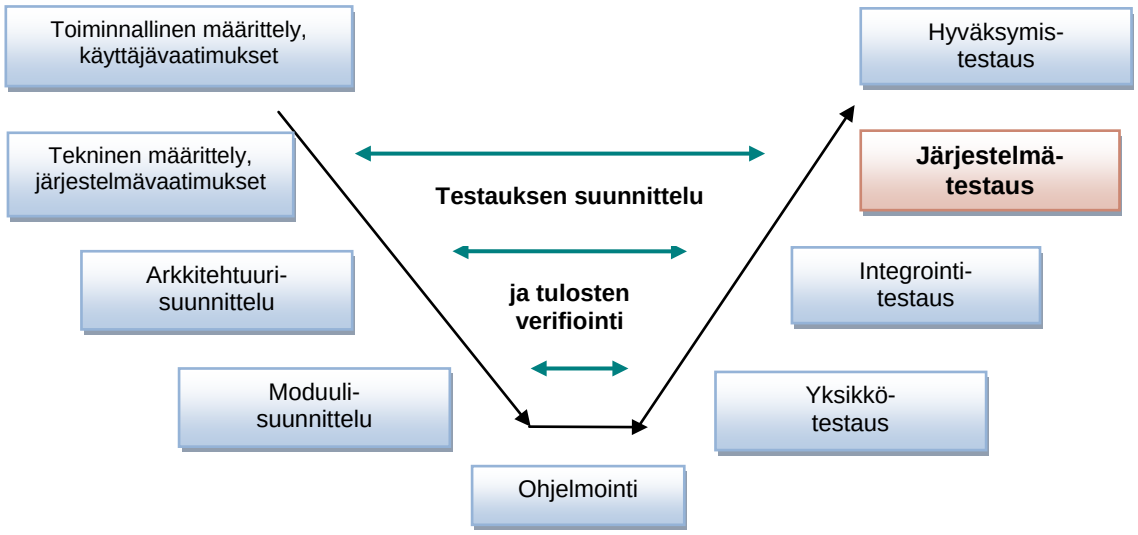
Vaikka menetelmä on päinvastainen jäsentävän integroinnin suhteen, ei valmista ohjelmaa voida simuloida ajurien avulla varhaisessa kehitysvaiheessa samalla tavalla kuten jäsentävässä integroinnissa tynkien. Voidaan ennemmin sanoa, että ohjelmaa ei ole, ennen kuin ylimmän komponentit on sijoitettu paikoilleen, mutta kun tämä on tehty, on tuloksena valmis, testattu ohjelma. Ajureiden kirjoittaminen on yleensä helpompaa kuin tynkien, sillä ne sijoittuvat välittömästi testattavien komponenttien yläpuolelle, eikä jäsentävässä integroinnissa ilmaantuvia syötteen ja tulostuksen etäisyyden ongelmia synny. (Myers ym, 2004, s. 117.)

4.5.3 Jatkuva integrointi

Jatkuva integrointi on yksi ketterän ohjelmistokehityksen menetelmistä. Ketterässä kehityksessä ohjelmistoa koostetaan ja integroidaan jatkuvasti, usein jopa kerran vuorokaudessa. Jatkuvan integroinnin suorittamiseen on olemassa erilaisia ohjelmia, jotka ajastetusti käynnistävät ohjelmiston koostamisen, ajavat testisarjat ja julkaisevat koostamisen lopputulokset. Jos kooste epäonnistuu, ohjelma ilmoittaa siitä. Jatkuvassa integroinnissa koostamisen yhteyteen automatisoiduissa testeissä voi olla sekä yksikkötestejä että erilaisia integrointitestejä, ”savutestejä”, jotka varmistavat kokonaisuuden toiminnan (Vuorinen, 2005).

4.6 Järjestelmätestaus

Testaussanasto määrittelee järjestelmätestauksen t. systeemitestauksen (system testing) testaukseksi, jolla varmistetaan, että integroitu järjestelmä täyttää sille asetetut vaatimukset (ISTQB 2007).
Lähteestä riippuen nämä vaatimukset vaihtelevat, mutta tässä raportissa järjestelmätestauksella tarkoitetaan, että integrointitestattua järjestelmää testataan sen järjestelmävaatimusten suhteen (kuva 4.9). Järjestelmätestaus erotetaan näin hyväksymistestauksesta, jossa järjestelmää testataan sen selvittämiseksi, toimiiko ohjelma käyttäjävaatimusten mukaisesti. Näiden kahden testausvaiheen raja ei ole selkeä, ja usein hyväksymistestaus yhdistyy järjestelmätestaukseen.



Kuva 4.9 Järjestelmätestauksen sijoittuminen V-mallissa

Tässä testausvaiheessa suurin osa toiminnallisista häiriöistä on jo löydetty, ja testaus keskittyy järjestelmän ei-toiminnallisiin vaatimuksiin. Järjestelmätestausvaiheessa havaitut virheet ovat usein kriittisiä, ja niiden korjaukset heijastuvat muihin sovelluksen osiin ja saattavat aiheuttaa uusia virheitä (Haikala, ym, 2004). Tästä syystä järjestelmätestauksessakin on harjoitettava riittävästi regressiotestausta. Kirjassa Art of Software Testing (Myers ym, 2004, 130) todetaan, että järjestelmätestaus on vaikein ja väärinymmärretyin testauksen osavaihe ja ettei sitä voida suorittaa, jos ei ole olemassa kirjattuja, mitattavissa olevia tavoitteita, joita vastaan testata ohjelmistoa. Järjestelmätestauksen osa-alueita ovat muun muassa toiminnallisuustestaus, volyymitestaus, kuormitustestaus, käytettävyystestaus, tietoturvatestaus, suorituskykytestaus, resurssien käytön testaus, kokoonpanon testaus, yhteensopivuustestaus, asennettavuustestaus, luotettavuustestaus, toipuvuustestaus, ylläpidettävyystestaus, dokumentoinnin testaus sekä prosessin testaus. Järjestelmätestauksen piiriin kuuluu ohjelmiston testaus myös liittyen sen käyttöympäristöön ja laitteistoihin. Se voi sisältää myös järjestelmäintegraatiotestausta, jos ohjelmistolla on rajapintoja, joilla se liittyy toisiin sovelluksiin.

4.6.1 Toiminnallisuustestaus

Toiminnallisuustestauksessa (Functionality testing, feature testing) tutkitaan, tuottaako ohjelmistotuote toiminnot, jotka täyttävät määrättyjen käyttöolosuhtei-

den edellyttämät tarpeet (ISTQB, 2007). Tämä testaus suoritetaan ns. mustalaatikkotekniikalla, eli luomalla testitapaukset esimerkiksi järjestelmän toiminnallisen määrittelyn tai käyttäjädokumentaation pohjalta, kiinnittämättä huomiota ohjelmiston sisäiseen rakenteeseen. Tällä testauksella tutkitaan, ovatko vaaditut toiminnallisuudet ylipäänsä olemassa ja toimivatko ne halutusti.

4.6.2 Volyymitestaus

Volyymitestaus eli määrätestaus (Volume testing) tarkoittaa testausta, jossa järjestelmä altistetaan mielikuvituksellisen suurelle määrälle tietoa ja tutkitaan, kuinka se selviytyy siitä. Tietoa voidaan lisätä mm. lisäämällä syötetietojen määrää, palvelupyyntöjen tiheyttä, samanaikaisia käyttäjiä tai käyttäjien samanaikaisten toimintojen määrää. Tämän testauksen tarkoituksena on tutkia, pystyykö ohjelmisto käsittelemään käyttäjävaatimusten määrittelemän määrän tietoa kerrallaan. (Myers ym, 2004, s. 133.)

4.6.3 Kuormitustestaus

Kuormitustestaus tai rasitustestaus (Stress testing) on suorituskykytestausta, jossa järjestelmää ylikuormitetaan mittavasti samalla, kun järjestelmän resursseja alimitoitetaan esimerkiksi vähentämällä keskusmuistia tai palvelimia (ISTQB, 2007). Vaikka kuormitustestauksen testitapausten tilanteet ovat sellaisia, ettei niitä tosielämässä todennäköisesti tule koskaan tapahtumaan, saadaan näin selville, minkälaisia virheitä järjestelmässä on odotettavissa samankaltaisissa, järjestelmää vähemmän kuormittavissa tilanteissa (Myers ym, 2004, s. 135). Järjestelmiä suunnitellessa pitäisi ottaa nämä tilanteet huomioon niin, että kun kuormitus on liian kova normaalin toiminnan jatkamiseksi, toimisi järjestelmä vieläkin järkevästi, eli antaisi virheilmoituksia, jättäisi osan palvelukäskyistä täyttämättä tai kaatuisi hallitusti, tietoja menettämättä.

4.6.4 Käytettävyystestaus

Käytettävyystestaus (Usability testing) testaa ohjelman ymmärrettävyyttä, omaksuttavuutta ja helppokäyttöisyyttä sitä käyttävän ihmisen näkökulmasta (ISTQB, 2007). Käytettävyystestauksessa on mietittävä muun muassa, onko käyttöliittymä räätälöity loppukäyttäjän älyllisen tason ja koulutustaustan mukaiseksi, ovatko ohjelmiston tulosteet kuten opastustekstit, varoitukset ym. tarkoituksenmukaisia ja oikeakielisiä ja etteivät ne ole epäasiallisia. On myös tutkittava, ovatko järjestelmän virheilmoitukset riittävän informatiivisia loppukäyttäjälleen. Käytettävyyden testaukseen kuuluu myös tutkia, onko käyttöliittymien logiikka samanlaista kaikkialla järjestelmässä, ovatko valikkorakenteet loogisia ja pääseekö järjestelmän aloitustasolle helposti takaisin, jos eksyy järjestelmässä navigoidessaan. Siihen kuuluu myös sen tutkiminen, onko virheettömyyttä vaativissa toimissa, kuten esimerkiksi verkkopankkiohjelmistossa, riittävästi päällekkäisiä tarkistuksia käyttäjän identiteetin varmistamiseksi, esimerkiksi kysytään sekä käyttäjätunnus, salasana että avainluku. Käytettävyysasioita on myös se, onko järjestelmässä päällekkäisiä tai ylimääräisiä toimintoja, joita ei edes tarvita, ja että käsitteleekö järjestelmä loogisesti ja havainnollisesti käyttäjän toimia. Viimeksi mainitulla tarkoitetaan esimerkiksi linkkien värjäytymistä jos niissä on käyty, valintalistasta poimitun tiedon valituksi tulemisen näkymistä käyttäjälle selkeästi, tai käyttäjän aloittaessa jonkin aikaa vievän toiminnon, järjestelmän antamaa ilmoitusta siitä, mitä on tapahtumassa ja kauanko se kestää. Käytettävyystestauksen tuloksena on selvitys siitä, vastaako järjestelmän käytettävyys sitä tasoa, joka sille on määritelty. (Myers ym, 2004, s. 135–137.)

4.6.5 Tietoturvatestaus

Tietoturvatestaus (Security testing) määritellään testaukseksi, jolla määritetään järjestelmän tietoturvan taso (ISTQB, 2007). Nykypäivän tietoyhteiskunnassa yksityisyyden suoja on kasvava huolenaihe, ja järjestelmille määritellään vaatimukset tietoturvan tasosta. Erityisesti web-sovelluksille, joita pääsee käyttämään Internetissä kuka tahansa, on syytä tehdä huolelliset tietoturvatestit, sillä käyttäjien menetettyä luottamusta on usein mahdotonta saada takaisin tietovuotojen jäljiltä. Tietoturvatestauksen pohjaksi voi ottaa selvää, minkälaisia tie-

toturvaongelmia muissa, samantyylisissä sovelluksissa on. Sen jälkeen voi luoda testejä, jotka kertovat, onko testattavassa järjestelmässä vastaavia ongelmia. Selvityksiä tunnetuista tietoturvaongelmista käyttöjärjestelmissä ja ohjelmistotuotteissa voi etsiä esimerkiksi erilaisilta Internetin keskustelufoorumeilta ja uutisryhmistä ja ohjelmistoalan lehdistä. Muun muassa viestintäviraston CERT-FI -tietoturvaviranomainen kokoaa tietoa tietoturvaloukkauksista ja niiden estämisestä. Palvelu löytyy osoitteesta http://www.cert.fi/index.html. Tietoturvauhkia ovat muun muassa:

Heikot varmenteet, kuten liian yksinkertaiset salasanat tai se, ettei tietoa salata esimerkiksi kryptaamalla salasanoja salausalgoritmeillä. Myös tietoliikenteen salauksen vaillinaisuus voi aiheuttaa tietoturvaongelmia.

Puskurin ylivuoto on yksi yleisimpiä tapoja käyttää hyväkseen tietoturvahaavoittuvuuksia. Muistinhallinnan heikkoudet koodissa mahdollistavat tämän. Ylivuototilanne aiheutetaan kasvattamalla tai vähentämällä puskurin indeksiä niin, ettei se enää osoita sille varatulle muistialueelle. Ylivuodon avulla voidaan muistiin sijoittaa vahingollista koodia, joka mahdollisesti ajetaan myöhemmin.

Evästeiden käyttäminen huolettomasti, esimerkiksi käyttäjäoikeuksien hallintaan. Se on vaarallista, sillä niiden varastaminen ja muokkaaminen on helppoa.

Erilaiset **injektiot**, kuten SQL-injektio, jossa web-sovelluksen input-kenttiin syötetään hakusanan tai muun oikean syötteen sijasta merkkijono, joka muuttaa SQL-hakulauseen merkityksen. Tällä tavoin voidaan esimerkiksi huijata sisäänkirjautumistoiminnoissa. (Microsoft, 2008.)

Ristiinskriptaus (cross-site scripting, XSS), jossa verkkosivuille voidaan tunkeutua huijaamalla käyttäjän ohjelma suorittamaan muualta peräisin olevaa koodia. XSS-haavoittuvuuksien avulla voidaan varastaa käyttäjien yksityisiä tietoja kuten pankkitunnuksia, luottokorttien numeroita, evästeitä yms. (Moisio, 2008.)

RFI, Remote File Inclusion, on tekniikka, jossa hyökätään php:llä koodatulle web-sivulle käyttämällä etätiedostoja, jotka otetaan käyttöön yksinkertaisesti editoimalla sivun osoitetta. Tekniikalla voidaan saada näkyviin kansiot ja tiedostot, joita sivustolla on. Myös roskapostia lähettäviä skriptejä voidaan näin syöttää web-sivuille. (Wikipedia, 2009.)

Tietoturvatestaus kertoo, täyttyvätkö järjestelmään kohdistuvat odotukset tietoturvallisuuden tasosta. (Myers ym, 2004.)

4.6.6 Suorituskykytestaus

Suorituskykytestauksella (Performance testing) testataan sitä, ovatko järjestelmän vasteajat ja läpimenoajat odotetulla tasolla. Testitapaukset vaihtelevat sen mukaan, minkälaisesta järjestelmästä on kyse. Testit voivat mitata esimerkiksi tietokantahakujen kestoa tai simuloida normaalia järjestelmän käyttöä ja tutkia, paljonko eri toimintojen suorittamiseen kuluu aikaa. Suorituskykytestaus kertoo, onko järjestelmä riittävän nopea käyttää.

4.6.7 Resurssien käytön testaus

Resurssien käytön testausta (Resource utilization testing), jota kirjassa Art of Software Testing kutsutaan tietovarastotestaukseksi (Storage testing), käytetään testaamaan, onko järjestelmän resurssien käyttö toivotulla tasolla. Resurssit, joita tässä yhteydessä tarkoitetaan, ovat esimerkiksi ohjelman käyttämä muistin ja levytilan määrä tai heittotiedostojen koko niissä olosuhteissa, joissa järjestelmää aiotaan käyttää (ISTQB, 2007).

4.6.8 Kokoonpanon testaus

Kokoonpanon testaus (Configuration testing), jota kutsutaan myös siirrettävyystestaukseksi (Portability testing), tutkii, miten järjestelmä käyttäytyy, jos se siirretään eri laite- tai ohjelmistoympäristöön (ISTQB, 2007). Selainpohjaiset sovellukset on syytä testata eri www-selaimilla ja mielellään niiden käytössä olevilla eri versioilla eri käyttöjärjestelmissä, sillä selaimet toimivat usein eri tavalla riip-

puen siitä, missä käyttöjärjestelmässä niitä ajetaan. Kokoonpanon testaus kertoo, kuinka järjestelmä toimii eri ympäristöissä.

4.6.9 Yhteensopivuustestaus

Yhteensopivuustestauksella (Compatibility testing, Interoperability testing), tutkitaan, kuinka hyvin järjestelmä toimii yhdessä muiden, siihen liittyvien järjestelmien tai komponenttien kanssa (ISTQB, 2007). Tämä liittyvä järjestelmä voi esimerkiksi olla jo käytössä oleva tietokantapalvelin, johon testattava järjestelmä ottaa yhteyttä, ja testaus voi koskea sitä, onko järjestelmän syötteet sellaisia, että ne kirjautuvat oikein tietokantaan. Yhteensopivuustestaus kertoo, onnistuuko järjestelmien liittäminen toisiinsa määritellysti (Myers ym, 2004,s. 138).

4.6.10 Asennettavuustestaus

Asennettavuustestauksen (Installability testing) tarkoituksena on testata, kuinka ohjelman asennus onnistuu. Asennettavuus on tärkeää, sillä se on usein käyttäjän ensimmäinen kokemus järjestelmästä, ja jos jo asennuksessa on ongelmia, käyttäjä saattaa luopua järjestelmän käytöstä ja etsiä jonkin muun tuotteen saman tien. Epävakaa asennettavuus murentaa käyttäjän luottamusta järjestelmän laatuun. (Myers, ym, 2007, s. 139.)

4.6.11 Luotettavuustestaus

Luotettavuustestaus (Reliability testing) testaa järjestelmän luotettavuutta toimintavarmuuden näkökulmasta, eli sitä, mikä on todennäköisyys sille, että järjestelmä toimii riittävän häiriöttömästi riittävän pitkään. Luotettavuus sisältää myös vikasietoisuuden, eli sen, etteivät järjestelmän vikatilanteet johda järjestelmävirheisiin, jotka aiheuttaisivat häiriöitä järjestelmän käytössä. Luotettavuuden mittaamiseen on olemassa laskennallisia mittareita, joilla voidaan ajonaikaisesta suorituksesta tutkia, kuinka luotettava järjestelmä on. Näitä mittasuureita on esimerkiksi POFOD (propability of failure on demand), pyynnön häiriötodennäköisyys, joka kertoo epäonnistumisien lukumäärän suoritetuista palveluista. Se ilmoitetaan suhdelukuna, esimerkiksi 1:1000 tarkoittaa todennäköi-

syyttä että yksi tuhannesta toiminnosta epäonnistuu (Paavilainen, 2004). Muita mittareita ovat esimerkiksi MTBF (mean time between failure) eli keskimääräinen aika häiriöiden välillä ja MTTR (mean time to recovery), keskimääräinen korjausaika. Järjestelmän saavutettavuutta (AVAIL, availability) voidaan laskea näiden suureiden avulla: AVAIL = MTBF / (MTBF+MTTR). Jos MTBF olisi esimerkiksi 1000 h, ja AVAIL:in arvoksi tulisi 0,998, tarkoittaisi se, että järjestelmä olisi pois käytöstä keskimäärin kaksi tuntia tuhannesta (Hecht, 2003, 15). Järjestelmän luotettavuus on sitä tärkeämpää, mitä suurempia vahinkoja tai häiriöitä aiheutuu järjestelmän käyttökatkoksista. Esimerkiksi tietoliikennejärjestelmien tulisi toimia ilman käyttökatkoksia tai ne ovat käyttökelvottomia.

4.6.12 Toipuvuustestaus

Toipuvuustestaus (recovery testing) selvittää, miten järjestelmä palautuu tai uudelleenkäynnistyy ohjelmointivirheiden, laitevikojen tai virheellisten syötteiden jäljiltä. Toipuvuustestausta voidaan tehdä vaikkapa poistamalla äkillisesti järjestelmän osia käytöstä erilaisilla odottamattomilla tavoilla kuten virrankatkaisulla. Toipuvuustestauksessa tutkitaan myös, onko järjestelmän MTTR (mean time to recovery) määrittelyjen rajoissa. Toipumisaika on tärkeä, koska käyttökatkoksien pitkittyminen aiheuttaa usein taloudellisia tappioita. (Myers ym, 2004, s. 141.)

4.6.13 Ylläpidettävyystestaus

Ylläpidettävyystestaus (Serviceability testing, Maintainability testing) selvittää, miten järjestelmä on muokattavissa korjattaessa virheitä, täytettäessä uusia vaatimuksia, tehtäessä toimenpiteitä ylläpidon helpottamiseksi tulevaisuudessa tai ympäristömuutoksiin vastattaessa (ISTQB, 2007). Testauksen kohteena on esimerkiksi järjestelmän virhediagnostiikka, huoltoprosessit ja järjestelmälogiikan dokumentoinnin laatu. (Myers ym, 2004, s. 142.)

4.6.14 Dokumentoinnin testaus

Dokumentoinnin testaus (Documentation testing) tutkii järjestelmän käyttö- ja asennusohjeita. Itse ohjeet tulee katselmoida ensiksi niiden tarkkuuden ja selkeyden varmistamiseksi. Tarkastettujen ja oikeanlaisiksi todettujen dokumenttien perusteella kirjoitetaan testitapaukset. Kaikki käyttötapaukset, joita dokumentaatiosta löytyy, tulee kirjoittaa testeiksi ja syöttää ohjelmaan. Näin saadaan selville, kohtaavatko järjestelmän toiminta ja sen käyttäjädokumentaatio toisensa. (Myers ym, 2004, s. 142.)

4.6.15 Prosessin testaus

Prosessin testaus (Procedure testing) tutkii, onko järjestelmä yhteensopiva käyttäjien olemassa olevien prosessien ja proseduurien kanssa (ISTQB, 2007). Kaikki järjestelmän käytön prosessit, myös sellaiset, jotka eivät ole täysin automatisoitavissa, tulisi olla dokumentoituina ja testattavissa tässä testausvaiheessa. Prosessi voi olla esimerkiksi tietokannan pääkäyttäjän suorittama tietokannan varmuuskopiointi ja sen palauttaminen varmuuskopion avulla. Prosessin testaus olisi hyvä teettää muulla henkilöllä kuin sillä, joka yleensä suorittaa testauksen kohteena olevan toiminnan. (Myers ym, 2004, s. 142.)

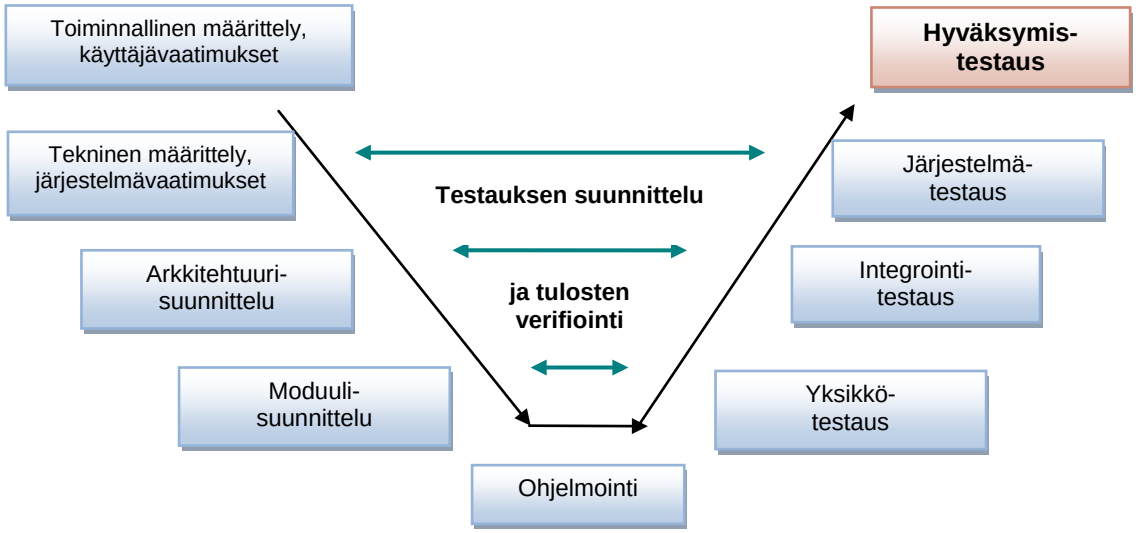
4.6.16 Järjestelmäintegrointitestaus

Järjestelmien ja ohjelmistojen integroinnin testaus kohdistuu rajapintoihin testattavan järjestelmän ja siihen liitoksissa olevien organisaatioiden tai järjestelmien välillä (ISTQB, 2009).

4.7 Hyväksymistestaus

Hyväksymistestaus on testausprosessin vaihe, jolla tutkitaan, täyttääkö testauksen kohteena oleva järjestelmä loppukäyttäjän tarpeet liiketoiminnallisessa mielessä ja onko järjestelmän toimivuus ja käytettävyys kelvolliset loppukäyttäjälle (Watkins, 2001, s. 73). Hyväksymistestauksessa voidaan hyväksyttää myös

käyttöohjeet ja muu dokumentaatio, jos sitä ei ole tehty jo järjestelmätestauksen yhteydessä. Hyväksymistestaus käy läpi samoja asioita kuin järjestelmätestauskin, ja usein hyväksymis- ja järjestelmätestaus on sulautettu yhdeksi testausvaiheeksi. Suurin ero järjestelmätestauksen ja hyväksymistestauksen välillä on se, että testauksen suorittaa asiakas. Hyväksymistestauksen menetelmiä ovat ohjelman normaalin toiminnan simuloiminen tai sen testaaminen pilottiprojektin puitteissa. Hyväksymistestaus sijoittuu testauksen v-mallissa toiminnallisen määrittelyn vastinpariksi (kuva 4.10). Hyväksymistestauksen vaiheita ovat alfatestaus ja betatestaus, tosin joissain lähteissä ko. vaiheet luetellaan suoritettaviksi hyväksymistestauksen jälkeen.



Kuva 4.10 Hyväksymistestauksen sijoittuminen V-mallissa

4.7.1 Alfatestaus

Alfatestaus (Alpha testing) on simuloitua tai todellista toiminnallista testausta, jonka suorittavat joko tuotteen loppukäyttäjät tai itsenäinen, riippumaton testausryhmä. Testaus suoritetaan kehittäjän tiloissa, mutta sen suorittaa kehittäjäorganisaation ulkopuolinen taho, joko asiakas itse tai asiakkaan alihankintana ostama testausryhmä. Alfatestausta voidaan käyttää valmisohjelmistojen sisäisenä hyväksymistestauksena.(ISTQB, 2007.)

4.7.2 Betatestaus

ISTQB:n sanaston mukaan (ISTQB,2007) betatestaus (Beta testing) tarkoittaa potentiaalisten tai jo olemassa olevien käyttäjien tai asiakkaiden suorittamaa toiminnallista testausta. Betatestausvaiheessa ohjelmisto on yleensä asennettu sen lopulliseen ympäristöön, ja tällä testauksella varmistetaan, että järjestelmä täyttää asiakkaan tarpeet ja toimii määriteltyjen liiketoimintaprosessien mukaan. Betatestausta voidaan käyttää osana valmisohjelmistojen ulkoista hyväksymistestausta, koska sillä saadaan palautetta markkinoilta. Julkinen betatestaus voidaan suorittaa antamalla ohjelmistotuotteen betaversio ilmaislevitykseen, jolloin ihmiset, jotka lataavat ohjelmiston itselleen, toimivat sen testausryhmänä. Tässä menetelmässä haittapuolena on se, että teollisuusvakoilun mahdollisuus kasvaa. Lisäksi ohjelmasta saatu palaute voi olla vähäistä suuresta testiryhmästä huolimatta. Julkisella betatestauksella voi myös menettää potentiaalisia asiakkaita, jos asiakaspalautteeseen ei vastata asiallisesti tai ohjelman korjaus havaittujen vikojen osalta on hidasta ja sattumanvaraista. Rajatussa betatestauksessa betaversiota tai sen käyttöoikeutta jaetaan valitulle, profiloidulle ryhmälle käyttäjiä. Tällöin teollisuusvakoilun mahdollisuus vähenee, ja palautetta on mahdollista saada enemmän jaettaessa betaversiota testattavaksi ihmisille, jotka oikeasti ovat kiinnostuneita testattavan palvelun käytettävyydestä ja sen parantamisesta.

4.8 Regressiotestaus

Testaussanasto (ISTQB, 2007) määrittelee regressiotestauksen (regression testing) aiemmin testatun ohjelman testaukseksi siihen tehtyjen muutosten jälkeen. Se varmistaa, että muutokset eivät itsessään ole tuottaneet tai paljastaneet piilossa olleita vikoja niillä ohjelmiston alueilla, jota ei ole muutettu. Ohjelman muutoksiksi lasketaan koodin muuttaminen, lisääminen tai poistaminen. Muita muutoksia, joiden jälkeen regressiotestaus suoritetaan, ovat ohjelman käyttöympäristön muutokset, kuten käyttöjärjestelmän tai laitteiston muutokset. Jos ohjelmaan sen sijaan lisätään uusi toiminnallisuus, puhutaan progressiotestauksesta.

Tyypillisesti regressiotestausta suoritetaan iteratiivisessa, ketterässä ohjelmistokehityksessä, jossa regressiotestejä voidaan ajaa päivittäin ja uudelleenkäyttöön perustuvassa ohjelmistotuotannossa, jossa ohjelmisto koostetaan valmiista koodinpätkistä, luokista tai muista komponenteista. Myös integraatiotestauksessa on hyvä aloittaa regressiotestauksesta. Ohjelmiston ylläpitoprosesseissa, jotka voivat olla luonteeltaan joko korjaavia, täydentäviä, sopeuttavia tai ennakoivia, tehdään regressiotestejä. (Binder, 1999, s. 176.)

Regressiotestauksen testitapauksina käytetään uudelleen vanhoja testejä, jotka on läpäisty onnistuneesti edellisellä testauskierroksella. Lisäksi joskus joudutaan laatimaan erillisiä regressiotestitapauksia vanhan toiminnallisuuden oikean toiminnan selvittämiseksi. Jos ohjelmaa on muutettu pyrkimyksenä korjata ohjelmasta löytynyt virhe, testataan ohjelma sillä testitapauksella, joka virheen oli löytänyt. Jos regressiotestauksessa löytyy vanhoista, aiemmin virheettömistä toiminnallisuuksista virhe, sanotaan että ohjelma on regressoitunut eli taantunut. (Holopainen, 2005.)

Regressiotestauksen testijoukkojen valitsemiseen on olemassa erilaisia tekniikoita. Näitä ovat minimointitekniikat, joissa pyritään saamaan aikaiseksi testijoukko, joka mahdollisimman vähillä testitapauksilla kattaisi suurimman osan kohteista joissa on mahdollisia virheitä, kattavuuteen perustuvat tekniikat, jotka valitsevat regressiotestitapaukset jonkin kattavuuskriteerin perusteella sekä ns. turvalliset tekniikat, joissa testitapauksiksi valitaan alkuperäisestä testijoukosta ne testit, joiden uskotaan voivan paljastaa mahdolliset virheet muutetussa ohjelmassa.(Holopainen, 2005.)

4.9 Ylläpitotestaus

Ylläpitotestaus (Maintenance testing) tarkoittaa testattavan järjestelmän muutosten jälkeistä testausta. Muutoksia voivat olla järjestelmän muokkaaminen asiakkaalle toimittamisen jälkeen virheiden korjaamiseksi, suorituskyvyn parantamiseksi tai sen mukauttamiseksi muuttuneeseen käyttöympäristöön. Ylläpitotestausta ei tule sekoittaa ylläpidettävyystestaukseen, joka on osa järjestelmä-

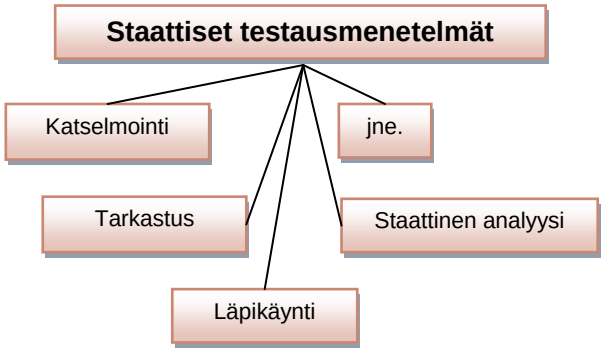
testausta ja testaa ohjelmistoa sen ylläpidettävyyden määrittelemiseksi. (ISTQB, 2007)

5 TESTAUKSEN KÄYTÄNNÖT

Tässä luvussa esitellään testauksen käytäntöjä, selvitetään staattisten ja dynaamisten testaustekniikoiden eroja ja kerrotaan testauksen laatikkotekniikoista. Luvussa selvitetään myös tekniikoita, joilla valita testisyötteitä niin, että mahdollisimman vähillä testitapauksilla saataisiin mahdollisimman kattava testaustulos.

5.2 Staattiset testausmenetelmät

Staattinen testaus tarkoittaa komponentin tai järjestelmän testausta määrittely- tai toteutustasolla suorittamatta ohjelmistoa. Kuvassa 5.1 esitettäviä staattisia testausmenetelmiä ovat katselmoinnit sekä staattinen koodin analysoiminen (ISTQB, 2007). Staattisilla menetelmillä voidaan jo varhaisessa vaiheessa löytää virheitä. Esimerkiksi suunnittelu- ja määrittelydokumenttien katselmoinneissa voidaan löytää suunnitteluvirheitä ennen kuin ohjelmistoa on lähdetty toteuttamaan. Tällöin virheiden korjaaminen on edullista, koska turhaa ohjelmointityötä ei ole tehty.



Kuva 5.1 Staattiset testausmenetelmät

5.2.1 Katselmoinnit

Katselmointi tarkoittaa sitä, että ohjelmiston dokumentteja tarkastetaan visuaalisesti vikojen löytämiseksi. Katselmoida voidaan mitä tahansa ohjelmiston vaihetuotetta: vaatimuskuvauksia, suunnittelukuvauksia, ohjelmakoodia, testaussuunnitelmia, testitapauksia, testiskripitejä, käyttöohjeita tai www-sivuja. Vikoja, joita katselmoimalla löytää, ovat esimerkiksi poikkeamat standardista tai ristiriidat ylemmän tason asiakirjojen suhteen, viat vaatimuksissa tai suunnittelussa, heikko ylläpidettävyys tai virheelliset rajapintamääritykset. Staattisella katselmoinnilla löydetäänkin enemmän vikojen aiheuttajia kuin varsinaisia virheitä. Katselmuksia voidaan suorittaa eri menettelyillä, jotka ovat enemmän tai vähemmän muodollisia. Katselmointi voi olla täysin manuaalista, mutta katselmointityökalujakin on olemassa. IEEE:n Standardi 1028-1997, ”IEEE Standard for Software Reviews”, määrittelee käytännöt johdon katselmukselle, tekniselle katselmoinnille sekä muodolliselle tarkastukselle, läpikäynnille ja auditoinnille (IEEE, 1998). Termejä ”katselmointi” (Review) ja ”tarkastus” (Inspection) käytetään kirjallisuuslähteissä vaihtelevasti niin, että puhutaan katselmoimisesta tarkoittaen itse asiassa tarkastusta, mutta tässä dokumentissa katselmoinnilla tarkoitetaan kaikenlaisia visuaalisia dokumenttien tutkintamenetelmiä ja tarkastuksella nimenomaan muodoltaan tarkkaan määrättyä tarkastustilaisuutta.

Katselmointityypin valinta riippuu siitä, mikä on katselmoinnin tarkoitus, sillä vikojen löytämisen lisäksi katselmointia voidaan suorittaa myös katselmoinnin kohteesta keskustelemiseksi tai yhteispäätöksen aikaansaamiseksi siitä, onko jokin dokumentti oikeellinen. Katselmoinnit vaihtelevat tyypeiltään hyvin epämuodollisista, ohjeistamattomista tilaisuuksista erittäin muodollisiin. Se, kuinka muodollisesti katselmointi suoritetaan, riippuu muun muassa siitä, missä vaiheessa kehitysprosessi on. Samalle dokumentille voidaan tehdä useammantasoisia katselmuksia, esimerkiksi ennen teknistä katselmointia voidaan järjestää epämuodollinen katselmus, tai vaatimusmäärittely tarkastaa ennen kuin se käydään läpi asiakkaan kanssa. (IEEE, 1998, ISTQB, 2009.)

Katselmoinnin vaiheet ovat suunnittelu, aloitus, yksilöllinen valmistautuminen, katselmointikokous, uusintatyö ja seuranta. **Suunnittelussa** valitaan henkilöt ja jaetaan roolit, sekä valitaan tarkasteltavat dokumentin osat. Muodollisissa katselmoinneissa, kuten tarkastuksissa, määritellään aloitus- ja lopetusehdot. **Aloituksessa** jaetaan tarkasteltavat dokumentit osallistujille ja selitetään heille tavoitteet ja katselmointiprosessin kulku. Muodollisissa menetelmissä tarkastetaan tässä vaiheessa myös aloitusehtojen täyttyminen. **Yksilöllinen valmistautuminen** tarkoittaa sitä, että katselmoinnin osallistujat tutustuvat dokumentteihin itsekseen ennen varsinaista katselmointikokousta. Tähän kuuluu potentiaalisten vikojen ja kysymysten kirjaaminen muistiin. Tässä vaiheessa osallistujille jaetaan usein myös tarkistuslistat, joiden perusteella dokumenttia kannattaa tutkia. Tarkistuslistat voivat olla roolipohjaisia perustuen esimerkiksi käyttäjän tai ylläpitäjän tehtäviin tai siihen, minkälaisia käyttöoperaatioita järjestelmällä on. Tarkistuslista voi myös olla yleisempi sisältäen tyypillisiä vaatimuksiin liittyviä ongelmia. **Katselmointikokous** tarkoittaa sitä, että katselmointiin osallistuvat kokoontuvat ja keskustelevat havainnoistaan, jotka luetteloidaan, ja jotka kirjataan pöytäkirjaan muodollisissa katselmuksissa. Yleensä katselmoinnissa ainoastaan keskitytään löytämään virheitä ja niiden korjaukseen liittyvät asiat jätetään käsittelemättä, mutta on myös mahdollista, että vikojen lisäksi tehdään suosituksia niiden käsittelylle. **Uusintatyöllä** tarkoitetaan löydettyjen vikojen korjaamista. Tämän suorittaa tyypillisesti katselmoitavan kohteen tekijä. **Seurannassa** tarkistetaan, että viat on käsitelty ja tarkastetaan muodollisissa katselmoinneissa lopetusehtojen täyttyminen. (IEEE, 1998, ISTQB, 2009.)

Katselmointiin osallistuvien henkilöiden roolit ovat johtaja, vetäjä, kirjuri, tekijä ja tarkastaja. Yhdellä henkilöllä voi olla useita rooleja katselmointiprosessissa. **Johtaja** päättää katselmuksen suorittamisesta ja varaa ajan projektiaikataulussa ja päättää, onko katselmoinnilla saavutettu halutut tavoitteet. Katselmoinnin **vetäjä** on henkilö, joka johtaa katselmointia. Hänen tehtäviinsä kuuluu katselmoinnin suunnittelu, kokouksen johtaminen sekä seuranta katselmuksen jälkeen. **Tekijä** on se, kenen kirjoittama katselmoitava dokumentti on, tai henkilö, jolla on päävastuu dokumentista. **Tarkastajiksi** kutsutaan henkilöitä, joilla on joku tietty tekninen tai liiketoiminnallinen tausta, ja jotka tunnistavat ja kuvaavat katselmoinnin kohteesta löytämänsä viat. Katselmoijat pyritään valitsemaan

niin, että he edustavat eri näkökulmia, jotta saataisiin mahdollisimman laaja käsitys dokumentin oikeellisuudesta. **Kirjuri** eli tallentaja toimii dokumentoijana: hän kirjaa ylös kaikki löydökset, jotka tulevat kokouksen aikana ilmi. (IEEE, 1998, ISTQB, 2009.)

Jotta katselmoinneista saataisiin mahdollisimman suuri hyöty, on syytä kiinnittää siihen, että katselmoinnilla tulee olla selkeä tavoite, ja osallistujiksi tulee valita tavoitteen kannalta oikeita ihmisiä. Lisäksi vikojen löytäminen pitäisi pystyä kokemaan positiivisena asiana ja ne tulisi esittää objektiivisesti niin, ettei tilaisuus ole katselmoitavan tuotoksen tekijälle epämiellyttävä ja tukala. Katselmointitekniikat tulisi valita katselmoinnin kohteen tason ja katselmoijien mukaan, ja tarkistuslistoja ja roolituksia käytettään soveltuvin osin auttamaan vikojen löytämisessä. Katselmoijia pitäisi kouluttaa varsinkin muodollisempien katselmointitekniikoiden, kuten tarkastusten läpiviemiseen ja keskittyä katselmointiprosessissa oppimiseen ja prosessin parantamiseen.

5.2.2 Katselmointityyppejä

Epämuodollinen katselmointi

Epämuodollinen katselmointi ei sisällä muodollista prosessia. Se voi muodostua pariohjelmoinnista, tai se voi tarkoittaa sitä, että tekninen johto katselmoi suunnitteludokumentteja tai ohjelmakoodia epämuodollisesti. Yksinkertaisimmillaan epämuodollinen katselmointi on sitä, että luetetaan tuotos työtoverilla läpi ja kysytään hänen mielipidettään. Tämänkaltaisen katselmoinnin tulokset voidaan dokumentoida haluttaessa ja menetelmän hyödyllisyys riippuu pitkälle siitä, kuka katselmoinnin suorittaa. Menetelmä on edullinen, ja siitä on mahdollista saada selville edes jotain dokumentista, jonka parissa työskennellään. (ISTQB, 2009.)

Läpikäynti

Läpikäynnillä tarkoitetaan kokousta, jonka vetäjänä on katselmoitavan dokumentin tekijä itse, eli hän esittelee tuotostaan henkilöille, jotka osallistuvat kokoukseen. Kokouksen kestoa ei ole erikseen määritelty, käytännöt ovat kirjavia, vaihdellen epämuodollisesta muodolliseen, sisältäen etukäteen valmistautumista, katselmoinnin raportointia ja havaintolistaa. Menetelmän tarkoitus on, että dokumentin tekijä selvittää muille tuotostaan ja samalla voidaan löytää vikoja. (ISTQB, 2009.)

Tekninen katselmointi

Tekninen katselmointi on dokumentoitu ja määritelty vianetsintäprosessi. Siihen osallistuvat ovat kollegoja ja teknisiä asiantuntijoita. Parhaassa tapauksessa teknisellä katselmoinnilla on koulutettu vetäjä, ei kuitenkaan tuotteen tekijä. Teknisiin katselmointeihin kuuluu valmistautuminen ennen kokousta, mutta tarkistuslistojen käyttö, raportointi ja havaintolistan kirjoittaminen ovat valinnaisia, ja tilaisuuksien muodollisuus vaihtelee. Tämänkaltaisen katselmoinnin tarkoitukset ovat tehdä päätöksiä ja arvioida vaihtoehtoja, sekä ratkaista teknisiä ongelmia ja tarkastaa dokumentin yhdenmukaisuus määrittelyjen ja standardien suhteen. (ISTQB, 2009.)

Tarkastus

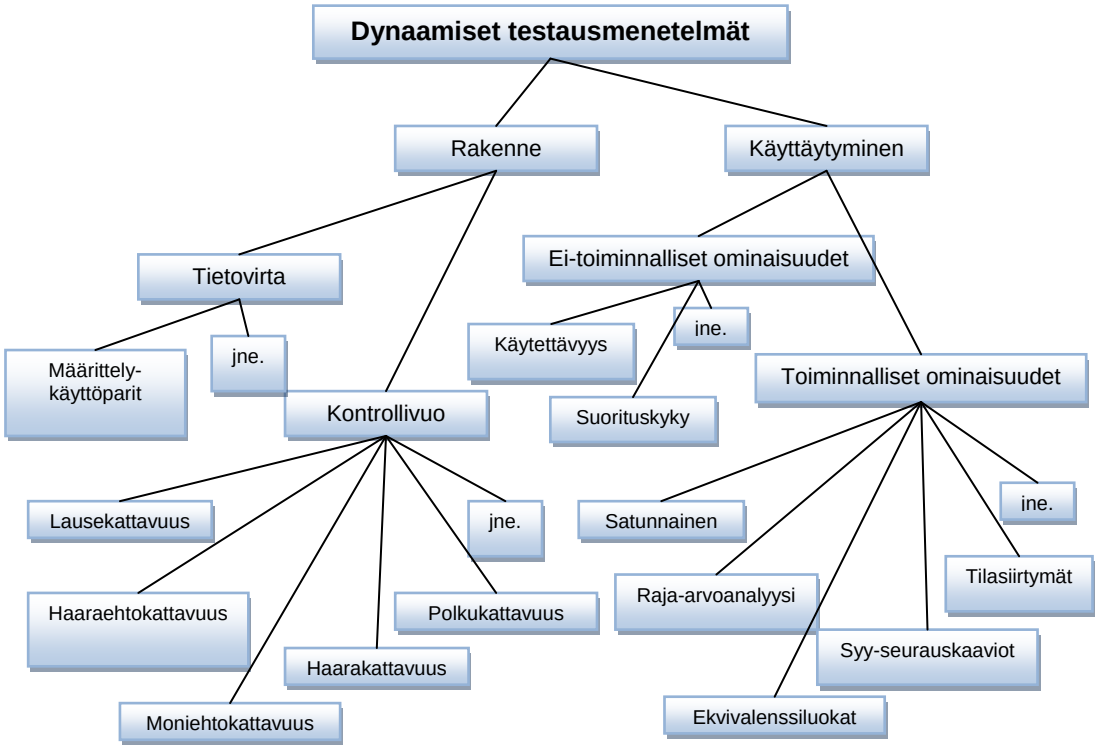
Tarkastus on muodollinen prosessi, joka perustuu sääntöihin ja tarkistuslistoihin, ja jolla on selkeät aloitus- ja lopetuskriteerit. Tarkastusta vetää koulutettu vetäjä, joka ei ole tarkastuksen kohteena olevan dokumentin tekijä, ja tarkastuksessa kaikilla osallistujilla on määritetyt roolit. Tarkastukseen valmistaudutaan ennen kokousta ja tarkastusraporttien ja havaintolistojen kirjoittaminen kuuluu prosessiin. Tarkastuksella on myös muodollinen seurantaprosessi, joka tutkii, että vikoihin, jotka tarkastuksessa löydetään, puututaan jollain tavalla. Tarkastuksen päätavoite on löytää vikoja. (ISTQB, 2009)

5.2.3 Staattinen analyysi

Staattinen analyysi on rakenteellista testausta, menetelmä, jonka tavoitteena on löytää vikoja lähdekoodista ja ohjelmistomalleista. Staattinen analyysi suoritetaan niin, että tutkittavaa koodia ei suoriteta, kuten dynaamisessa testauksessa, vaan se syötetään ohjelmaan, joka analysoi sitä löytääkseen vikoja. Staattisen analyysin edut ovat siinä, että vikoja löytyy jo ennen dynaamisen testauksen aloittamista. Mittaritietojen avulla saadaan varoituksia koodin tai suunnittelun heikkouksista, sillä ne varoittavat esimerkiksi ohjelmakoodin korkeasta kompleksisuusluvusta. Staattisen analyysin avulla löydetään riippuvuuksia ja epäjohdonmukaisuuksia ohjelmistomalleista ja parannetaan koodin ja suunnittelun ylläpidettävyyttä. Sen löytämiä tyypillisiä vikoja ovat viittaaminen muuttujaan, jolle ei ole määritetty arvoa, käyttämättömät muuttujat, osoittimien ja indeksien väärinkäyttö, parametrien väärinkäyttö, saavuttamaton koodi eli ”kuollut koodi”, ohjelmointistandardien vastainen koodi, tietoturvaheikkoudet ja syntaksivirheet. Staattista analyysia käytetään komponentti- ja integraatiotestauksessa, ja ohjelmoijan tulee hallita käyttämänsä analysointityökalut hyvin voidakseen käyttää niitä tehokkaasti. Jos analyysin suorittaja ei osaa tulkita kunnolla analysaattorin antamia ilmoituksia, on analysointi tehotonta. Kääntäjä (compiler) tukee jonkin verran staattista analyysia, ja oikeastaan kääntäjää voi pitää staattisen testivälineen triviaalina tapauksena. (ISTQB, 2009)

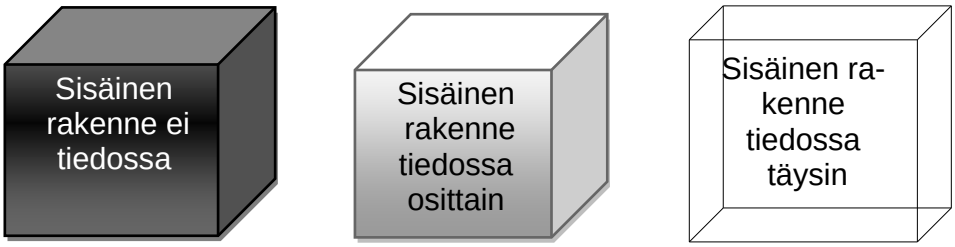
5.3 Dynaamiset testausmenetelmät

Dynaaminen testaus on testausta, joka tehdään suorittamalla testattavaa ohjelmaa selvittääkseen ohjelman ja ohjelmakoodin suorituksenaikaista käyttäytymistä (ISTQB, 2007). Dynaaminen testaus voidaan jakaa rakenteen ja käyttäytymisen testaamiseen, joista käyttäytymisen testaus jakautuu kahteen alaryhmään: toiminnallisten ja ei-toiminnallisten ominaisuuksien testaukseen. Rakenteen testaaminen jakautuu tietovuon ja kontrollivuon testaukseen. (kuva 5.2).



Kuva 5.2 Dynaamiset testausmenetelmät

Testaustekniikoita, joita tässä käsitellään, ovat lasilaatikko- mustalaatikko ja harmaalaatikkotestaus. Ohjelmiston testauksessa tarvitaan yleensä ainakin kumpaakin kahdesta ensin mainituista kattavan testaustuloksen aikaansaamiseksi. Tekniikat eroavat toisistaan kuvan 5.3 osoittamalla tavalla:



Kuva 5.3 Eri testaustekniikoiden käsitteelliset eroavaisuudet

Mustalaatikkotestaus on vaatimuksiin perustuvaa, ja testitapaukset perustuvat toiminnallisiin määrittelyihin, sillä ohjelman rakennetta ei tunneta. Testitapaukset suoritetaan ohjelmaa käyttämällä. Harmaalaatikkotestauksen testitapauksilla saadaan sitä enemmän syvyyttä testaukseen, mitä enemmän ohjelman rakenteesta tiedetään. Lasilaatikkotestaus vaatii mahdollisuuden suorittaa koodia realistisilla syötteillä, ja siinä ohjelman sisäinen rakenne on testaajan tiedossa. Mustalaatikkotestausta ja harmaalaatikkotestausta voidaan käyttää hyväksymistestauksesta, koska se onnistuu systeemin loppukäyttäjiltäkin, mutta lasilaatikkotestausta tekevät yleisesti ottaen kehittäjät ja testaajat.

5.3.1 Rakenteen testaus ja lasilaatikkotekniikat

Rakenteen testaamisessa käytetään staattisia menetelmiä ohjelman dynaamisen suorituksen analysointiin. Esimerkiksi kontrollivuon (Control flow), joka tarkoittaa tapahtumapolkujen suoritusjärjestystä komponentin tai järjestelmän läpi, tilaa testataan kattavuusyksiköiden selvittämiseksi. Kattavuusyksiköt ovat prosenttilukuja, jotka ilmoittavat, missä määrin suoritettu testijoukko on käsitellyt kyseistä kattavuusaluetta. Tietovirta-analyysissa (Data flow analysis) analysoidaan muuttujien määrittämistä ja käyttöä, esimerkiksi oliotestauksessa olioiden luomisjärjestystä ja niiden tilan muutoksia. (ISTQB, 2007.)

Lasilaatikkotestauksella eli rakennepohjaisella testauksella (white box testing, glass-box testing, clear box testing, structural testing) tarkoitetaan testausta, joka perustuu ohjelman tai komponentin näkyvissä olevan rakenteen analyysiin, eli testaajan tiedossa olevia asioita ohjelmiston toteutuksesta, kuten suunnitteludokumentteja ja ohjelmakoodia voidaan käyttää testitapausten kirjoittamisessa. Lasilaatikkotestaukselle on ominaista myös se, että testitapausten kattavuutta pystytään mittaamaan ja sitä voidaan lisätä suunnittelemalla kattavuustestitapauksia. Testausta voidaan tehdä eri tasoilla: komponenttitestauksessa tutkitaan nimenomaan lähdekoodin rakennetta, eli sen lauseita, ehtoja ja haaraumia. Integrointitasolla rakenne voi käydä ilmi kaaviosta, joka kertoo, miten moduulit liittyvät toisiinsa ja missä järjestyksessä ne kutsuvat toisiaan, ja järjestelmätasolla lasilaatikkotekniikalla voidaan testata esimerkiksi valikkorakennetta, liiketoimintaprosessia tai www-sivun rakennetta. (ISTQB, 2009)

Kuten todettiin, lasilaatikkotestausta suoritettaessa lasketaan erilaisia koodikattavuuslukuja, jotka kertovat, kuinka suuren prosentuaalisen osan ohjelmakoodista testijoukko on käynyt läpi. Kattavuuslukuja ovat muun muassa lausekattavuus (statement coverage), päätöskattavuus (decision coverage), ehtokattavuus (condition coverage), moniehtokattavuus (multiple condition coverage) sekä haarakattavuus (branch coverage). Kattavuuksien laskemiseen on olemassa ohjelmallisia työkaluja. (ISTQB, 2007.)

Lausekattavuus

Yksikkötestauksessa lausekattavuudella arvioidaan, kuinka monta prosenttia suoritettavista lauseista testijoukko on käynyt läpi. Lauseella tarkoitetaan ohjelmointikielen yksikköä, joka on ohjelman suorituksen pienin jakamaton osa. Lausetestauksessa testitapaukset laaditaan yksittäisten lauseiden suorittamiseksi, jotta lausekattavuutta saataisiin kasvatettua. (ISTQB 2009, ISTQB, 2007.)

Päätöskattavuus

Päätöskattavuudella arvioidaan, kuinka monta prosenttia päätösten tuloksista testijoukko on käynyt läpi. Päätöksillä tarkoitetaan esimerkiksi if-lauseen vaihtoehtoja tosi ja epätosi, ja päätöstestauksen testitapaukset laaditaan niin, että testeissä suoritetaan mahdollisimman monta eri päätöstulosta, eli testitapaus suorittaa sekä tosi- että epätosi- päätöksiin päätyvät haarat. Päätöstestaus on kontrollivuotestauksen muoto, koska siinä luodaan tietty kontrollivuo päätöskohtien läpikäymiseksi. Päätöskattavuus on vahvempi mittaustulos kuin lausekattavuus, sillä 100 % päätöskattavuus tarkoittaa 100% lausekattavuutta, mutta täydellinen lausekattavuus ei takaa täydellistä päätöskattavuutta. (ISTQB 2009, ISTQB, 2007.)

Ehtokattavuus ja moniehtokattavuus

Ehtokattavuus kertoo, kuinka monta prosenttia ehtojen tuloksista testijoukko on käynyt läpi. 100 % ehtokattavuus tarkoittaa että jokainen yksittäinen ehto jokaisessa päätöslausekkeessa on testattu sekä totena että epätotena. Ehtotestauksen testitapaukset suunnitellaan suorittamaan ehtojen lopputuloksia. Moniehtokattavuus on prosenttiosuus, joka kertoo testijoukon suorittamien yksittäisten ehtojen tuloksien yhdistelmien osuuden yhdessä lauseessa. Moniehtotestauksen testitapaukset suunnitellaan niin, että ne toteuttavat yksittäisten ehtojen tuloksien yhdistelmiä yhdessä lauseessa. (ISTQB, 2007.)

Haarakattavuus

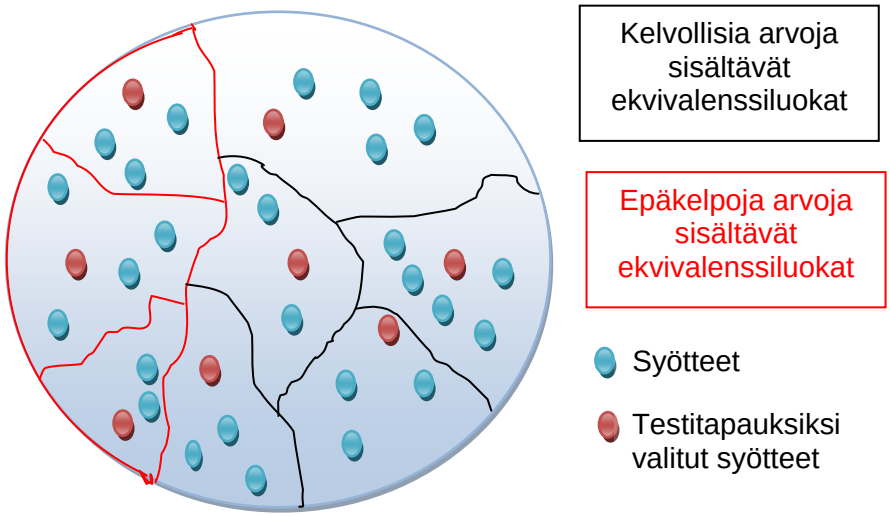
Haaralla (Branch) tarkoitetaan ohjelman peruslohkoa, joka voidaan valita suoritettavaksi tilanteessa, jossa on useampia vaihtoehtoisia ohjelmapolkuja, kuten switch-rakenteessa tai if-then-else-rakenteessa. Haara- tai haarautumiskattavuutta testataan laatimalla testitapauksia, jotka suorittavat eri haarojen koodit, ja haarakattavuus kertoo, kuinka monta prosenttia testijoukko on käynyt läpi testauksen kohteen haaroista. 100 % haarakattavuus merkitsee sitä, että testauksen kohteen päätöskattavuus ja lausekattavuus ovat myös 100%.

5.3.3 Käyttäytymisen testaus ja mustalaatikkotekniikat

Käyttäytymisen testaus jakautuu ohjelman käyttäytymisen toiminnallisten ja ei-toiminnallisten ominaisuuksien testaukseen, ja perustuu testattavan kohteen määrittelyyn. Ohjelmistojen käyttäytymistä testataan ns. mustalaatikkotekniikoilla (Black box testing). Testaus tapahtuu tutkimalla järjestelmän syötteitä (input) ja vasteita (output) ja vertaamalla vasteita odotettavissa oleviin vasteisiin. Testitapaukset johdetaan kohteen määrittelyn sekä testaajien ja käyttäjien kokemuksen perusteella tietämättä mitään testattavan kohteen sisäisestä rakenteesta. Mustalaatikkotestauksessa, kuten muissakin testaustekniikoissa joudutaan sen tosiasian eteen, ettei kaikkea voi testata, koska järjestelmät ovat monimutkaisia ja kaiken kattava testaus mahdotonta jo laskennallisesti. Tarvitaan menetelmiä, joilla saada aikaan tehokkaita testijoukkoja mahdollisimman pienellä määrällä testitapauksia. Mustalaatikkotekniikoissa voidaan käyttää erilaisia menetelmiä testitapausten ja niiden syötteiden valitsemiseksi, kuten testitapausten jakoa ekvivalenssiluokkiin (Equivalence partitioning), raja-arvoanalyysia (Boundary value analysis), päätöstaulutestausta (Decision table testing), tilasiirtymätestausta (State transition testing), syötteiden parittaista testausta (Pairwise testing), käyttötapaustestausta (Use case testing) ja virheen arvausta (Error guessing). (ISTQB, 2007, ISTQB, 2009)

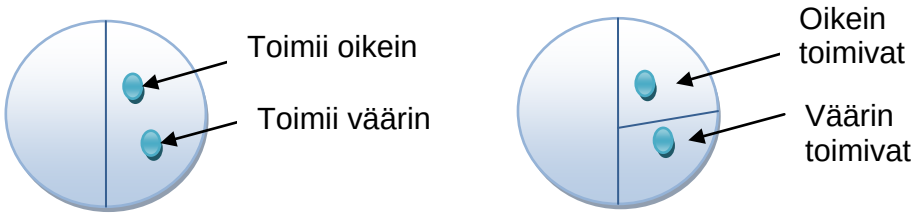
Ekvivalenssiluokkiin jako

Jako ekvivalenssiluokkiin on testaustekniikka, jossa testitapaukset suunnitellaan niin, että testitapaukset kattavat jokaisen ekvivalenssiluokan ainakin kerran. Ekvivalenssiluokkajako tarkoittaa sitä, että syötteet jaetaan luokkiin sen mukaan, että niiden toiminta on samantapaista, ja oletetaan, että samaan ekvivalenssiluokkaan kuuluvat syötteet toimivat samalla tavalla, jolloin mikä tahansa ekvivalenssiluokan jäsen kelpaa testitapaukseksi. Ekvivalenssiluokat voidaan jakaa vielä kelvollisiin (valid) ja epäkelpoihin (invalid), ja valita syötteitä kattavasti kummastakin luokasta (kuva 5.4).



Kuva 5.4 Testitapausten jako ekvivalenssiluokkiin

Menetelmän ongelma on se, että samaan ekvivalenssiluokkaan sijoitetut syötteet eivät välttämättä todellisuudessa kuulukaan siihen. Jos testauksen aikana ekvivalenssihypoteesi osoittautuu epäkelvoksi, on ekvivalenssiluokkia toisinaan pilkottava osiin, kuten kuvassa 5.5.



Kuva 5.5 Ekvivalenssiluokan jakautuminen

Syötteiden luokkiin jakamisessa voidaan käyttää avuksi dokumentteja, jotka kuvaavat ohjelmiston arkkitehtuuria, käyttäjävaatimuksia ja järjestelmävaatimuksia.

Esimerkki, jossa työnhakijan ikä rajaa sen, voiko hänet palkata:

IKÄ	STATUS
0-16	Ei voi palkata
16-18	Voi palkata osa-aikaiseksi
18-55	Voi palkata täysiaikaiseksi työntekijäksi
55-99	Ei voi palkata

Ohjelmallisesti kirjoitettuna:

```
If (tyonhakijanIka >= 0 && tyonhakijanIka <=16)        palkStatus="EI";
If (tyonhakijanIka >= 16 && tyonhakijanIka <=18)       palkStatus="OSA";
If (tyonhakijanIka >= 18 && tyonhakijanIka <=55)       palkStatus="TAYSI";
If (tyonhakijanIka >= 55 && tyonhakijanIka <=99)       palkStatus="EI";
```

Kuten huomataan, raja-arvoilla tulee ongelmia, sillä määrittelyn mukaan sama ikä antaa erilaisia palkkausstatuksia. Toteutusta ei voi tehdä tällä tavalla. Ekvivalenssiluokkia on muutettava vaikkapa seuraavanlaiseksi:

IKÄ	STATUS
0-15	Ei voi palkata
16-17	Voi palkata osa-aikaiseksi
18-54	Voi palkata täysiaikaiseksi työntekijäksi
55-99	Ei voi palkata

Toteutus näyttää silloin tältä:

```
If (tyonhakijanIka >= 0 && tyonhakijanIka <=15)        palkStatus="EI";
If (tyonhakijanIka >= 16 && tyonhakijanIka <=17)       palkStatus="OSA";
If (tyonhakijanIka >= 18 && tyonhakijanIka <=54)       palkStatus="TAYSI";
 If (tyonhakijanIka >= 55 && tyonhakijanIka <=99)       palkStatus="EI";
```

Esimerkin kelvolliset ekvivalenssiluokat ovat luvut 0-15, 16-17, 18-54 ja 55-99. Epäkelpojen syötteiden ekvivalenssiluokka käsittää esimerkiksi kirjaimet, erikoismerkit sekä negatiiviset kokonaisluvut, luvut, jotka ovat suurempia kuin 99 ja desimaaliluvut. Ekvivalenssiluokkien perusteella testitapaukset voisivat olla {8}, {16}, {35}, {79}, sekä {ks, #, -45, 122, 6.78787}.

Raja-arvoanalyysi

Raja-arvolla tarkoitetaan syötteen tai vasteen arvoa, joka on ekvivalenssiluokan reuna-arvo tai sen jommankumman reunan pienin arvo, esimerkiksi arvojoukon minimi- ja maksimiarvo. Raja- tai reuna-arvoanalyysiksi (Boundary value testing, BVT), kutsutaan tekniikkaa, jossa testitapauksiksi valitaan rajoilla olevia, äärimmäisiä arvoja. Näitä voivat olla esimerkiksi mahdollisimman lyhyt ja pitkä syöte tai mahdollisimman pieni ja suuri syöte. Raja-arvoanalyysin peruste on se, että jos testi epäonnistuu rajojen arvoilla, se yleensä epäonnistuu myös rajojen sisällä olevilla arvoilla. Tekniikalla saadaan testitapausten lukumäärää pienennettyä.

Edellisen kappaleen esimerkkiin perustuen raja-arvoanalyysin testitapaukset määriteltäisiin seuraavalla tavalla:

IKÄ	STATUS
0-15	Ei voi palkata
16-17	Voi palkata osa-aikaiseksi
18-54	Voi palkata täysiaikaiseksi työntekijäksi
55-99	Ei voi palkata

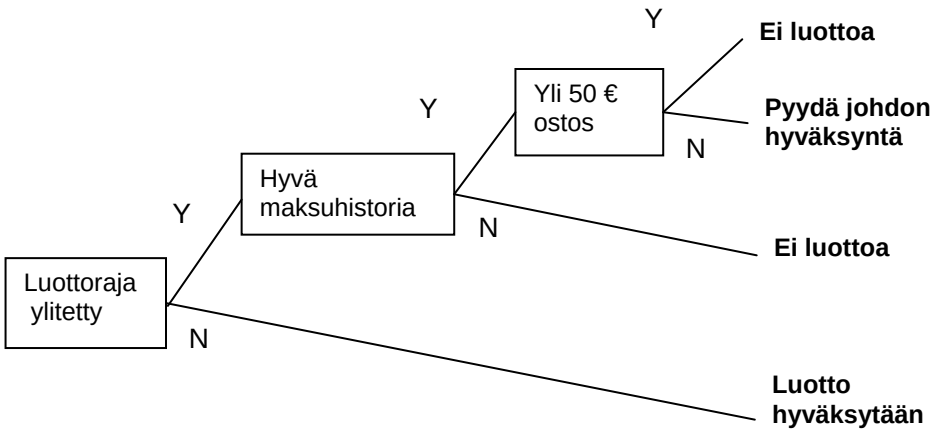
```
If (tyonhakijanIka >= 0 && tyonhakijanIka <=15)          palkStatus="EI";
If (tyonhakijanIka >= 16 && tyonhakijanIka <=17)         palkStatus="OSA";
If (tyonhakijanIka >= 18 && tyonhakijanIka <=54)         palkStatus="TAYSI";
If (tyonhakijanIka >= 55 && tyonhakijanIka <=99)        palkStatus="EI";
```

Raja-arvoanalyysissa testitapauksiksi valitaan seuraavat arvot juuri rajojen ulkopuolella, nimenomainen raja-arvo sekä seuraavat arvot juuri rajojen sisäpuolella, joten tämän esimerkin raja-arvotestauksen syötteet ovat {-1,0,1}, {15, 16, 17}, {17, 18, 19}, {54, 55, 56}, ja {98, 99, 100}.

Päätöstaulutestaus

Päätöstaulu on taulukko, joka näyttää syötteiden tai herätteiden, eli syiden yhdistelmät ja niihin liittyvät tulokset eli vaikutukset, joita voidaan käyttää suunnitellessa testitapauksia. Päätöstaulutestaus on tekniikka, jossa testitapaukset suunnitellaan päätöstaulun perusteella (ISTQB 2007). Menetelmän etuna on, että sillä tulee luotua sellaisia ehtojen yhdistelmiä, joita ei muuten tulisi testattua ollenkaan. Menetelmä on käyttökelpoinen tilanteissa, joissa ohjelmiston toiminta etenee loogisten päätösten seurauksena (ISTQB 2009).

Päätöstaulun laatiminen tapahtuu seuraavasti: ensin tunnistetaan testattavasta materiaalista kaikki ehdot, eli määritelmän mukaan syötteet tai herätteet, jotka on täytettävä sekä kaikki muuttujat, joita ehdot koskevat. Tämän jälkeen lasketaan sääntöjen kokonaismäärä, joka on muuttujien määrä kerrottuna päätöshaarojen määrällä. Taulukon vaakariveillä luetellaan ensin ehdot eli syötteet ja niiden alle toimenpiteet, eli tulokset, ja varataan riittävä määrä sarakkeita säännöille. Ehdot voidaan tunnistaa esimerkiksi rakentamalla kuvan 5.6 kaltainen päätöspuu.



Kuva 5.6 Päätöspuu binäärimuuttujista

Edellä kuvatussa päätöspuussa on päättyviä haaroja neljä, ja muuttujan arvoja kaksi, joten sääntöjä on 8. Taulukko täytetään aloittamalla alimmasta muuttujasta ja luettelemalla kaikki mahdolliset arvot. Jos kyseessä on binäärimuuttuja, on mahdollisia arvoja kaksi, kyllä (Y) ja ei (N). Muuttujavaihtoehdot muodostavat ryhmän, tässä esimerkissä YN. Ryhmää toistetaan alimman muuttujan kohdalla kunnes rivi on täynnä. Tämän jälkeen täytetään seuraavaksi alin muuttujarivi, jolla on samat arvot kuin alimmallakin muuttujalla. Tämän muuttujan arvoa asetetaan sarakkeisiin peräkkäin yhtä monta kappaletta vasemmalta oikealle kuin mikä oli alimman muuttujan ryhmän koko, eli esimerkin mukaan kaksi kappaletta, eli kaksi kappaletta N:ää ja kaksi kappaletta Y:tä. Ryhmän koko on nyt 4, YYNN. Tätä ryhmää toistetaan kunnes rivi on täynnä. Taas seuraavalle riville laitetaan edellisen rivin ryhmäkoon verran kumpaakin muuttujaa, eli ryhmäkooksi tulee 8, YYYYNNNN. Tällä tavalla jatketaan, kunnes koko taulukon säännöt on merkitty. Kun taulukko on valmis, käydään se läpi sääntö kerrallaan ja merkitään rasti sopivien toimenpiteiden kohdalle (Kuva 5.7):

Luottoraja ylitetty	Y	Y	Y	Y	N	N	N	N
Hyvä maksuhistoria	Y	Y	N	N	Y	Y	N	N
Yli 50 € ostos	Y	N	Y	N	Y	N	Y	N
Pyydä johdon hyväksyntä		x						
Luotto hyväksytään					x	x	x	x
Ei luottoa	x		x	x				

Kuva 5.7 Päätöstaulu binäärimuuttujista, ensimmäinen vaihe

Näin muodostettua päätöstaulua voidaan usein yksinkertaistaa poistamalla päällekkäiset säännöt. Se tapahtuu etsimällä sääntöparit joilla on samat toimenpiteet ja joiden muuttujien arvot poikkeavat toisistaan ainoastaan yhden muuttujan kohdalla. Sääntöpari korvataan yhdellä säännöllä ja yhdentekevän ehdon kohdalle asetetaan viiva. Tämä toistetaan kaikille niille sääntöpareille, jotka täyttävät edellä mainitut kriteerit (Kuvat 5.8, 5.9 ja 5.10):

Luottoraja ylitetty	Y	Y	Y	Y	N	N	N	N
Hyvä maksuhistoria	Y	Y	N	N	Y	Y	N	N
Yli 50 € ostos	Y	N	Y	N	Y	N	Y	N
Pyydä johdon hyväksyntä		x						
Luotto hyväksytään					x	x	x	x
Ei luottoa	x		x	x				

Kuva 5.8 Päätöstaulu binäärimuuttujista, sääntöparien etsintä 1

Luottoraja ylitetty	Y	Y	Y	N	N
Hyvä maksuhistoria	Y	Y	N	Y	N
Yli 50 € ostos	Y	N	-	-	-
Pyydä johdon hyväksyntä		x			
Luotto hyväksytään				x	x
Ei luottoa	x		x		

Kuva 5.9 Päätöstaulu binäärimuuttujista, sääntöparien etsintä 2

Luottoraja ylitetty	Y	Y	Y	N

Hyvä maksuhistoria	Y	Y	N	-
Yli 50 € ostos	Y	N	-	-
Pyydä johdon hyväksyntä		x		
Luotto hyväksytään				x
Ei luottoa	x		x	

Kuva 5.10 Valmis päätöstaulu binäärimuuttujista

Päätöstaulujen avulla saadaan testitapausten määrää vähennettyä ilman, että testitapausten kattavuus vähenisi. Testisyötteet valitaan niin, että päätöstaulun säännöt täyttyvät, suoritetaan testit ja tutkitaan, poikkeaako lopputulos odotetusta lopputuloksesta (Kuva 5.10).

Tapahtuma	Syötteet			
Luottoraja ylitetty	yli luottorajan	yli luottorajan	yli luottorajan	saldo ok
Hyvä maksuhistoria	luottotie-dot ok	luottotie-dot ok	merkintä luottotie-doissa	-
Yli 50 € ostos	65 €	45€	-	-
Toimenpide	Odotetut lopputulokset			
Pyydä johdon hyväksyntä		x		
Luotto hyväksytään				x
Ei luottoa	x		x	

Kuva 5.11 Päätöstaulun avulla valitut testitapaukset

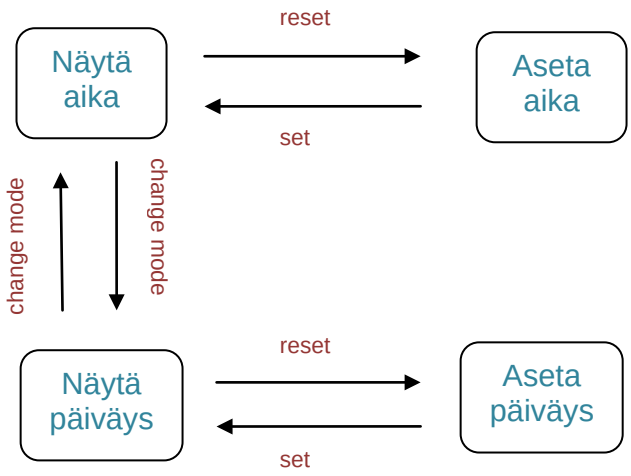
Tilasiirtymätestaus

Tilasiirtymätestaus on testitapausten suunnittelutekniikka, jossa testitapaukset suunnitellaan suorittamaan kelvollisia ja epäkelvollisia tilasiirtymiä Siirtymiä voidaan testata eri tasoilla erilaisten testikattavuuksien saavuttamiseksi, jolloin puhutaan n-siirtymätestauksesta, kuvaten n-kirjaimella niiden tasojen määrää, joiden kaikki siirtymät testataan. (ISTQB, 2007). Tilasiirtymätestauksen käyttökohteet ovat sulautetuissa järjestelmissä ja teknisessä automaatiossa, mutta sillä

voidaan mallintaa muutakin, kuten liiketoiminnan osia, joilla on tietyt tilat tai navigoimista www-sovellusten näytöstä toiseen (ISTQB, 2009).

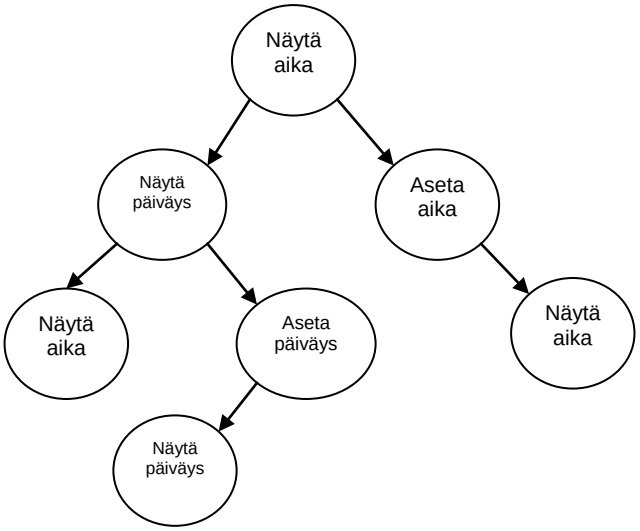
Tilasiirtymätestauksen testitapaukset suunnitellaan niin, että ensin mallinnetaan järjestelmästä tilasiirtymäkaavio joka määrittelee jokaisen tilan, jossa järjestelmässä voi esiintyä ja näiden tilojen siirtymät: aloitustila, syöte (input), vaste (output) ja lopetustila. Seuraavaksi piirretään testipuu, jonka tasojen määrästä riippuu se, kuinka kattavaa testaus on. Testitapaukset laaditaan seuraamaan tilasiirtymäpuun haaroja seuraten.

Esimerkkitapaukseksi otetaan digitaalikello, jossa on neljä eri toimintoa: ”näytä aika”, ”muuta aikaa”, ”näytä päiväys” ja ”muuta päiväystä”. Siinä on kolme painiketta, joista ”change mode”-painike vaihtaa kellon näkymää ajan ja päivämäärän näytön välillä, ”reset”-painike tekee vaihdon ajan näyttämisestä ajan asettamiseen tai päiväyksen näyttämisestä päiväyksen asettamiseen ja ”set”-painike palauttaa kellon ajan asettamisesta ajan näyttämiseen tai päivämäärän asettamisesta päivämäärän näyttämiseen (kuva 5.12).



Kuva 5.12 Digitaalikellon tilasiirtymäkaavio

Tilasiirtymistä laaditaan puu, jossa esitetään mahdolliset siirtymät kuvan 5.13 osoittamalla tavalla:



Kuva 5.13 Digitaalikellon toiminnan tilasiirtymäpuu 0-tasolla

Tilasiirtymäpuun avulla laaditaan testit, kulkien läpi jokaisen haaran kaikki tilasiirtymät puussa vasemmalta oikealle (kuva 5.14):

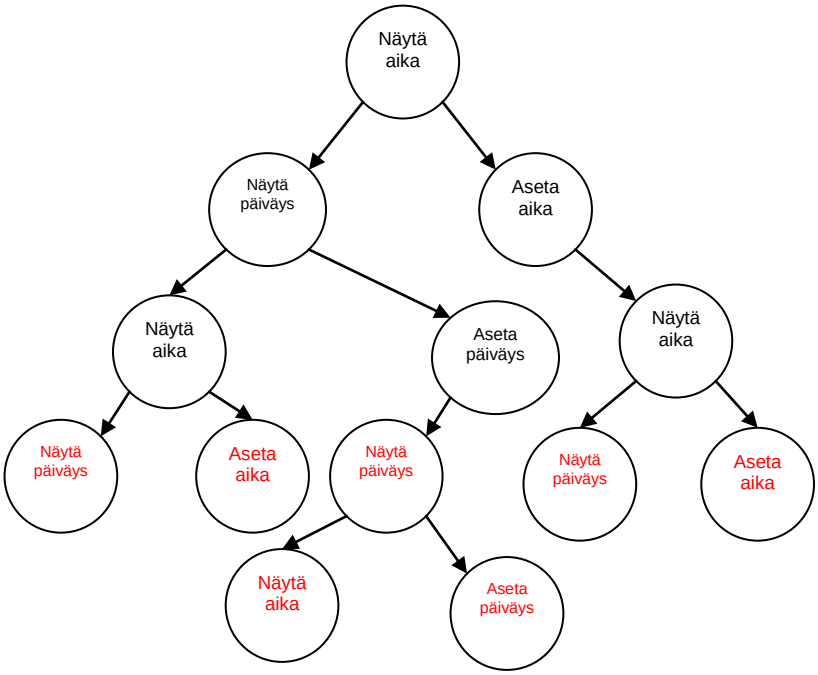
TESTI 1	vaihe 1	vaihe 2
aloitustila	NÄYTÄ AIKA	NÄYTÄ PÄIVÄYS
syöte	CHANGE MODE	CHANGE MODE
vaste	NÄYTÄ PÄIVÄYS	NÄYTÄ AIKA
lopetustila	NÄYTÄ PÄIVÄYS	NÄYTÄ AIKA

TESTI 2	vaihe 1	vaihe 2	vaihe 3
aloitustila	NÄYTÄ AIKA	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS
syöte	CHANGE MODE	RESET	SET
vaste	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS	NÄYTÄ PÄIVÄYS
lopetustila	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS	NÄYTÄ PÄIVÄYS

TESTI 3	vaihe 1	vaihe 2
aloitustila	NÄYTÄ AIKA	NÄYTÄ AIKA
syöte	RESET	SET
vaste	ASETA AIKA	NÄYTÄ AIKA
lopetustila	ASETA AIKA	NÄYTÄ AIKA

Kuva 5.14 Digitaalikellon 0-tason tilasiirtymäpuun testijoukot

Haluttaessa enemmän tasoja, testauksen kattavuuden lisäämiseksi, lisätään puuhun uusia haaroja (kuva 5.15):



Kuva.5.15 Digitaalikellon tilasiirtymäpuu 1-tasolla

Tällöin testiryhmien määrä lisääntyy, koska jokaiselle päättyvälle haaralle on oma testiryhmänsä (kuva 5.16):

TESTI 1	vaihe 1	vaihe 2	vaihe 3
aloitustila	NÄYTÄ AIKA	NÄYTÄ PÄIVÄYS	NÄYTÄ AIKA
syöte	CHANGE MODE	CHANGE MODE	CHANGE MODE
vaste	NÄYTÄ PÄIVÄYS	NÄYTÄ AIKA	NÄYTÄ PÄIVÄYS
lopetustila	NÄYTÄ PÄIVÄYS	NÄYTÄ AIKA	NÄYTÄ PÄIVÄYS

TESTI 2	vaihe 1	vaihe 2	vaihe 3
aloitustila	NÄYTÄ AIKA	NÄYTÄ PÄIVÄYS	NÄYTÄ AIKA
syöte	CHANGE MODE	CHANGE MODE	RESET
vaste	NÄYTÄ PÄIVÄYS	NÄYTÄ AIKA	ASETA AIKA
lopetustila	NÄYTÄ PÄIVÄYS	NÄYTÄ AIKA	ASETA AIKA

TESTI 3	vaihe 1	vaihe 2	vaihe 3	vaihe 4
aloitustila	NÄYTÄ AIKA	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS	NÄYTÄ PÄIVÄYS
syöte	CHANGE MODE	RESET	SET	CHANGE MODE
vaste	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS	NÄYTÄ PÄIVÄYS	NÄYTÄ AIKA
lopetustila	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS	NÄYTÄ PÄIVÄYS	NÄYTÄ AIKA

TESTI 4	vaihe 1	vaihe 2	vaihe 3	vaihe 4
aloitustila	NÄYTÄ AIKA	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS	NÄYTÄ PÄIVÄYS
syöte	CHANGE MODE	RESET	SET	RESET
vaste	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS
lopetustila	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS	NÄYTÄ PÄIVÄYS	ASETA PÄIVÄYS

TESTI 5	vaihe 1	vaihe 2	vaihe 3
aloitustila	NÄYTÄ AIKA	ASETA AIKA	NÄYTÄ AIKA
syöte	RESET	SET	CHANGE MODE
vaste	ASETA AIKA	NÄYTÄ AIKA	NÄYTÄ PÄIVÄYS
lopetustila	ASETA AIKA	NÄYTÄ AIKA	NÄYTÄ PÄIVÄYS

TESTI 6	vaihe 1	vaihe 2	vaihe 3
aloitustila	NÄYTÄ AIKA	ASETA AIKA	NÄYTÄ AIKA
syöte	RESET	SET	RESET
vaste	ASETA AIKA	NÄYTÄ AIKA	ASETA AIKA
lopetustila	ASETA AIKA	NÄYTÄ AIKA	ASETA AIKA

Kuva 5.16 Digitaalikellon 1-tason tilasiirtymäpuun testijoukot

Syötteiden parittainen testaus

Syötteiden parittaisen testauksen testitapaukset suunnitellaan siten, että ne suorittavat kaikki mahdolliset syöttöparametrien parien yhdistelmät (ISTQB, 2007) Parittaisia yhdistelmiä etsitään käyttämällä ortogonaalista matriisia, joka on matemaattisin perustein laadittu kaksiulotteinen matriisi jonka rakenne sisältää matemaattisia yhtälöitä, ja joka toimii niin, että valitaan mitkä tahansa kaksi saraketta, antaa matriisi kaikki pariyhdistelmät jokaiselle matriisin numerolle. Tämä menetelmä on myös tapa supistaa mahdollista testijoukkoa järkevällä tavalla. Menetelmää ei harjoiteta yleensä manuaalisesti vaan tarkoitukseen on kehitetty erilaisia työkaluja, joilla generoida testitapaukset.

Käyttötapaustestaus

Käyttötapaustestaus tarkoittaa testisuunnittelutekniikkaa, jossa testitapaukset suunnitellaan seuraten erilaisia käyttäjäskenaarioita (ISTQB, 2007). Testit kuvaavat toimijoiden (joko ihmiskäyttäjä, komponentti tai toinen järjestelmä) ja järjestelmän vuorovaikutusta. Käyttötapauksilla tulee olla esiehdot, joiden täyttyminen mahdollistaa käyttötapauksen onnistuneen suorituksen, ja jokainen käyttötapaus päättyy jälkiehtoihin, joihin kuuluu käyttötapauksen suorittamisen tulos ja järjestelmän lopputila käyttötapauksen päättyessä. Käyttötapaustestaus, ja varsinkin sen todennäköisimmän etenemispolun testaaminen, on hyödyllistä hyväksymistestauksissa, koska asiakkaan tai käyttäjän suorittamina ne löytävät helposti vikoja prosessin etenemisessä. (ISTQB, 2009.)
Esimerkkinä käyttötapaustestauksesta yksinkertainen Internet-sovelluksen sisäänkirjautuminen (kuva 5.17):

Käyttötapaus	Sisäänkirjautuminen
Osallistujat	Rekisteröitynyt asiakas
Alkuehdot	Asiakkaan tulee olla rekisteröitynyt
Kuvaus	Asiakas saapuu järjestelmän etusivulle, jossa on linkki sisäänkirjautumiseen. Linkkiä seuraten asiakas siirtyy sisäänkirjautumissivulle, jossa hän syöttää käyttäjätunnuksen ja salasanan. Jos ne ovat oikein, käyttäjä on sisäänkirjautunut.
Poikkeukset	Käyttäjätunnus ja/tai salasana on väärä. Järjestelmä kertoo virheellisestä syötteestä ja pyytää tarkistamaan tunnuksen. Asiakkaalle annetaan mahdollisuus tilata järjestelmältä salasana käyttäjätunnusta vastaavaan sähköpostiosoitteeseen.
Lopputulos	Asiakas on kirjautunut sisään järjestelmään.

Kuva 5.17 Käyttötapauskuvaus

Käyttötapauksen todennäköisimmät etenemispolut testataan syöttämällä järjestelmään rekisteröityneen asiakkaan oikeellinen käyttäjätunnus ja salasana. Sen lisäksi syötetään vääriä yhdistelmiä poikkeustilanteen käsittelyn testaamiseksi.

Virheen arvaus

Virheen arvaus (Error guessing) voidaan myös laskea testitapausten suunnittelutekniikaksi. Se tarkoittaa sitä, että testaajan kokemusta käytetään hyväksi määrittelemään ennalta, minkä tyyppisiä virheitä voi esiintyä, ja suunnittelemaan testejä, jotka paljastavat nämä virheet (ISTQB, 2007). Virheen arvaukseen on turvauduttava testauksen alueilla, joita ei kyetä käsittelemään muodollisemmilla testisuunnittelutekniikoilla kuten ekvivalenssiluokkien tai raja-arvojen avulla. Virheen arvauksessa voidaan käyttää apuna esimerkiksi aikaisemmin tehtyjen, formaalien testien tuloksia tai kohteen määrittelydokumentteja. Tyypillisiä tilanteita, joissa virheen arvaus voi olla tarpeen (Borysowich, 2007), ovat:

- Tiedon alustaminen, esimerkiksi sen tutkiminen, onko data asianmukaisesti poistettu.
- Vääräntyyppinen tieto, esim. negatiiviset luvut, ei-numeeriset syötteet jne.
- Oikean tietosisällön käsitteleminen, esimerkiksi testaaminen tiedolla, joka on luotu testattavassa järjestelmässä. Ohjelmoijien luomat testitietueet valitettavasti yleensä heijastavat sitä, mitä he olettavat järjestelmän tekevän, eivätkä pyri aiheuttamaan virhetilanteita.
- Virheiden hallinta, esimerkiksi asianmukainen virheiden priorisointi, selkeät virheilmoitukset, asianmukainen tiedon palauttaminen virhetilanteessa ja prosessin jatkuminen virheen jälkeen.
- Laskennat, esimerkiksi lasketaan käsin jokin tietuejoukko, ja verrataan palauttaako ohjelma saman tuloksen.
- Uudelleenkäynnistys ja toipuminen, esimerkiksi käytetään syötteitä, joka lopettaa ohjelman suorituksen ennenaikaisesti ja tutkitaan, toimiiko toipumisprosessi asianmukaisesti.
- Rinnakkaisten prosessien hallinta, esimerkiksi testataan tapahtumavetoista järjestelmää ajaen useampaa prosessia samanaikaisesti, ja tutkitaan, onko prosessien hallinta asianmukaista.

Edellä mainittujen tilanteiden lisäksi voidaan joutua tilanteeseen, jossa kohteendokumentointi on puutteellista, ja testaaja joutuu työskennellessään turvautumaan ainoastaan kokemukseensa ja ammattitaitoonsa. Esimerkki tilanteesta, jossa testauksen kohteen määrittely on tulkinnanvaraista, Laskutus-funktion määrittely:

”Tileille joiden velan saldo on > tai = 10 euroa ja joiden saldon kertymisestä on = tai > 30 päivää, pitää laskea myöhästymismaksu joka on isompi luvuista 3 euroa tai 1 %.”

Määritys ei selvästi kerro miten tulee menetellä jos velan saldo on > 10 euroa mutta se koostuu useammasta velasta jotka ovat kertyneet eri päivinä ja se

osuus jonka kertymisestä on = tai > 30 päivää on < 10 euroa. Virheenarvausta käyttäen suunnitellaan testitapaus tätä tilannetta varten:

- Velan kokonaissaldo on 13 euroa
- Se koostuu
    - 6 euron velasta jonka kertymisestä on < 30 päivää
    - 7 euron velasta jonka kertymisestä on > 30 päivää.

Voidaan todeta, että virheen arvaus on tekniikka, jonka käyttäjältä vaaditaan mielikuvitusta ja kokemusta testauksesta. Tekniikkaa käytetään tutkivassa testauksessa, josta enemmän luvussa 5.3.5.

5.3.4 Harmaalaatikkotestaus

Lasi- ja mustalaatikkotekniikoiden lisäksi puhutaan harmaalaatikkotestauksesta, joka yhdistää kahden edellä mainitun ominaisuuksia. Se on käyttökelpoinen testaustekniikka esimerkiksi testattaessa www-sovelluksia, sillä monimutkaista, useista erilaisista ohjelmisto- ja laitekomponenteista koostuvaa järjestelmää testatessa mustalaatikkotestaus ei paljasta tietovirta- tai raja-arvovirheitä, ja lasilaatikkotestaus ei löydä yhteensopivuusongelmia, aikaan liittyviä virheitä tai käytettävyysongelmia, vaan tekniikoita täytyy yhdistellä. (OAMK, 2006.)

Esimerkki, joka selvittää, miten järjestelmän toteutuksen yksityiskohtia voidaan käyttää hyväksi www-sovelluksen toimintaa testatessa:

Sovelluksen toiminnallisuus on yksinkertainen. Web-lomake sisältää kentän sähköpostiosoitteelle ja kenttiä, johon voi merkitä kiinnostuksen kohteita. Lomake täytetään ja tiedot lähetetään palvelimelle. Palvelin käsittelee tietoja niin, että se lähettää annettuun sähköpostiosoitteeseen artikkeleita sen perusteella, mitä kiinnostuksen kohteita lomakkeelle on syötetty. Sähköpostiosoitteen validointi hoidetaan käyttäjän koneessa JavaScriptillä.

Tässä tapauksessa, jos toteutuksen yksityiskohdat eivät ole tiedossa, lomaketta testataan oikeilla ja väärillä sähköpostiosoitteilla ja erilaisilla kiinnostuksen koh-

teiden valinnoilla ja todetaan, että sovellus toimii oikein, eikä lähetä postia vääränmuotoisiin osoitteisiin.

Jos taas toteutuksen yksityiskohdat ovat tiedossa, tiedetään, että järjestelmä tekee seuraavat olettamukset syystä, että osoite tarkastetaan jo ennen lomakkeen lähettämistä palvelimelle:

- Palvelin ei koskaan saa väärässä muodossa olevaa sähköpostiosoitetta
- Palvelin ei koskaan lähetä postia väärään sähköpostiosoitteeseen
- Palvelin ei koskaan vastaanota häiriöilmoitusta, jos syötteenä on väärä sähköpostiosoite

Tästä johtuen, osana harmaalaatikkotestausta, on testijoukko suoritettava myös JavaScript pois päältä kytkettynä. Tällaista voi tapahtua syystä tai toisesta, ja silloin sähköpostiosoitteen validointi ei onnistukaan käyttäjän koneella. Tässä tapauksessa järjestelmän oletukset ovat väärät, ja

- Palvelin saa vääränmuotoisen sähköpostiosoitteen
- Palvelin lähettää postia väärään sähköpostiosoitteeseen
- Palvelin vastaanottaa häiriöilmoituksen.

Esimerkkitilanteessa harmaalaatikkotestauksella pystytään osoittamaan, mitä väärän tiedon kulkeutuminen palvelimelle aiheuttaa.

5.3.5 Tutkiva testaus

Tutkiva testaus (Exploratory testing) on kokemusperäinen testaustekniikka, jossa käytetään ihmisen tietämystä ja kokemusta testitapausten määrittämiseksi. Tekniikka on erityisesti kontekstiohjatun koulukunnan suosiossa. Ketterissä menetelmissä tutkivaksi testaukseksi kutsutaan tekniikan lisäksi kokonaista testausprosessia, jossa prosessin rakenne perustuu tarpeeseen suunnata testausta sen mukaan, mitä testausprosessissa opitaan testattavasta kohteesta.

Yksinkertaisesti selitettynä tutkiva testaus tarkoittaa testien suunnittelemista ja suorittamista samalla kertaa, joka on vastakohtaista perinteiseen suunnitelmalliseen testaukseen (scripted testing) verrattuna, jossa testausproseduurit ja testitapaukset määritellään etukäteen, ennen testauksen suorittamista. Tutkiva testaus sekoitetaan toisinaan ”ad hoc” -testaukseen, joka tarkoittaa valmistelematonta testausta (ISTQB, 2007). Ad hoc -testauksessa testauksen tavoite ei ole selvillä, tuloksille ei ole odotuksia ja testaus sattumanvaraista.

Tutkivaa testaustekniikkaa käytettäessä testaussuunnitelma muuttuu sitä mukaa, kun testaaja oppii tietämään enemmän kohteestaan. Vaikka varsinaista lukkoon lyötyä testaussuunnitelmaa ei olekaan, on tiettyjä asioita, jotka vaikuttavat testaukseen. Tällaisia ovat esimerkiksi testausprojektin tarkoitus, testaajan rooli ja hänen taitonsa ja lahjakkuutensa, saatavilla olevat testaustyökalut, käytettävissä oleva aika ja testiaineisto, mahdollinen muilta ihmisiltä saatava apu, itse tuote ja sen käyttöliittymä, käyttäytyminen, testattavuus ja tarkoitus, se mitä testaaja tietää tuotteesta, se, mitä tapahtui edellisissä testeissä, tuotteen tiedossa olevat ongelmat, tuotteen käyttäjät ja heidän käyttäytymisensä, tieto siitä, kuinka tuotteen pitäisi toimia ja siitä, mitä erilaisuutta tai samankaltaisuutta on verrattuna muihin samanlaisiin tuotteisiin ja tärkeimpänä, mitä testaaja haluaa tietää tuotteesta. Tutkivan testauksen onnistuminen vaatii kurinalaisuutta ja kokeneita testaajia, muuten ei voi puhua testauksesta vaan suoritus jää ”kokeiluksi”. (Bach, 2003)

Tutkivassa testauksessa testaajan odotetaan oppivan koko testausprosessin ajan uusia asioita testauksen kohteesta. Uudet testit laaditaan sitä mukaa kun opitaan lisää tuotteesta, sen toiminnasta, riskeistä ja siitä, minkälaisia virheitä siitä on löytynyt edellisissä testeissä. Testausta voidaan tehdä testisessioissa, jotka kestävät 60 – 90 minuuttia kerrallaan. Testisessiolle hahmotellaan tavoite, mitä testataan, millä työkaluilla, mitä taktiikoita aiotaan käyttää, mitä riskejä on selvillä, minkälaisia virheitä on odotettavissa ja mitä tulisi saada selville. Näin saadaan päämäärä, mihin keskittyä testauksessa. Nämä tavoitteet kirjataan ylös, kuten myös testisession tuloksena saatava virheraportointi. Näitä voidaan käyttää hyväksi uusissa testisessioissa. (Kaner ym, 2002, s. 41, 177.)

Tutkiva testaus on hyödyllistä erityisesti monimutkaisissa testaustilanteissa, joissa testattavasta kohteesta tiedetään hyvin vähän. Tutkivalle testaukselle on sijansa aina, kun ei ole ilmiselvää, mitä seuraavaksi pitäisi testata ja silloin, kun halutaan päästä testauksessa syvemmälle kuin mitä ennalta suunnitellut testitapaukset kattavat. (Bach, 2001.)

6 TESTAUKSEN HALLINTA

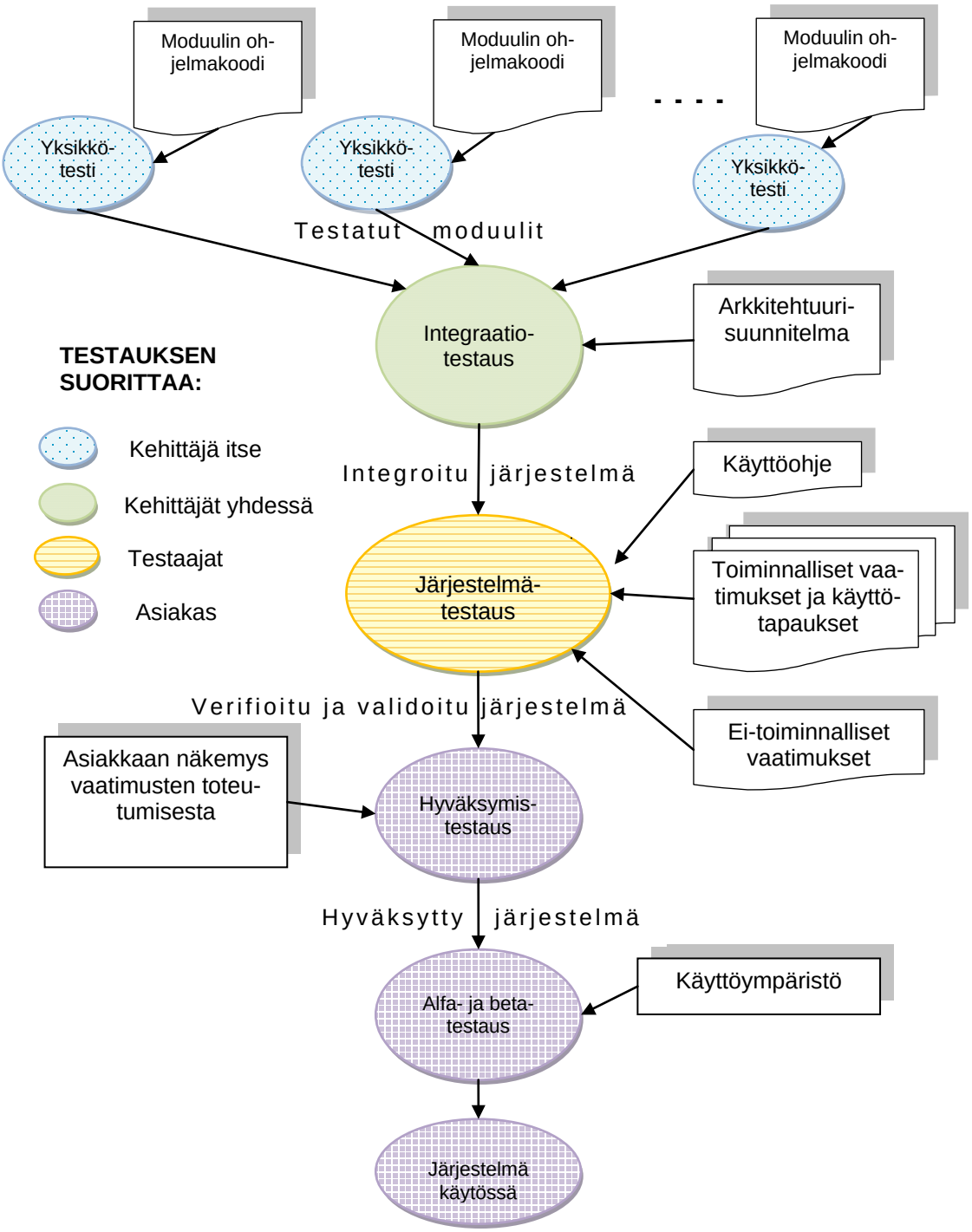
Tämä luku käsittelee testauksen hallintaa. Hallinnan tarkoituksena on pitää testauksen kustannukset ja ajankäyttö budjetin rajoissa ja mitata testauksen edistymistä ja sen tuloksia. Testauksen hallintaan kuuluu testimateriaalin hallinta, testauksen resurssien hallinta, testivaatimusten hallinta, testauksen suunnittelu, testitapausten suoritus ja suoritettujen testien seuranta sekä testauksen tuloksien raportointi ja seuranta.

6.1 Organisointi

Testauksen organisointi perustuu ohjelmistotuotantoa harjoittavan yrityksen tai organisaation toimintaprosesseihin. Jos organisaatiolla on laatukäsikirja, jossa ohjelmistotuotantoprosessin eteneminen on kuvattu, löytyy kuvauksista myös ohjeet testauksen organisointiin, sen työvälineet, sekä vaiheistus-, suoritus-, dokumentointi- ja raportointiohjeet. Laatukäsikirjat perustuvat yleisesti ISO 9001-stadardeihin.

Perusperiaate, joka tulee ottaa huomioon testausta suunnitellessa, on kriittisyys: suunnittelija ei koskaan saisi testata tuotoksiaan yksin, kuten ei myöskään suunnitteluorganisaatio, vaan aina pitäisi testata ristiin sekä yksilö- että ryhmätasolla. Näin toimien saadaan lisää näkökulmia testaukseen, sillä omaa tuotostaan on vaikea arvioida objektiivisesti. Testauksessa tulisi myös käyttää asiantuntijoita, jotka eivät ole vastuussa testattavan ohjelmiston suunnitteluratkaisuista tai toteutuksesta.

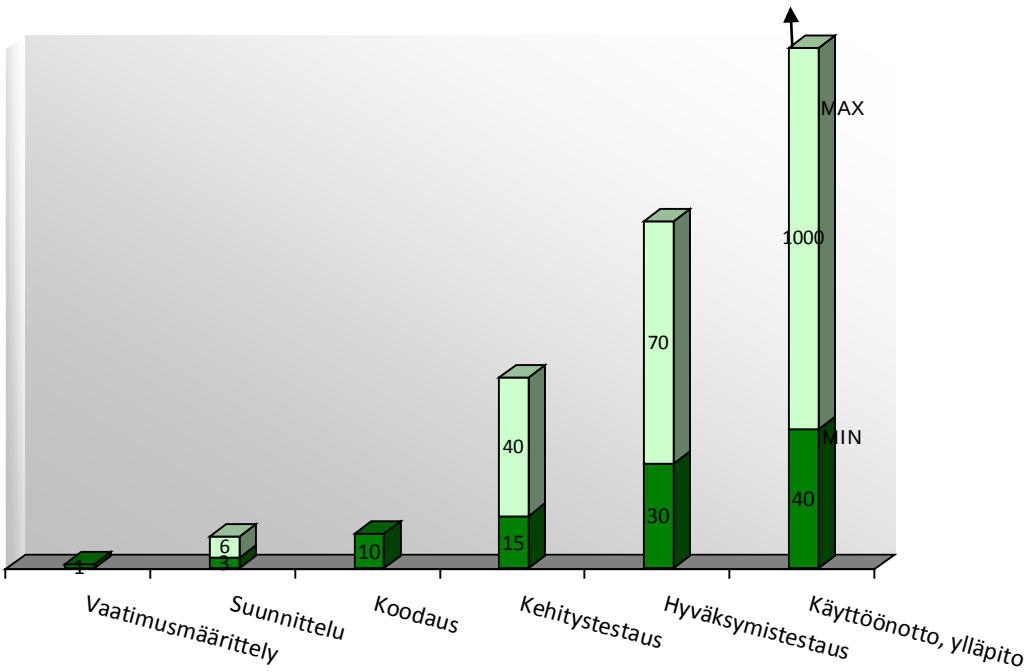
Tyypillisessä testausprosessin organisoinnissa yksikkötestauksen suorittaa toteuttaja itse, integraatiotestauksen moduulien toteuttajat yhdessä, järjestelmätestauksen varsinaiset testaajat, ja hyväksymistestauksen asiakas tai järjestelmän käyttäjät. Kuvassa 6.1, joka havainnollistaa testauksen aktiviteetteja, on merkitty testausvaiheiden toteuttajat.



Kuva 6.1 Testauksen vaiheet ja suorittajat

6.2 Suunnittelu

Testausta suunnitellessa on muistettava, mikä on testauksen määritelmä: sen on tarkoitus löytää virheitä. Tarkoituksena ei ole varmistaa, että kaikki toimii, vaan päinvastoin, ja tämä lähtökohta on pidettävä mielessä testitapauksia ja testauksen aikataulua suunnitellessa. Toinen tärkeä asia on aloittaa testauksen suunnittelu ajoissa: alustavat suunnitelmat järjestelmätestaukseen kannattaa tehdä jo järjestelmän määrittelyn valmistuttua, kunhan määrittely on katselmoitu ja hyväksytty, sillä testausta suunnitellessa voidaan havaita puutteita määrittelyssä jo ennen järjestelmän arkkitehtuurin suunnittelemisen aloittamista. Mitä aikaisemmassa vaiheessa virheet löydetään, sitä edullisempaa ja helpompaa niiden korjaaminen on. Barry W. Boehmin (1981) klassinen COCOMO-mallin (Constructive Cost Model) kehittämiseen liittyvä tutkimus, joka tehtiin IBM:n ohjelmistokehityksen kustannusten perusteella, totesi että toteutusvaiheessa löydetyn, vaatimuksiin liittyvän virheen korjaamisen kustannukset olivat kymmenkertaiset siihen verrattuna, että se olisi löydetty jo määrittelyvaiheessa, ja hyväksymistestausvaiheessa löydetyn virheen korjauskustannukset vähintään 30-kertaiset verrattuna määrittelyvaiheeseen. Kuvassa 6.2 esiintyvät tunnusluvut kuvaavat kertalukuina (kustannus minimissään ja maksimissaan) verrattuna määrittelyvaiheen korjauskustannuksiin, jossa kertaluku on 1:



Kuva 6.2 Vaatimuksiin liittyvän virheen suhteellinen korjauskustannus ohjelmistoprojektin eri vaiheissa (Boehm, 1981)

6.3Testaussuunnitelman laatiminen

Testaussuunnitelmaa laatiessa kannattaa ensin tehdä yleissuunnitelma, joka kertoo yleisen lähestymistavan testaukseen. Siinä tulisi kartoittaa käytössä olevat resurssit, mahdolliset järjestelmän riskit sekä päättää testauksen lopetuskriteereistä. Näiden pohjalta voidaan testaukselle muodostaa vaiheistus ja aikataulu. Testauksen suunnittelun perustana voi käyttää IEEE:n (1998) standardia 829. Standardin mukainen testaussuunnitelma käsittää 16 kohtaa, jotka esitellään myöhemmin.

6.3.1 Testauksen dokumentointi

Dokumentointi on suunniteltava ennen testausta, ei sen aikana tai sen jälkeen. Dokumentaation määrä riippuu testattavan ohjelmiston laajuudesta. Pienessä projektissa testauksen dokumentaatio voi sisältää vain yhden dokumentin, joka kattaa järjestelmä-, integraatio- ja moduulitestauksen, mutta suurissa projekteissa dokumentteja syntyy paljon. Testauksen dokumentteja ovat jo edellisessä kappaleessa mainitut testaussuunnitelma, joka on testauksen projektisuunni-

telma (master test plan), jonka sisältö on jo käsitelty, ja testisuunnitelma (test design specification), jossa kerrotaan jokaisen toiminnallisuuden osalta testauksen kohteet, testitapaukset ja läpäisykriteerit. Testitapauksen määrittely (test case specification) sisältää yhden testitapauksen yksityiskohtaisen kuvauksen: syötearvot, esiehdot, odotetut tulokset ja testisuorituksen jälkiehdot. Testiproseduurien määrittelyt kertovat, miten yksi tai useampi testitapaus toteutetaan tietyssä testiympäristössä, ja mitä testitapauksen aikana tapahtuu. Näiden lisäksi voidaan dokumentoida testilokeja sekä kirjoittaa havaintoraportteja, joissa kuvataan mikä tahansa testauksen aikana sattunut tapahtuma, jota aiotaan tutkia tarkemmin. Kun testaus on saatu päätökseen, laaditaan testauksen yhteenvetoraportti, johon kootaan selvitys suoritetuista testeistä ja niiden tuloksista. Yhteenvedossa arvioidaan testauskohteet ja verrataan testien tuloksia hyväksymiskriteereihin. Suositukset testauksen dokumenttien sisällöstä löytyvät IEEE:n standardista 829.(ISTQB, 2007, IEEE 1998.)

On kuitenkin muistettava, että käytössä on muitakin tapoja raportoida kuin standardilähtöinen, esimerkiksi ketterissä menetelmissä pyritään välttämään ylimääräistä raportointia. Asiasta kerrotaan ketterää testausta käsittelevässä luvussa 7.

6.3.2 Testauksen dokumentit standardin 829 mukaan

Tässä luvussa esitellään IEEE:n standardin 829 mukaisen testausdokumentaation sisältö.

Testaussuunnitelman tunniste

Tunniste, joka identifioi testaussuunnitelman.

Johdanto

Johdanto sisältää yhteenvedon testauksen kohteena olevista ohjelmiston osista ja ominaisuuksista. Tässä kappaleessa esitellään myös jokaisen testauksen kohteen tarkoitus ja historia. Korkean tason testaussuunnitelmissa johdanto-kappaleessa voidaan viitata seuraaviin dokumentteihin:

- projektin auktorisointi
- projektisuunnitelma
- laatukäsikirja
- kokoonpanon hallintasuunnitelma
- käytettävät toimintasuunnitelmat
- käytettävät standardit.

Johdantokappaleen tarkoituksena on antaa yleiskuva testaussuunnitelmasta.

Testauksen kohteet

Testauksen kohteet tulee määritellä versiotietoineen. Jos testiaineistolle on suoritettava muunnoksia ennen kuin testaus voi alkaa, kerrotaan siitä tässä, ja jos seuraavia dokumentteja on olemassa, viitataan niihin:

- määrittelydokumentit
- suunnitteludokumentit
- käyttöohjeet
- asennusohjeet.

Testauksen kohteisiin liittyviin vikaraportteihin on myös viitattava jos sellaisia on, ja määritellä kohteet, jotka erityisesti aiotaan jättää testauksen ulkopuolelle.

Testattavat ominaisuudet

Määritellään kaikki testattavat ominaisuudet ja ominaisuuksien yhdistelmät, jotka on tarkoitus testata.

Ominaisuudet, joita ei testata

Määritellään kaikki testattavat ominaisuudet ja ominaisuuksien yhdistelmät, joita ei aiota testata, ja kerrotaan syyt, miksi ei.

Testausstrategia

Kuvaillaan yleinen lähestymistapa testaukseen. Kerrotaan, millä tavoin huolehditaan siitä, että tärkeimmät ominaisuudet tai niiden yhdistelmät tulevat testattua. Määritellään tekniikat ja työkalut joita käytetään edellä mainittujen ominaisuuksien tai ominaisuusyhdistelmien testauksessa. Strategia pitää kuvailla riittävän yksityiskohtaisesti, jotta se mahdollistaa tärkeimpien testaustehtävien havaitsemisen ja havaittujen tehtävien arvioidun keston määrittämisen. Kuvaillaan testauksen kattavuuden vähimmäisvaatimukset, ja tekniikat, joita käytetään testauksen kattavuuden arvioimisessa. Määritellään muut käytettävät lopetusehdot, kuten virhetaajuus. Jos on käytössä vaatimustenjäljitystekniikoita, niistä kerrotaan tässä. Luetellaan huomattavat rajoitteet, kuten testauksen kohteiden saatavuus, testausresurssien saatavuudet ja takarajat testauksen valmistumiselle.

Testien läpäisy- ja epäonnistumiskriteerit

Kerrotaan kriteerit, joita käytetään sen määrittämisessä, onko testauksen kohde läpäissyt testin vai ei.

Testauksen keskeyttämiskriteerit ja vaatimukset testauksen jatkamiselle

Määritellään kriteerit, joiden perusteella keskeytetään kyseessä olevan testisuunnitelman toimintoja. Määritellään testaustoiminnot, jotka pitää toistaa, kun testausta jatketaan.

Laadittavat dokumentit

Määritellään dokumentit, joita testauksen aikana on tarkoitus tuottaa. Näitä ovat

- testaussuunnitelma
- testisuunnitelma
- testitapausten määrittelyt
- testausproseduurien määrittelyt
- julkaisuselosteet
- testien lokitiedostot
- havaintoraportit
- testauksen yhteenvetoraportti.

Edellä mainittujen lisäksi testien syötteet ja tulosteet tulee dokumentoida. Testien suoritustyökalut, kuten ajurit ja tynkämoduulit lasketaan myös testausprojektin tuotoksiin.

Testaustehtävät

Luetellaan tehtävät, jotka ovat välttämättömiä testauksen suorittamiseksi, ja kerrotaan erityistaidoista, joita tarvitaan.

Ympäristövaatimukset

Luetellaan sekä testausympäristön välttämättömät että toivotut ominaisuudet, kuten laitteisto, tietoliikenneyhteydet, käyttöjärjestelmät ja muut ohjelmistot, joita testauksen toteutus vaatii. Määritellään tietoturvan taso koskien yllä olevia. Määritellään testityökalut, joita tarvitaan, sekä muut tarvittavat asiat, kuten kirjalliset julkaisut tai työtila.

Vastuut

Luetellaan vastuuryhmät testauksen johtamiselle, suunnittelulle, valmistelulle, suorittamiselle, todistamiselle, tarkistamiselle ja päättämiselle. Nämä vastuuryhmät voivat sisältää esimerkiksi kehittäjiä, testaajia, käyttäjiä, teknisen tuen henkilökuntaa, ylläpitäjiä ja laadunvarmistajia

Henkilöstönhallinnan ja koulutuksen tarpeet

Määritellään tarvittavan testaushenkilöstön ammattitaidon taso ja mahdollisuudet koulutukseen välttämättömien taitojen hankkimiseksi.

Aikataulut

Aikatauluun sisällytetään kaikki ohjelmistoprojektin aikataulun virstanpylväät kuten osaluovutusten ajankohdat. Jos testaus tarvitsee omia virstanpylväitään, lisätään ne. Arvioidaan aika, joka kuluu testaustehtävien suorittamiseen, aikataulu jokaiselle testaustehtävälle ja päivämäärä jokaiselle virstanpylväälle. Määritellään testausresursseille, kuten työkaluille tai henkilökunnalle, ajat, jolloin niiden tulisi olla käytettävissä.

Riskit ja ylimääräiset menot

Määritellään testisuunnitelman riskit ja suunnitelmat ylimääräisten menojen varalle, kuten esimerkiksi viivästymien aiheuttamien ylitöiden järjestämiseksi.

Hyväksymiset

Luetellaan henkilöt, joilla suunnitelma hyväksytetään hyväksymispäivämäärineen ja allekirjoituksineen.

6.4 Havaintojen raportointi

Kun testausstrategiat, suunnitelmat ja lähtökohdat on selvitetty ja testaus on käynnissä, alkaa syntyä havaintoja testattavasta tuotteesta. Havainnoista raportoidessa tulee muistaa, ettei testauksen tarkoitus ole ratkaista ongelmia, vaan ainoastaan löytää ne. Havainto- tai virheraportoinnissa tärkeää on nopeus: vaikka kohteen testaus olisi kesken, kannattaa virheistä raportoida heti, sillä se nopeuttaa koko ohjelmistoprosessia. Raportoinnissa on tärkeää myös sen selkeys: raportoidaan ainoastaan tarpeen olevat tosiasiat ja yksityiskohdat, kerro-

taan mitä tarkalleen tapahtui, ja jos aiheellista, otetaan kuvankaappaus ongelmatilanteesta. Havainnon toistettavuus selvitetään selittämällä askel askeleelta, miten ongelma saatiin näkymään. Raporttiin kuuluu myös testaajan käsitys siitä, mikä on ongelman vakavuus. Kirjassa Lessons Learned in Software Testing (Kaner ym, 2002, 85) annetaan neuvoja, kuinka kirjoittaa virheraportti:

- Käy virhetilanne läpi askel askeleelta.
- Numeroi jokainen askel.
- Älä ohita yhtään askelta, jota tarvitaan ongelman selvittämiseksi.
- Listaa lyhin mahdollinen tie virheeseen.
- Asettele raportti väljästi ryhmitellen, että se olisi helppolukuisempi.
- Käytä lyhyitä, yksinkertaisia lauseita.
- Selitä, mitä tapahtui, ja mitä oletit tapahtuvaksi.
- Jos vaikutukset ovat vakavat mutta sinulla on syy, miksi epäilet ohjelmoijan epäilevän ongelman vakavuutta, niin selitä kantasi.
- Sisällytä raporttiin ylimääräisiä kommentteja jos ne helpottavat joko virheen havaitsemista tai sen uusintatestausta ongelman korjaamisen jälkeen.
- Jos kyseessä on monimutkainen ongelma, kirjoita kolmelle ensimmäiselle riville yhteenveto, ja vasta sen jälkeen yksityiskohdat.
- Pidä ulosantisi neutraalina.
- Älä yritä vitsailla, vitsit ymmärretään väärin.

Kyky raportoida virheistä ymmärrettävästi on ensiarvoisen tärkeä taito testaajalle, sillä testauksen tarkoitus on saada lisää informaatiota tuotteesta, sen laadusta ja sen tilasta. Jos raportointi ei onnistu, ei informaatio siirry, ja testaajan työ on ollut turhaa. Myös virheiden vakavuuden arvioinnissa testaajan pitäisi osata olla objektiivinen. Jos testaajan mielestä virheluokitukseltaan alhainen ongelma vaatisi korkeaa korjausprioriteettia, tulee testaajan selittää syy, miksi, eikä yliarvioida virhettä vakavammaksi saadakseen sille huomiota, sillä väärät arviot murentavat testaajan uskottavuutta. Asialliset ja informatiiviset raportit ovat se tapa, jolla projektin johdolle saadaan realistinen kuva ongelmasta. Näin se kyke-

nee tekemään hyviä päätöksiä sen suhteen, mitä virheitä korjata ja millä prioriteetilla.(Kaner ym, 2002, s. 83.)

6.5 Testauksen päättäminen

Ei ole helppoa päättää, milloin on testattu tarpeeksi. Kaikkea ei voi koskaan testata, vaan aina on tyydyttävä vaillinaiseen lopputulokseen. Yksinkertainenkin järjestelmä on liian monimutkainen siihen, että laskennallinen aika riittäisi kaiken kattavaan testaukseen. Kattava testaus tarkoittaisi, että tulisi testata jokaisen muuttujan jokainen mahdollinen syöte, jokainen mahdollinen yhdistelmä syötteitä ja muuttujia, jokainen mahdollinen toimintoketju ohjelman läpi, jokainen mahdollinen laitteisto- ja ohjelmistokonfiguraatio sekä testata jokainen tapa, jolla käyttäjä saattaisi yrittää käyttää sovellusta. Tehtävä on mahdoton, joten testausta suunnitellessa on mietittävä, miten testata ja mitä testata niin, että käytössä olevassa ajassa, niillä resursseilla jotka ovat käytettävissä, löydettäisiin mahdollisimman suuri osa vakavista virheistä. Kun sitten testataan, joudutaan miettimään, onko tavoite jo saavutettu.

Testaussuunnitelmassa kerrotaan erilaisia päätöskriteereitä sille, milloin testaus on valmis. Kriteerit ovat usein tilastollisia: testauksen tulee saavuttaa tiettyjä kattavuusprosentteja, että se todettaisiin riittäväksi. Kattavuudet selitettiin luvussa 5.3.1. Voidaan myös harjoittaa virheen kylvämistä (error seeding), jossa testattavaan ohjelmaan kylvetään tahallisia virheitä, jotta saataisiin aikaan virheiden löytymistä kuvaava mittari jäljellä olevien vikojen määrän arvioimiseksi (ISTQB, 2007). Menetelmä on seuraavanlainen:
Kylvetyt virheet = Y,
Oikeat virheet = X
Löydetyt kylvetyt virheet = y
Löydetyt oikeat virheet = x

Voidaan olettaa, että virheistä löytyy y/Y

Ohjelman alkuperäisten virheiden määrä: X = x * (Y / y)
Virheet, joita ei vielä ole löydetty: X - x

Menetelmä on ongelmallinen siksi, että kylvetyt virheet voivat olla niin erilaisia kuin oikeat, että ne eivät löydy samoilla testaustekniikoilla. Se voi olla jopa vahingollinen, jos kylvettyjä virheitä unohtuu koodiin (Watkins, 2001). Erilaisten tilastollisten päätöskriteerien ongelmana on myös se, että on vaikea arvioida etukäteen, paljoko itse asiassa joudutaan tekemään testejä jonkin kattavuus- tai virhetaajuusarvon saavuttamiseksi. Monesti testaus joudutaan lopettamaan ”kesken”, vaikka kriteerit eivät olekaan täysin täyttyneet.

Toinen tapa lähestyä ongelmaa on ajatella, että koska testaus on tiedonkeräysprosessi, voidaan prosessi lopettaa, kun on saatu kerätyksi riittävästi tietoa. Riittävästi on sen verran, että asiakas pystyy tekemään hyviä päätöksiä testauksen tulosten perusteella. Tässäkään lähestymistavassa ei voida ajatella että lopetetaan, kun kaikki viat on löydetty, koska kukaan ei voi tietää, milloin tämä on tapahtunut. On ajateltava, että lopetetaan sitten, kun on syytä uskoa, että mahdollisuus vakavien virheiden löytymisestä ohjelmistosta on alhainen. Tämä usko voidaan perustaa muun muassa seuraaviin seikkoihin:

- Testaaja on ollut tietoinen siitä, minkälaisia ongelmia on tärkeää löytää, jos niitä esiintyy.
- Testaaja on ollut tietoinen siitä, kuinka havaita ohjelmiston eri osissa se, esiintyykö siellä ongelmia, jotka ovat tärkeitä löytää.
- Testaaja on tutkinut ohjelmiston tarkasti edellä mainitut kohdat huomioon ottaen.
- Testausstrategia on ollut tarkoituksenmukainen ja monipuolinen.
- On käytetty kaikki resurssit joita testauksen käytössä on ollut.
- On toimittu kaikkien niiden testausprosessin standardien mukaan, joiden mukaan asiakas on olettanut testaajan toimivan.
- Testausstrategia, testauksen tulokset ja laadun arviointi on kuvattu mahdollisimman selkeästi asiakkaalle.

Jos yllä luetellut vaatimukset täyttyvät, ja kuitenkin testattavasta ohjelmistosta löytyy vakava ongelma sen julkaisemisen jälkeen, syyt löytyvät todennäköisesti seuraavasta joukosta: joko testaaja ymmärsi riskit huonommin kuin kuvitteli tai

testauksessa tapahtui virhe. Mahdollista on myös, että testaaja arvioi riskit oikein, mutta projektin johto päätti ottaa riskin. (Kaner, ym, 2002, 171)

7 KETTERÄT MENETELMÄT JA TESTAUS

Tässä luvussa kerrotaan ketterien menetelmien perusperiaatteet ja se, kuinka ketteriä periaatteita toteutetaan ohjelmistotestauksessa.

7.1 Ketterät periaatteet ja menetelmät

Ketterissä menetelmissä ohjelmisto kehitetään osissa, jotka toimitetaan asiakkaalle aina, kun osa eli iteraatio on valmis. Ohjelmiston toteutusta ei suunnitella tarkasti, vaan muutoksia odotetaan tapahtuvaksi sen määrittelyissä ja muutostarpeita tarkastellaan säännöllisesti ja usein. Ketterässä kehityksessä pyritään työ tekemään yhdessä: liiketoiminnan ja teknisen puolen edustajat tekevät yhteistyötä päivittäin ymmärtääkseen ja saavuttaakseen projektin tavoitteet, jotka ovat yhteisiä kummallekin. Ketterä kehitys on tiimityötä, jossa tiimit pyritään luomaan niin, että ne ovat toimivia ja motivoituneita. Työtahti pidetään tasaisena eikä ylitöitä suosita, koska ylirasittunut henkilöstö ei toimi tehokkaasti. Tiimit työskentelevät samassa työtilassa, jolloin informaatio kulkee sujuvasti. Ketterissä menetelmissä pyritään yksinkertaisimpaan mahdolliseen ratkaisuun tinkimättä teknisestä tasosta, ja itseohjautuva tiimi osaa suunnitella itse arkkitehtuurin, selvittää vaatimukset syvällisesti ja löytävät parhaat tekniset ratkaisut. Nämä ratkaisut voivat tosin muuttua matkalla, jos on tarpeen. Toimintaa parannetaan jatkuvasti, oppien siitä, mitä lähimenneisyydessä on tapahtunut. Ketterän kehityksen ominaispiirre on, että kaikki ohjelmistokehityksen aktiviteetit vaatimusmäärittelystä testaukseen ja toimitukseen toteutetaan yhden iteraation puitteissa. (Pyhäjärvi, Pöyhönen, 2009.)

Ketteriä menetelmiä ovat esimerkiksi XP ja Scrum. XP, eli eXtreme Programming, on ohjelmointikeskeinen menetelmä, joka muodostuu joukosta käytäntöjä, joita kaikkia tulee noudattaa yhdessä. Käytännöt on rakennettu tasapainottamaan toisiaan, ja niistä poikkeaminen saattaa vesittää koko käytännön. Toinen yleisesti käytössä oleva menetelmä on Scrum, joka on sovelluskehityksen malli, joka kertoo, kuinka ohjata projektia. Se ei ota kantaa matalamman tason ratkaisuihin, vaan keskittyy nimenomaan projektin vaiheistamiseen ja sen kontrollointiin. Scrum, kuten muutkin ketterät mallit, katsoo ohjelmistokehityksen koostuvan erimittaisista sykleistä, joista tärkeimmät ovat sprintti eli pyrähdys, sekä

päivä. Pyrähdyksellä tarkoitetaan kehitysjaksoa, iteraatiota, jonka jälkeen ohjelmisto on valmis julkaistavasti. Sprintin tyypillinen kesto on yksi kuukausi, mutta se voi vaihdella viikosta kahteen kuukauteen organisaation mukaan. Päivään taas liittyy Scrum-palaveri. Palaverissa Scrum-mestarin johdolla kerrataan, mitä on tehty, mitä aiotaan tehdä ja mitkä ongelmat hidastavat kehitystä. Scrum-mestarin on pyrittävä poistamaan hidastavat ongelmat ja valvoa sitä, että Scrumia noudatetaan oikein. (Poimala, Heikniemi, Blåfield, 2008.)

7.2 Ketteryys testauksessa

Ketterän testauksen määritelmä on, että se on mitä tahansa testausta, joka on sidottu ketteriin arvoihin. Se on käytäntö, joka kohtelee kehitystä testauksen asiakkaana tai käytäntö projekteille, joissa käytetään ketteriä menetelmiä. Ketterän testauksen menetelmiä ovat muun muassa ns. ääritestaus, tutkiva testaus ja pariohjelmointi. Ääritestaus on menetelmä, joka perustuu XP- menetelmään, jossa yksikkötestaus suoritetaan TDD:n avulla, ohjelmointia korostamalla. Ohjelmistoa koostetaan jatkuvalla integraatiolla päivittäisine regressiotesteineen, joten iteraatio on periaatteessa milloin tahansa julkaistavissa. Hyväksymistestauksen tekee asiakas, mutta myöhemmin testaaja tai tiimi toimii asiakkaan sihteerinä testien laatimisen osalta. Tutkivassa testauksessa, joka on selitetty luvussa 5.3.4, pyritään löytämään uutta tietoa testattavasta ohjelmistosta, sen virheistä tai mistä tahansa, millä on merkitystä. Pariohjelmointi on menetelmä, jossa kaksi ohjelmoijaa työskentelee samalla koneella, toinen ohjelmoi ja toinen etsii virheitä. Menetelmä parantaa koodin laatua: vaikka ohjelmointinopeus ei juuri kasva, on lopputulos huomattavasti virheettömämpää kuin yksin ohjelmoidessa.

Testaus voi olla ketteriin arvoihin perustuvaa, vaikka ohjelmistoa ei kehitettäisi ketterästi. Ketterät arvot, joissa arvostetaan vuorovaikutusta ja yksilöllisyyttä prosessien ja työkalujen sijaan, kehitettävän järjestelmän toimivuutta perusteellisen dokumentoinnin sijaan, yhteistyötä asiakkaan kanssa sopimusneuvottelujen sijaan ja muutokseen vastaaminen suunnitelmien seuraamisen sijaan, voivat tosin olla vaikeita yhdistää hyvin muodolliseen ohjelmistoprojektiin, jossa

dokumentaatiolla ja suunnitelmissa pysymisellä on suuri painoarvo, mutta toisaalta, testaukselle on yleensä varattu projekteissa liian vähän aikaa sen vaatimaan työmäärään verrattuna ja siitä huolimatta olisi testauksen pysyttävä joustavana ja vastaamaan muutokseen. Käytössä olevasta strategiasta ja toimintatavasta huolimatta, koska tahansa testauksen loppuessa, olisi tarkoitus että on tehty parasta mahdollista testausta käytetyn työmäärän puitteissa. Tämä tavoite tulisi pitää mielessä testausta suunnitellessa. (Pyhäjärvi ym, 2009.)

7.3 Ketteryyden vahvuudet ja ongelmat

Ketterällä testauksella on etunsa verrattuna perinteisiin testausmenetelmiin. Esimerkiksi perinteisissä keskitytään kattavuusprosentteihin, jotka muuttuvat helposti 100 % kattavuudesta 10 % kattavuudeksi katsantokantaa muuttamalla. Myös jäljitettävyydellä on kritiikkinsä: linkkien luominen ja ylläpito on aikaa vievää ja jäykkää. Ketterissä menetelmissä tiimien tiivis yhteistyö tekee jäljitettävyysvaatimukset turhaksi: tiedonvälitys toimii, kun tehdään töitä yhdessä. Perinteisen testauksen raskaasti dokumentoitu vaatimusmäärittely johtaa positiivisen testauksen ylikorostumiseen seuraavista syistä: testaaja testaa vain sitä mikä on mainittu, eikä mielikuvitusta ei vaadita. Testaus keskittyy toiminnallisuuksiin unohtaen tai rajaten pois ei-toiminnalliset ominaisuudet ja testaaja keskittyy vain omaan työalueeseensa unohtaen sen, mikä olisi kokonaisuuden kannalta paras. Dokumentoinnin paljouden vuoksi perinteiset menetelmät ovat usein kalliita ja tehottomia, kun korostetaan askelten toistettavuutta tavoitteiden sijaan. Perinteisessä testauksessa valmiusaste usein lasketaan työmääristä: se, että tunnit on tehty, ei todellisuudessa tarkoita sitä, että työ on valmis. Itseohjautuvuuden ja yhteistyön puute aiheuttaa viivästymiä, kun odotellaan päätöksiä, joista ei tarvitsisi tehdä hidasteita, ja keskustelun sijaan oletetaan, että tiedetään, mitä tahdotaan, kun vaatimukset on kerran kirjoitettu ja luettu. (Pyhäjärvi ym. 2009.)

Ketterässä testauksessa on myös ongelmansa: testitapausten suunnittelu voi olla vaikeaa jos kirjallista dokumentaatiota ei ole olemassa. Testaus voi jäädä jälkeen iteraatiosta, ja välillä testaus tuntuu tuhlaavan resursseja, kun testataan piirteitä, jotka muuttuvat jatkuvasti. Jos tiimityö ei toimi ja tuotanto ei muista in-

formoida muutoksista, on testaaja vaikeuksissa. Tuote muuttuu projektin edetessä, prioriteetit muuttuvat ja v-malli ei toimi testauksen ohjenuorana. Testejä on tehtävä koko ajan ja pitää mielessä testin kohden ja tavoite. Ketterissä menetelmissä testien automatisointi on tärkeää, jotta jatkuva regressiotestaus olisi edes mahdollista. Testaus on ketterissä menetelmissä integroitava selkeästi ohjelmistotyön eri vaiheisiin ja testaaja otettava mukaan kehitystiimin osaksi. Testaajan on osallistuttava tiimipalavereihin, jotta kommunikointi onnistuu. Näin testaajan on mahdollista pyytää epäselvyyksiin tarkennuksia ja jakaa omia tietojaan ja taitojaan tiimin käyttöön. Testaus voi auttaa ideoinnissakin, sillä testaaja usein näkee parhaiten, missä kunnossa projekti on. Testaajan ottaminen mukaan tiimiin voi parantaa testauksen ja laadun lisäksi kehitettävän ohjelmiston rakennetta.

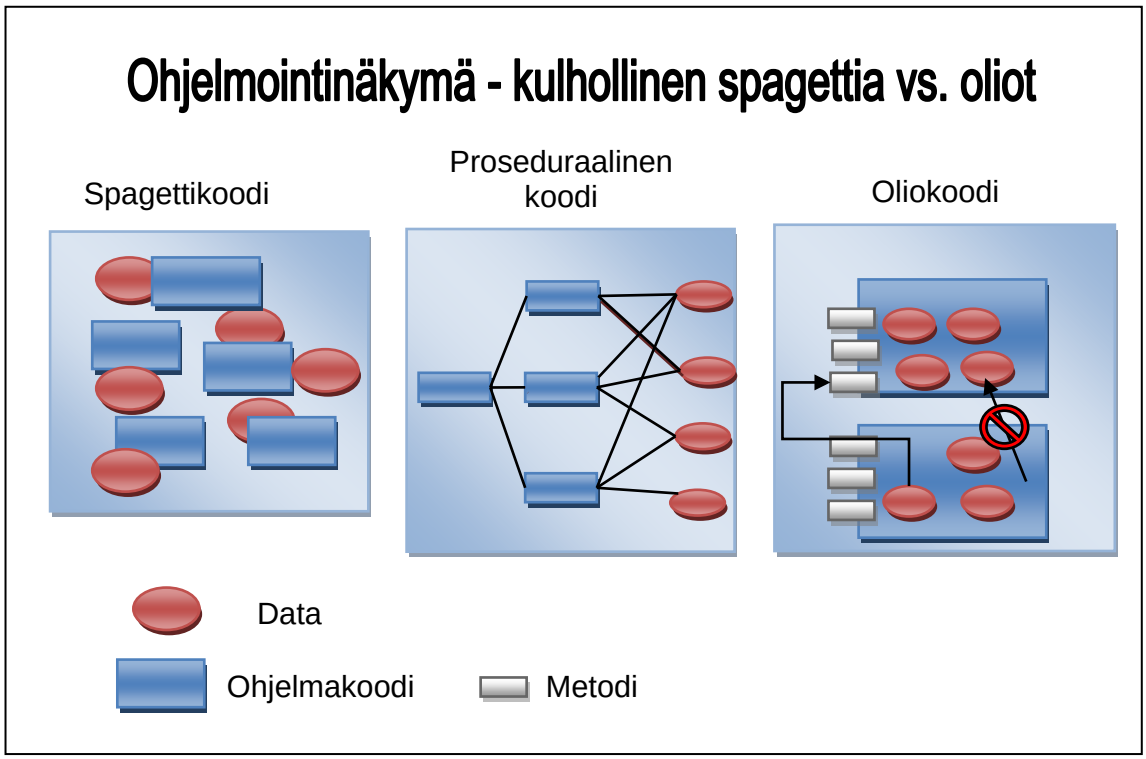
Ketterät menetelmät eivät ole ihmeaine, jolla saadaan koodista laadukasta ja asiakkaan toiveiden mukaista, aikaa säästymään ja budjetti kestämään rajoissaan. Ne voivat auttaa näissä asioissa, mutta vakiintuneita työtapoja ei tule muuttaa liikaa kerralla. Jos vanhassa prosessissa on toimiviksi koettuja käytäntöjä, ne on syytä säilyttää. Työntekijöiden muutosvastarinta hankaloittaa usein uusien menetelmien käyttöönottoa, ja ketteriäkin menetelmiä voi kyseenalaistaa. Se mikä yhdessä projektissa toimii, ei välttämättä toimi toisessa, ja ketterän menetelmän käyttämiselle on oltava perusteet.

8 OLIOTESTAUS

Tässä luvussa käsitellään olioperustaisten ohjelmien testaamista ja sitä, miksi se poikkeaa proseduraalisten, eli lausekielisten ohjelmien testaamisesta.

8.1 Ohjelmistojen rakenteen kehitys

Ohjelmistojen rakenne on käynyt läpi kuvan 8.1 mukaisen kehityskaaren: aivan ohjelmoinnin historian alussa ohjelmakoodi muistutti rakenteeltaan spagettikulhollista: tieto ja ohjelmakoodi sijaitsivat toistensa seassa, eivätkä käytetyt ohjelmointikielet tukeneet millään tavoin ohjelmien rakenteellista selkeyttä. Hyppykäskyt aiheuttivat saman, mitä spagetin haarukoiminenkin: yhdellä puolella koodia tapahtuva käsky aiheutti liikehdintää jossain aivan muualla. Koodia oli vaikea lukea ja vaikea ylläpitää.



Kuva 8.1 Ohjelmistojen rakenteen kehittyminen (Pöyhönen ym. 2002)

Ohjelmistojen koon kasvaessa todettiin, että edellä mainitun kaltainen koodi oli aivan liian vaikeasti ylläpidettävää, ja ohjelmien rakennetta alettiin selkeyttää. Ohjelmakoodi muodostui nyt pääohjelmasta joka kutsui aliohjelmia hierarkkisessa järjestyksessä, ja tietorakenteet erotettiin ohjelmakoodista. Tämänkaltaista koodia kutsutaan rakenteiseksi tai proseduraaliseksi ohjelmakoodiksi. Nyt koodi oli huomattavasti selkeämpää, mutta tämän kaltainen rakenne oli edelleen vaikeasti ylläpidettävä siksi, että pienikin muutos tietorakenteissa tai aliohjelmissa tarkoitti yleensä laajoja muutoksia koko ohjelmistoon. Lisäksi ohjelmistot olivat edelleen muuttuneet laajemmiksi ja monimutkaisemmiksi, ja tästä syystä kaivattiin tapoja käyttää uudelleen jo kerran kirjoitettua ohjelmakoodia.

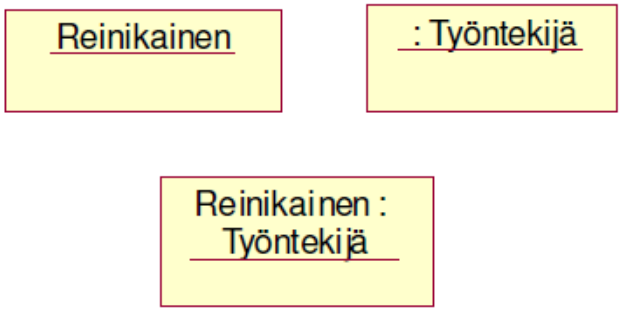
Edellä mainituista syistä alettiin kehittää olioperustaista ohjelmointirakennetta: tässä mallissa ohjelmat koostuvat moduuleista, joilla on oma tehtävänsä ja omat tietonsa. Nämä moduulit ovat olioita: yksiköitä, joilla on tietyt ominaisuudet ja tietty käyttäytyminen. Ohjelmaa ei ajeta pääohjelmalla, joka kutsuu aliohjelmia, vaan ohjelman suoritus etenee olioiden vuorovaikutuksena niiden käyttäessä hyväkseen toistensa tarjoamia palveluita. Tämänkaltaisen ohjelmointirakenteen edut ovat, että moduulit ovat uudelleenkäytettävissä toisessa sovelluksessa, sekä se, että tarpeen vaatiessa moduuli voidaan vaihtaa toisella ilman, että muutoksia tarvitsee tehdä muualle ohjelmakoodiin. Olioperustaisuus on kehittänyt ohjelmistotuotantoa yksittäisten sovellusten laatimisesta kohti sovellusabstraktioita, joita ovat ohjelmistoalustat ja sovellus- tai tuoteperheet sekä sovelluskehykset. (Jaakkola, 2006.)

On kuitenkin todettava, että vanhempia ohjelmointirakenteita ei ole hylätty, vaan niillä on edelleen paikkansa. Spagettikoodia käytetään laiteläheisessä ohjelmoinnissa, ja proseduraalista koodia käytetään paikoissa, joissa oliototeutus olisi liian raskas käyttää, kuten teknis-tieteellisissä laskentasovelluksissa ja systeemiohjelmissa, eli ohjelmissa, jotka ohjaavat tietokoneen omaa toimintaa.

8.2 Olio-ohjelmoinnin käsitteitä

Olio

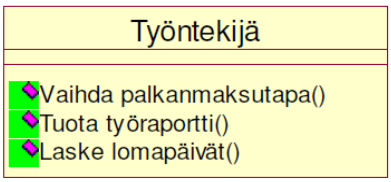
Olio (object) on olio-ohjelmoinnin peruskäsite, ja se sisältää tiedon ja siihen kohdistuvat toiminnot eli metodit. Olio kapseloi omat tietonsa, jotta niihin ei pääsisi käsiksi suoraan vaan ainoastaan metodien kautta. Rakenteiseen ohjelmointiin verrattuna olio-ohjelmoinnin perusyksikkö, olio, poikkeaa siinä, ettei se ole puhtaasti toiminnallinen, kuten aliohjelma, eikä puhtaasti tietoa säilyttävä, kuten tietue, vaan oliossa nämä yhdistyvät (Koskimies, 1998). Kuvassa 8.2 esitetään UML-mallina työntekijä-luokan olio Reinikainen:



Kuva 8.2 Luokka ja olio

Metodi

Metodit (method, memeber function), joita myös operaatioiksi kutsutaan, ovat toimintoja, joilla käsitellään olion sisältämää tietoa. Olio voi suorittaa toiminnot, jotka sille on määrätty, ei muuta. Metodille määritellään aina nimi ja ohjelmoidaan toiminta, ja niille voidaan määritellä myös sellaisia parametreja, jotka välittävät metodille tietoa olion ulkopuolelta. Kuvassa 8.3 esitetään Työntekijä-olion metodit:



Kuva 8.3 Metodit

Tietojäsen

Olion tietojäsenet eli attribuutit (attribute) ovat nimettyjä, tavallisesti ulkopuolisilta olioilta suojattuja tietokenttiä, joihin olion sisältämä tieto talletetaan. Tietojäseniä voidaan kutsua myös olion ominaisuuksiksi. Kuva 8.4 esittää Työntekijäluokan tietojäsenet vasemmalla ja Työntekijä-olion tilan oikealla. Tietojäseniä käsitellään metodien avulla, ja olion tila muodostuu kaikkien olion tietojäsenien tilojen yhdistelmästä.

Kuva 8.4 Tietojäsenet

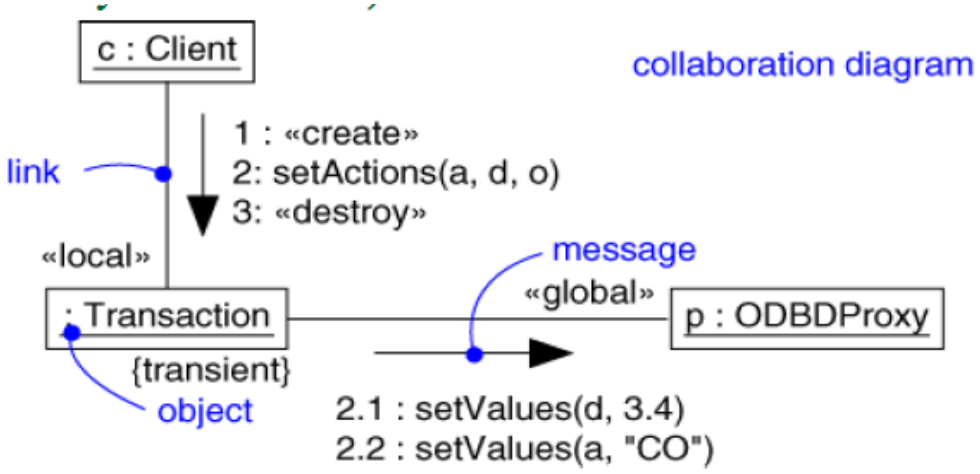
Rajapinta

Olioilla on julkinen rajapinta, jonka kautta olion tarjoamia palveluita voi käyttää. Olion rajapinta kertoo, mitä parametreja se tarvitsee pystyäkseen suorittamaan tarjoamansa palvelut. Rajapinta käy ilmi olion luokkakuvauksessa. Esimerkiksi Java-kielen luokkakirjaston valmiit luokat löytyvät Internetistä osoitteesta:

http://java.sun.com/j2se/1.4.2/docs/api/allclasses-noframe.html

Viesti

Oliot kommunikoivat keskenään lähettäen viestejä toisilleen. Tämän yhteistoiminnan avulla päästään ohjelman suorituksessa haluttuun lopputulokseen. Olioiden välistä viestintää voidaan kuvata yhteistyökaaviolla (collaboration diagram), jossa viestit numeroidaan niiden suoritusjärjestyksen selvittämiseksi (kuva 8.5).



Kuva 8.5 Olioiden väliset viestit

Luokka

Olion luomiseksi tarvitaan luokka (class), josta voidaan luoda yksittäisiä olioita. Esimerkiksi voitaisiin luoda luokka Henkilö, jonka tietojäsenet ovat nimi, ikä ja kengännumero. Tälle luokalle voitaisiin luoda yksittäisiä olioita, kuten olio, jonka tietojäsenet ovat Laura, 35 ja 41, ja toinen, jonka tietojäsenet ovat Ville, 22, ja 45. Yhdestä luokasta voidaan luoda niin monta oliota kuin haluaa. Olion luonti tekee tilanvarauksen koneen muistiin ja sen tuhoaminen poistaa tilanvarauksen. Luokka määrää, minkälaisia olioita siitä luodaan, määrittelemällä mitkä ovat kyseisten olioiden tietojäsenet ja metodit (kuva 8.6).

Kuva 8.6 Luokka

Työntekijä-luokan pseudokielinen kuvaus:

```
class Työntekijä {
        protected:
                String Etuninimi;
                String Sukuninimi;
                String Lähiosoite;
                String Henkilötunnus;
                Integer Työntekijänumero;
        public:
                function Vaihda palkanmaksutapa();
                function Tuota työraportti();
                function Laske lomapäivät();
}
```

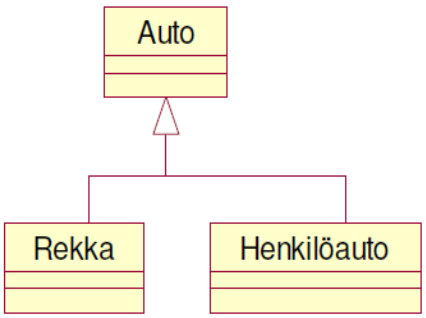
Tästä johtuen olion ja luokan väliset suhteet ovat seuraavat:

- Olio on luokan ilmentymä
- Olion ominaisuus on luokan ominaisuuden ilmentymä
- Olio käyttää ominaisuuksiaan luokan määrittelemillä metodeilla.

Tästä kaikesta seuraa, että toisin kuin proseduraalisessa ohjelmoinnissa, jossa ohjelmakoodi ja tieto (data) ovat erillään, olio-ohjelmoinnissa kaikki tieto kuuluu olioille, eikä siihen päästä käsiksi kuin luokkien metodien tarjoamilla rajapinnoilla.

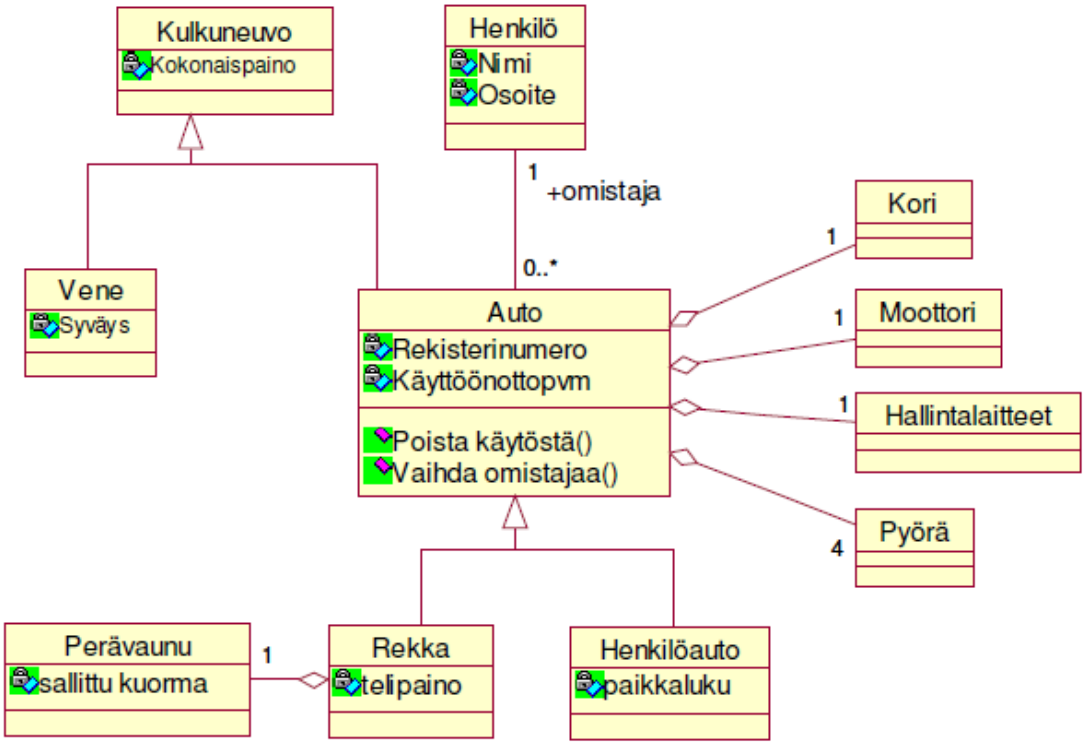
Perintä ja kooste

Perintä (inheritance) on yksi tärkeimpiä olio-ohjelmoinnin käsitteitä. Perimällä saadaan yliluokkaan (superclass) liitetyt ominaisuudet ja toiminnot käyttöön ilman erillistä määrittelyä. Perivää luokkaa kutsutaan aliluokaksi. Kuvassa 8.7 on yliluokka Auto, josta on periytetty aliluokat Rekka ja Henkilöauto. Perinnän tarkoitus on koodin uudelleenkäytettävyys sekä käsitteiden mallintaminen: perintä mahdollistaa myös abstraktien luokkien ja rajapintojen kirjoittamisen.



Kuva 8.7 Periytyminen

Sen lisäksi, että luokkia voidaan periyttää toisista luokista, niitä voidaan myös koostaa: kuvassa 8.8 esiintyvä auto-luokka koostuu Kori, Moottori, Hallintalaitteet ja Pyörä-luokista. Kuvasta käy ilmi myös, että Auto-luokallakin on oma yliluokkansa, Kulkuneuvo, josta on periytetty Auton lisäksi Vene.

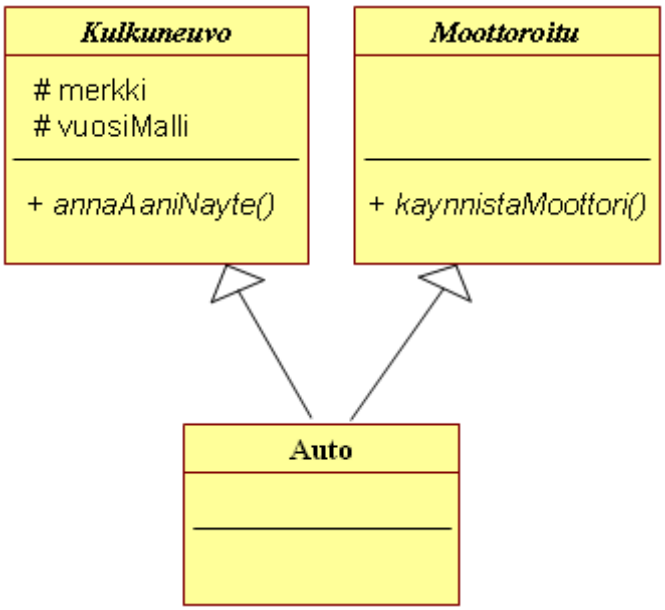


Kuva 8.8 Perintä ja kooste

Perinnän ja koostamisen tuloksena Rekka-luokalla on seuraavat tietojäsenet: *kokonaispaino*, joka on peritty Kulkuneuvo-luokalta, *rekisterinumero* ja *käyttöönottopvm*, jotka ovat peräisin Auto-luokasta. Lisäksi sillä on koostumussuhteen kautta Perävaunu-luokan *sallittu kuorma*-tietojäsen, sekä *telipaino*, joka on Rekka luokan oma tietojäsen. Tietojäsenien lisäksi Rekan käytössä on Auto-luokan metodit *Poista käytöstä()* ja *Vaihda omistajaa()*.

Moniperintä

Moniperintä tarkoittaa sitä, että luokalla on enemmän kuin yksi yliluokka. Moniperintä on mahdollinen C++:ssa, mutta esimerkiksi Java ei tue sitä, ja yleensä moniperintää ei suositeta käytettäväksi, sillä sen seurauksia on usein vaikea ennakoida. Kuvan 8.9 Auto-luokka on toisaalta Moottoroitu ajoneuvo ja toisaalta Kulkuneuvo, ja perii molempien yliluokkien ominaisuudet:



Kuva 8.9 Moniperintä (TTY ohjelmistotekniikan laitos / Timo Lehtonen 2004–2005)

8.3 Oliotestauksen erityispiirteitä

Olioiden testaaminen on erilaista proseduraalisen ohjelman testaamiseen verrattuna, sillä oliomenetelmillä ja -kielillä yksi asia voidaan tehdä monella eri tavalla. Kuten Murphyn laissa todetaan, jos on useampia tapoja tehdä jotain, ja yksi tavoista voi aiheuttaa katastrofin, joku tekee sen kyllä.

Tämä monimuotoisuus aiheuttaa omat riskinsä: esimerkiksi polymorfismi, dynaaminen sidonta ja perintä tekevät suunnittelijalle mahdolliseksi tehdä monimutkaisia rakenteita helposti, mutta virheiden tekeminen helpottuu samalla: perinnässä koko metodin toiminta voi muuttua ei-toivotuksi perintähierarkiastaan johtuen ja dynaaminen sidonta aiheuttaa sen, että metodin kutsu voi vaihtua suorituskertojen välillä.

Olio-ohjelmistojen testaamisessa on myös se ongelma, että oliot ovat tilariippuvia ja niiden tila täytyy huomioida testauksessa. Ongelmia aiheuttaa se, että kapselointi piilottaa olion tilatiedot, ja niihin, kuten muihinkin olion yksityisiin tietoihin, on hankalaa päästä käsiksi.

Myös se, että luokka on suhteellisen pieni kokonaisuus, aiheuttaa ongelmia: koska rajapintoja on huomattavasti enemmän kuin lausekielisessä ohjelmoinnissa, tulee kutsurajapinnan virheitäkin enemmän. Se, että olio-ohjelmoinnissa on käytettävissä myös abstrakteja luokkia ja rajapintoja, hankaloittavat testaajan työtä entisestään: suora testaus ei onnistu, vaikka luokat ja rajapinnat sisältävät ohjelman suorituksen kannalta oleellista tietoa.

Poikkeuskäsittelyssäkin on omat haasteensa: metodin suoritus voi keskeytyä tai siirtyä uuteen paikkaan ilman, että siirtymää voi nähdä suoraan koodissa. Rinnakkaisuus on myös yksi olio-ohjelmistojen testauksen haaste: useita toisistaan riippuvia metodeita suoritetaan samaan aikaan. Näin voi tapahtua proseduraalisessakin ohjelmoinnissa, mutta oliomaailmassa se on yleistä.

8.3.1 Oliojärjestelmien yksikkötestaus

Luonteva yksikkötestauksen kohde oliojärjestelmissä on luokka, sillä metodit, vaikka ne ovatkin pienempiä yksiköitä kuin luokka, ovat merkityksettömiä ilman luokkaa, johon ne kuuluvat. Muita yksikkötestauksen kohteita ovat luokkien yhteenliittymät: luokkaklusterit ja pienet aliohjelmat, jotka testataan yhdistettyinä, kunhan niiden sisältämät luokat on ensin testattu yksittäin. Luokan käyttökohde muuttaa testauksen vaatimuksia sen mukaan, onko kyseessä tiettyyn sovellukseen kirjoitettu luokka, yleiskäyttöinen, esimerkiksi merkkijonon käsittelyyn tarkoitettu luokka, abstrakti luokka vai parametrisoitu luokka (template class).

Olioita testatessa pitää muistaa, että yhden luokan ilmentymän, eli olion, testaaminen kertoo ainoastaan sen, miten kyseinen luokka toimii yksin käytettynä. Kun jo aiemmin testatuista luokista luodaan järjestelmä, on luokat syytä testata kokonaisuutena, että saataisiin oikea kuva siitä, miten ne toimivat juuri tässä kokoonpanossa.

8.3.2 Perinnän vaikutukset testaukseen

Periytettyjen (inheritance) metodien uusintatestauksen pitäisi olla enemmin sääntö kuin poikkeus, sillä periytyminen tarkoittaa sitä, että luokan käyttökohde muuttuu. Moniperintää käytettäessä muutoksia tulee vielä enemmän, ja testauksen kohde monimutkaistuu koko ajan. Koska metodin suoritus ei riipu pelkästään siitä luokasta jossa se on määritelty, vaan koko perintähierarkiasta, ei toimiva yläluokka takaa, että olio toimisi oikein edes niiden metodien osalta, joita ei ole muokattu perinnässä. Koska laajojen perintähierarkioiden vaikutusten ymmärtäminen on erittäin hankalaa, ne ovat potentiaalisia virhelähteitä. (Binder, 1994.)

Perintään liittyvä metodien kuormitus tarkoittaa, että samannimiselle metodille on useita toteutuksia luokkahierarkian eri tasoilla ja se, mikä luokka metodin toteuttaa, riippuu siitä, mitkä parametrit metodille annetaan. C++ mahdollistaa myös operaattoreiden kuormittamisen. Tämä johtaa siihen, että voidaan rakentaa ohjelmisto, jota on miltei mahdoton testata millään yleisillä testausmenetelmillä. (Taina, 2007.)

Voidaan todeta, että perintä vaikeuttaa luokan testaamista. Vaikka luokalla olisi vain muutama oma metodi, sillä voi olla perinnän kautta satoja perittyjä ominaisuuksia.

8.3.3 Polymorfismin ja dynaamisen sidonnan vaikutukset testaukseen

Olio-ohjelmoinnissa polymorfismi (polymorphism) tarkoittaa sitä, että operaattori tai metodi voidaan sitoa erilaisiin toteutuksiin tilanteesta riippuen. Sitominen voidaan tehdä muun muassa dynaamisen sidonnan kautta, joka tarkoittaa sitä, että ohjelman suorituksen aikana, metodia kutsuttaessa, kääntäjä tutkii muuttujan tyypin ja tarkastaa, että muuttujan luokalla tai yliluokalla on kutsuttu metodi. Kun käy ilmi, että olion tyyppi on eri, kuin olioon viittaavan muuttujan tyyppi, aletaan etsiä kutsuttua metodia ensin olion omasta luokasta. Jos metodia ei löydy, etsimistä jatketaan sen yliluokasta ja näin edelleen.

Polymorfismi ja dynaaminen sidonta on operaattoreiden kuormituksen lisäksi vaikeimmin testattavia olio-ohjelmoinnin piirteitä, sillä kaikki mahdolliset sidonnat on testattava, ja jos sidonnat ovat riippuvaisia toisistaan, myös sidontojen kombinaatioiden testausta tarvitaan. Jos olion sidonta on tilariippuvainen, tarvitaan sidontojen tilariippuvaa testausta. Tämä tarkoittaa, että testejä suunniteltaessa on ohjelman rakenteen lisäksi oltava tiedossa kaikki mahdolliset sidontakombinaatiot, ja niiden selville saaminen laajoista perintähierarkioista voi olla vaikeaa.(Taina, 2007.)

8.3.4 Tilariippuvuus

Olion tila määräytyy sen attribuuttien mukaan, ja jos metodin toiminta riippuu syöteparametrien lisäksi sen olion, joka metodin omistaa, tilasta, pitää myös olion tila huomioida testauksessa. Olion tilat ja tilasiirtymät muodostavat äärellisen automaatin, joka on eräänlainen tilakone. Tilakone on luokka, jossa on äärellinen määrä toisistaan eroavia tiloja. Luokan oliot siirtyvät tilasta toiseen metodien avulla. Tilakonetta testatessa jokainen tila ja jokainen siirtymä testataan. Tässä on ongelmansa, jos tilakone kuvaa monimutkaista ja monitasoista toimintaa. Tiloja ja varsinkin niiden siirtymiä tulee hallitsematon määrä, ja tapahtuu

tilaräjähdys. Tällöin äärellisen automaatin kattava testaus tuottaa liikaa testitapauksia ja niitä on karsittava. Karsimiseksi voidaan käyttää täyden tilakattavuuden sijasta yksinkertaista siirtymäkattavuutta (simple transition coverage), jossa jokainen siirtymä testataan ainakin kerran, mutta kaikkia kombinaatioita ei testata.(Taina, 2007.)

8.3.5 Tiedon kapseloinnin vaikutus testaukseen

Se, että olio omistaa omat tietonsa, aiheuttaa lisätyötä testauksessa. Metodien testaamiseksi sen omistama olio täytyy rakentaa sellaiseksi, että testauksessa päästään käsiksi myös olion suojattuihin metodeihin, tietojäseniin ja sen tiloihin. Tämä tarkoittaa, että oliopohjaisen ohjelman testaamiseksi on käytännössä toteutettava testausrajapinta, jonka kautta voidaan nähdä koko olion rakenne alioliоineen. (Taina, 2007.)

8.3.6 Abstraktien luokkien vaikutus testaukseen

Abstraktit luokat ja rajapinnat testataan aliluokkien avulla, koska abstraktilla luokalla ei ole toteutusta ja siitä ei näin ollen voi luoda oliota. Kun abstraktia luokkaa testataan, on huolehdittava siitä, että kaikki luokan ja sen rajapinnan ominaisuudet tulee testattua. On mahdollista, että on generoitava keinotekoinen aliluokka, jota ei käytetä muuhun kuin testitarkoituksiin. (Taina, 2007.)

8.3.7 Poikkeuskäsittelyn vaikutus testaukseen

Poikkeuskäsittely on teoriassa hankalaa, koska mikä tahansa poikkeus voi tapahtua missä tahansa kohtaa ohjelmakoodia, mutta käytännössä tiedetään suhteellisen hyvin, missä poikkeukset tapahtuvat ja mitä niistä seuraa. Tärkeää poikkeuskäsittelyn testaamisessa on, että kaikki poikkeukset ja niiden käsittelijät tulee testattua. Ongelmallisin tilanne on silloin, kun metodi palauttaa poikkeuksen, jota sitä kutsunut metodi käsittelee, eli poikkeus ja poikkeuksen käsittelijä ovat eri metodeissa, mutta tämä tilanne hoidetaan niin, että nämä luokat testataan yhdessä.(Taina, 2007.)

8.3.8 Rinnakkaisuuden vaikutus testaukseen

Olio-ohjelmat eivät tarvitse ohjaavaa pääohjelmaa, joten ne voidaan hajauttaa useammalle rinnakkaiselle prosessille. Tästä seuraa, että kaikki mahdolliset metodien ja äärellisten automaattien skeduloinnit on testattava, kuten kaikki mahdolliset resurssien käytön kilpailutilanteetkin. Rinnakkaisuuden ongelmat ovat ratkaistavissa, sillä niiden hallintaa on harjoitettu jo kymmeniä vuosia prosessien ja tietokantojen hallinnassa.(Taina, 2007.)

8.4 Oliojärjestelmän testaus

Oliojärjestelmiä testatessa ensimmäinen tehtävä on saada selkeä kuva siitä, miten järjestelmän olisi tarkoitus toimia. Tätä tehtävää helpottaa mallinnus: luokkakaavioista ja käyttötapauksista voidaan johtaa testitapauksia. Olioita mallinnetaan UML-kielellä, ja näiden mallien ja kaavioiden validointiin on käytettävä aikaa, sillä niissä olevat virheet tekevät testauksesta epäluotettavan. Myös ohjelmakoodia kannattaa katselmoida: se on käyttökelpoinen tapa löytää yleisiä ohjelmointivirheitä, kunhan katselmoitavaa on riittävän vähän kerrallaan.

Oliojärjestelmissä luokkien yksikkötestauksen suorittaa yleensä ohjelmoija itse, ja ohjelmiston luotettavuuden kannalta se on tärkein testausvaihe. Yksikkötestauksen jälkeen testataan luokkaklusterit, ja näiden testauksessa painopiste tulisi olla olioiden rajapintojen toiminnallisessa testauksessa. Toiminnallisella testauksella selvitetään, miten tieto kulkee klusterin sisällä, kuinka poikkeuskäsittelijät toimivat ja miten polymorfiset metodikutsut toimivat luokkien välillä.

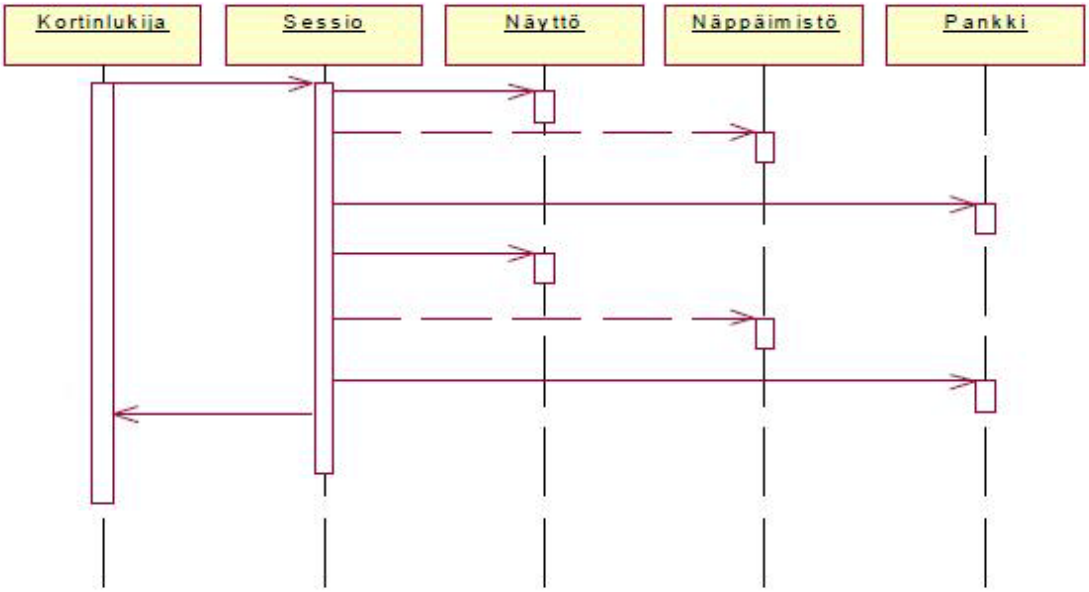
Olioiden järjestelmätestaus ei poikkea lausekielisten ohjelmistojen testauksesta: samat laatunäkökulmat pätevät täälläkin. Suorituskyky, luotettavuus, siirrettävyys, ylläpidettävyys ja käytettävyys loppukäyttäjän näkökulmasta ovat testauksen aiheina oliojärjestelmissäkin.

8.4.1 UML-mallit testitapauksien pohjana

Oliojärjestelmän testitapauksia voidaan johtaa sitä kuvaavista UML-malleista. Näistä käyttökelpoisimpia ovat sekvenssikaaviot ja luokkakaaviot. Sekvenssikaavioita voidaan käyttää eri testaustasoilla, sillä niitä voidaan tehdä eri arkkitehtuuritasoille: esimerkiksi järjestelmän toiminnallisena kuvauksena toimiva sekvenssikaavio esittää aikaperspektiivissä eri olioiden välisen sanomien lähettämisen ja vastaanottamisen. Sekvenssikaavioita laadittaessa ei yleensä kuvata kaikkia mahdollisia vuorovaikutuksia vaan ainoastaan tärkeimmät ja eniten käytetyt. Sekvenssikaaviot toimivat perustana vuorovaikutuksia testaaville testitapauksille: perustestinä voidaan testata kaikki kaavioissa kuvattu toiminnallisuus, eli jokainen skenaario alusta loppuun, ja kaikki mahdolliset sekvenssikaavion polut voidaan myös kirjoittaa testitapauksiksi, myös ne, joita ei ole erikseen kuvattu. Kaikki vaihtoehtoiset skenaariot on syytä kirjata ylös testitapauksiksi jo suunnitteluvaiheessa, jolloin niitäkin voidaan käyttää testitapauksina. Testitapauksia laadittaessa:

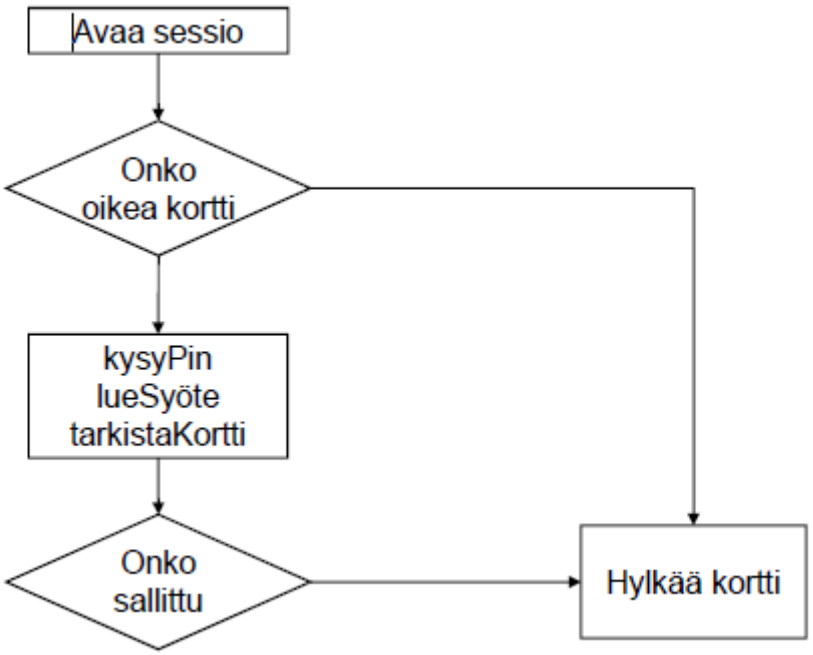
- Niiden laatiminen pohjautuu siihen, että tunnistetaan erilaiset käyttöskenaariot.
- Sanomien tarkemmat määrittelyt on haettava muista lähteistä, kuten käyttötapauskuvauksista.
- Tunnistetut skenaariot voidaan kuvata sanallisesti tai luomalla niistä vuokaavioita.

Kuvassa 8.10 on esimerkki sekvenssikaaviosta, joka on laadittu pankkiautomaatin toiminnan perusteella:



Kuva 8.10 Sekvenssikaavio (TTY Pori, Arto Stenberg)

Automaatin toimintaa, jota sekvenssikaavio kuvaa, voidaan kuvata seuraavanlaisella vuokaaviolla (kuva 8.11):

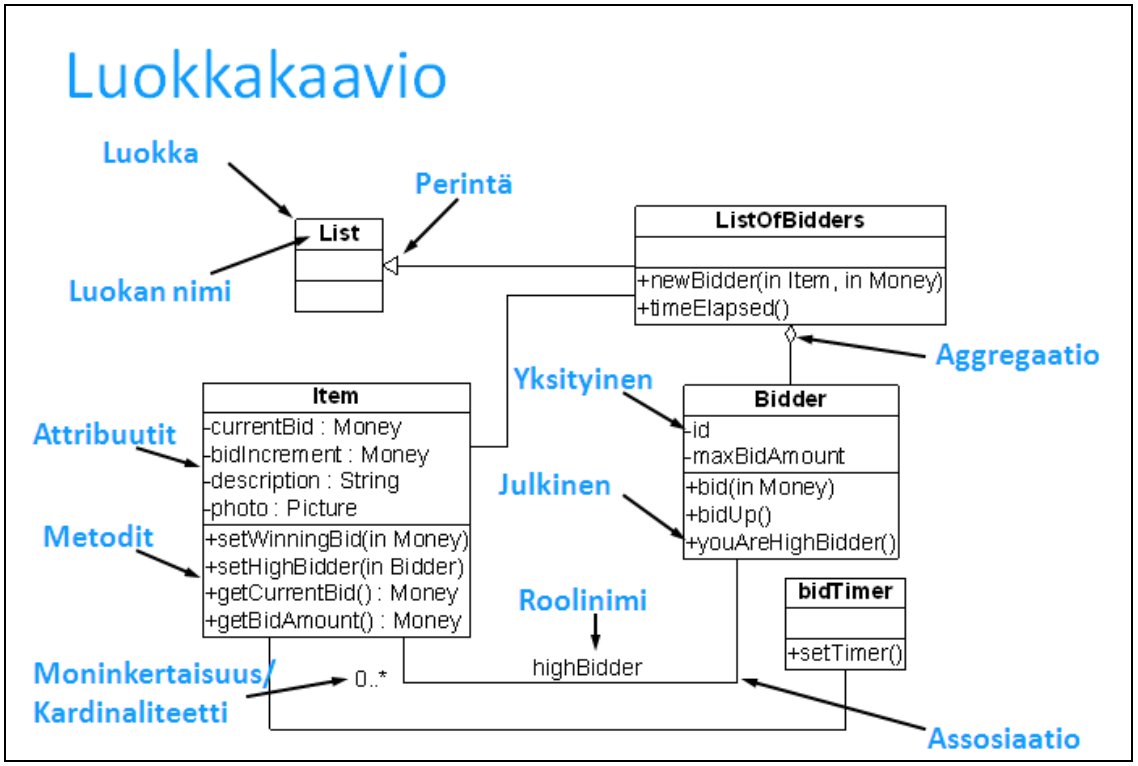


Kuva 8.11 Vuokaavio (TTY Pori, Arto Stenberg)

Sekvenssikaavioiden perusteella löydettäviä vikoja:

- Väärä tai puuttuva operaatio.
- Virheellinen sanoman määrittely tai operaation rajapinta.
- Olion operaation väärä toteutus.
- Oikea sanoma välitetään väärälle oliolle.
- Väärä sanoma välitetään oikealle oliolle.
- Oliot luodaan tai tuhotaan väärän aikaan.
- Puutteellinen poikkeuskäsittely.
- Suorituskykyyn liittyvät ongelmat.
- Resurssien käytön ongelmat, kuten kilpailutilanteet.

Oliojärjestelmien kuvaamisessa sekvenssikaavioiden ohella myös luokkakaaviot ovat hyödyllisiä testaukselle. Luokkakaavio esittää luokkien rakenteen ja niiden väliset suhteet. Testauksen suunnittelun kannalta oleellisia ovat luokkien hierarkia, luokan sisäinen vuorovaikutus, perintäsuhteet, monimuotoisuus ja assosiaatiot. Luokkakaavion sisältämää tietoa on selvitetty kuvassa 8.12:



Kuva 8.12 Luokkakaavio (Conformiq Software Ltd.)

8.4.2 Olioiden testaus

Olioiden testausta suunnitellessa on selvitettävä itselleen, minkälaisia testattavat oliot ovat: toisessa ääripäässä ovat oliot, jotka ottavat vastaan minkälaisia sanomia tahansa muuttamatta käyttäytymistään, ja toisessa oliot, jotka hyväksyvät vain tietynlaisia sanomia, tietynlaisessa järjestyksessä ja voivat siirtyä vain tiettyihin tiloihin. Ensin mainitut oliot eivät ole tilariippuvia mutta toiseksi mainitut ovat, ja on olemassa olioita, jotka ovat jotain tältä väliltä. Olion luonne vaikuttaa siihen, kuinka se tulisi testata.

Ei-tilapohjaista oliota testataan lähettämällä niille erilaisia sanomia ja tarkistamalla olion tila eli attribuutit säännöllisin väliajoin. Virheet, joita on tarkoitus löytää, ovat, että laillista sanomakutsua ei hyväksytä, sanomakutsu tuottaa väärän lopputuloksen tai attribuutin tilaa muuttava, tai sen arvon palauttava operaatio ei toimi.

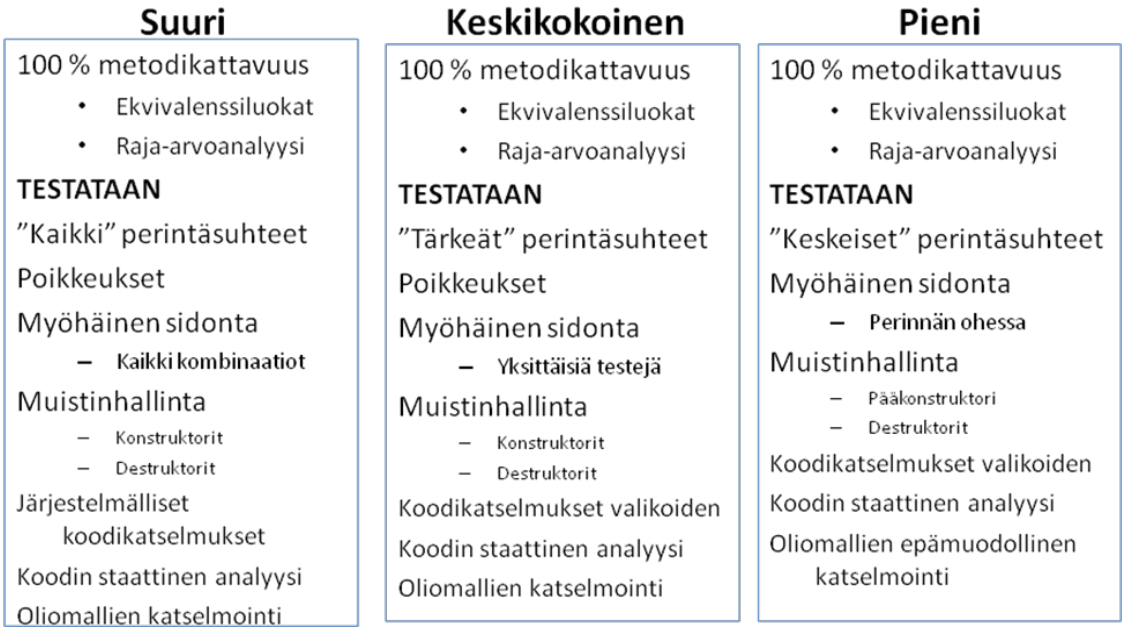
Tilapohjaisen olion testaamista varten kannattaa olion tilat mallintaa UML:lla jo sen suunnitteluvaiheessa. Tilamallista käyvät ilmi olion tilat, ehdot, siirtymät ja tapahtumat. Tilapohjaisten olioiden tyypillisiä virheet ovat muun muassa puuttuvat tai ylimääräiset tilat, puuttuvat tai ylimääräiset tilasiirtymät, väärät siirtymät, se, että malliin kuuluvaa tilaa ei huomioida oikeassa tilassa tai se, että se hyväksytään väärässä tilassa.

Yksittäisen olion elinkaari testataan seuraavalla tavalla:

- Luodaan olio.
- Kirjoitetaan vähintään yksi testitapaus jokaiselle muodostimelle.
- Alustetaan olio perustilaan, tai johonkin muuhun sopivaan tilaan.
- Käydään läpi kaikki tilasiirtymät.
- Edellä mainittu toistetaan jokaiselle olion attribuutille.
- Kutsutaan testattavaa metodia.
- Testijoukkoa karsitaan eri tekniikoin, kuten ekvivalenssiluokkiin jakamisella ja raja-arvoanalyysilla.

- Tarkistetaan kutsutun olion attribuuttien tila ja sivuvaikutukset jokaisesta. kutsusta.
- Tehdään sama olion kaikille parametreille.
- Tehdään sama jokaiselle testattavalle metodille.
- Tuhotaan olio.
- Testataan hajottaja ja tarkistetaan, että kaikki olion varaamat resurssit on todella vapautettu.
- Tarkistetaan kattavuusluvut ja tarvittaessa lisätään testitapauksia.

Se, kuinka kattavasti luokka testataan, riippuu siitä, miten kriittinen olio on ohjelmiston toiminnan suhteen. Kuvassa 8.2 kuvataan luokkatestien skaalaamista erilaisille riskitasoille.



Kuva 8.13 Luokkatestien skaalaaminen erilaisille riskitasoille (Pöyhönen ym. 2002)

Oliojärjestelmiä testattaessa on muistettava, etteivät perinteiset koodin kattavuusmitat kerro koko totuutta oliotestauksen perusteellisuudesta johtuen perinnästä ja dynaamisesta sidonnasta. Tästä syystä koodikattavuuden lisäksi voidaan seurata metodien kutsumista, määriteltyjen poikkeusten tuottamista sekä olioiden luontia eri tavoilla kuten dynaamisesti, viitaten tai staattisesti.

Olioiden kompleksisuus ja koko ovat myös asioita, joita on syytä tarkastella ohjelmointiriskien selvittämiseksi. Kompleksisuuden ja koon mittaamiseen löytyy perinteisten mittareiden lisäksi olioperusteisia mittareita, jotka ottavat huomioon kutsu- ja perintärakenteisiin, sekä olion rajapinnan ominaisuuksiin. (Pöyhönen ym. 2002.)

8.4.2 TDD ja Mock-objektit

TDD eli Test Driven Development, joka on selitetty luvussa 4.4.2, on ketterän XP-menetelmän tapa tuottaa yksikkötestejä. TDD:tä käytetään olioiden yksikkötestauksessa myös silloin, kun ohjelmistoprojekti ei seuraa ketteriä käytäntöjä. Työtavan hyvät puolet ovat, että testit toimivat luokan määrittelyinä ja ne ovat ajettavissa erikseen. Testausta ei myöskään laiminlyödä aikatauluongelmien takia, kun testit kirjoitetaan ennen tuotantokoodia. Perustoiminnallisuus tulee testattua hyvin, joten seuraavat testausvaiheet nopeutuvat: riippuvuuksien määrä pysyy mahdollisimman pienenä kun luokat pyritään aina testaamaan erikseen ja testikattavuus on lähellä 100 %:n lausekattavuutta. Menetelmän heikkouksia ovat testauksen laadun riippuvuus suunnittelijan osaamisesta ja motivaatiosta sekä se, että testit kattavat yleensä ainoastaan perustoiminnallisuudet. Myös se voidaan laskea heikkoudeksi, että testauskoodin määrä ylittää aina ohjelmakoodin määrän.(Pöyhönen ym, 2002)

Usein olion testaamiseksi ei riitä, että sen sisäinen toiminta on saatu testattua. Oliot voivat tarvita yhteyttä ulkoisiin tietolähteisiin, olla aikakriittisiä tai muulla tavoin hankalia niin, että tarvitaan ympäristö, missä rajapintoja voidaan testata. Tähän tarkoitukseen voi käyttää Mock-olioita (Mock Object) eli korvikeolioita, joilla voidaan korvata monimutkaisia komponentteja yksinkertaisemmilla, joiden toiminnallisuus on määritettävissä testikohtaisesti. Mock-olioita käytetään yleensä korvaamaan rajapintaluokkia, ja ne voidaan laatia itse tai valmiita apuvälineitä, kuten EasyMockia, hyväksikäyttäen.

9 TESTITYÖKALUT JA AUTOMAATIO

Tässä luvussa käsitellään testauksen automatisointia, testaustyökaluja ja niiden luokittelua. Testaustyökalulla tarkoitetaan sovellusta, joka tukee yhtä tai useampaa testauksen toimintoa, kuten suunnittelua, valvontaa, määrittelyä, testaustietokantojen luomista, testien suorittamista tai niiden analyysia. Määritelmässä sana ”tukea” on oleellinen, sillä testaustyökalun käyttämisen tarkoitus on testausprosessin parantaminen. Työkalulla pitäisi saada aikaiseksi yksinkertaisempi testausprosessi, ajansäästöä tai jotain muuta lisäarvoa testaukselle. Kun kysyin eräältä suuren suomalaisen tietotekniikan alan yrityksen testauspäälliköltä, mitä hänen mielestään oppilaitoksissa tulisi opettaa testauksen automaation tiimoilta, sain seuraavanlaisen vastauksen:

”Ainakin se, että automatisoida pitää harkiten, tai sillä ampuu itseään jalkaan. Mitä tahansa ei kannata automatisoida vaan vain stabiilit asiat / paljon toistoa vaativat asiat / yksikkötestaus mahdollisuuksien mukaan. Suurin harhaluulo on, että automatisointi vähentää testaustyön määrää - se pääasiassa vaan siirtää painopistettä toiseen kohtaan ja suorastaan räjäyttää työmäärän, jos automatisoidaan ilman harkintaa. Sen lisäksi ***testauksen automatisointi ei ole testausta, vaan ohjelmointia, jossa tulee ymmärtää testausta****, eli kuka tahansa testaaja ei automaattisesti ole testauksen automatisointiin kykenevä henkilö vaan tarvitaan erikoistaitoja (tämä tuli omalla painavalla kokemuspohjalla). Ja sitten vielä asia, joka minua on askarruttanut tässä, että kuka testaa sen, että automaatiokoodi toimii oikein?”*

Automaatio ei poista manuaalisten testien tarvetta, eikä tarvetta testaajille, jotka osaavat analysoida testien tuloksia, eivätkä automatisoidut testit ole verrattavissa manuaalisiin testeihin. Automaatiota tuleekin käyttää manuaalisen testauksen jatkeena silloin, kun sen avulla voidaan tehdä jotain, mihin ei manuaalisesti pystyttäisi.

9.1 Automaation vaikutus testauksen suunnitteluun

Testauksen suunnittelun ja automaation suhteen voi tehdä kaksi huomiota, jotka ovat ristiriidassa keskenään:

- **Testit tulee suunnitella ennen kuin päätetään, mitä automatisoidaan**. Jos näin ei tehdä, on vaarana, että automatisoidaan testejä, joita on helppo automatisoida, mutta jotka ovat huonoja löytämään vikoja.

- **Automaatiosta tulee päättää ennen testien suunnittelua**, sillä automatisoitujen testien suunnitteleminen on erilaista kuin manuaalisten testien: suurin hyöty automaatiosta on testeissä, joissa tietokone laitetaan tekemään asioita, joita ihminen ei pysty tekemään. Jos tätä ei muisteta, on vaarana, että pyritään vain automatisoimaan manuaalisiin testeihin kirjoitetun testaussuunnitelman testitapaukset unohtaen ne mahdollisuudet, jotka automaatio tarjoaa. Esimerkiksi: tietokoneohjelma voi toistaa samaa testitapausta tuhannella eri tiedostolla, mikä olisi ihmisen suorittamana hidas, vaivalloinen ja äärimmäisen tylsä tehtävä.

Näiden päätelmien perusteella voidaan todeta, että:

- Ilman hyvää testauksen suunnittelua automaation tuloksena voi olla paljon testaustoimintaa jonka tulokset ovat mitättömät.
- Testauksen suunnitteleminen ymmärtämättä automaation tarjoamia erityismahdollisuuksia voi aiheuttaa sen, että ne mahdollisuudet jäävät käyttämättä.

Onnistuneeseen testiautomaatioon vaaditaan sekä hyviä testauksen suunnittelijoita että hyviä automatisoijia. Tarvitaan erikoisosaamista, jotta kyetään valitsemaan testeistä ne, jotka kannattaa automatisoida ja että automatisoidut testit saadaan suunniteltua ja toteutettua tarkoituksenmukaisiksi. (Kaner ym. 2002, s. 93–94.)

Automaation käyttöä mietittäessä on muistettava myös se, että testaustyökalu ei ole testausstrategia: testaustyökalu ei kerro, kuinka tulisi testata. Hämmennys sen suhteen, mitä pitäisi tehdä, kasvaa entisestään, jos tilannetta monimutkaistetaan työkalujen hankinnalla. Testausprosessit on oltava kunnossa, ennen kuin niitä voidaan automatisoida. Ylipäänsä kaaoksen automatisoimisen arvo on kyseenalainen: jos testauksen laatu on huono jo alun perin, saadaan automatisoimalla tehtyä enemmän huonolaatuista testausta nopeammin. Jos testaus on epäorganisoitua ja testaajat eivät alun perinkään tiedä tai kerro, mitä ovat testaamassa, he tulevat todennäköisesti luomaan automaatiota, josta kukaan muu ei voi ymmärtää, miten sitä tulisi käyttää.

Yksi huomionarvoinen seikka suunnittelun suhteen on se, että käytetään työkaluja, jotka sopivat kulloinkin testauksen kohteena olevan järjestelmän testaamiseen. Ei ole tarkoituksenmukaista tuhlata työtunteja siihen, että testattava ohjelma saataisiin taipumaan valitun työkalun tarpeisiin. Testauksen kohteen ominaisuudet, kuten käytetyt ohjelmointikielet, ympäristöt ja komponentit vaikuttavat siihen, minkälaiset työkalut ovat käyttökelpoisia. Ohjelmiston rajapinnat vaikuttavat ratkaisevasti mahdollisuuksiin automatisoida sen testausta. Myös ohjelmiston ominaisuuksien kriittisyys vaikuttaa siihen, kannattaako testausta automatisoida: avainominaisuuksien tulee olla luotettavia ja vakaita, ja ne pyritään testaamaan kattavasti: automaatiolla on siellä sijansa. Vähemmän tärkeille ominaisuuksille voi riittää manuaaliset testitapaukset.

Lisäksi on otettava huomioon se, minkälaisia testaajia organisaatiossa on tarjolla: jotkut työkalut ovat hyviä apuvälineitä testaajille, jotka eivät osaa ohjelmoida, mutta toisten käyttö vaatii osaavien testaaja-ohjelmoijien taitoja. (Kaner ym. 2002, s. 97.)

Yhteenvetona voidaan todeta, että yhtä, oikeaa tapaa automatisoida ei ole: vaikka projektit ja testattavat sovellukset olisivatkin samankaltaisia, voivat testaushenkilöstön kapasiteetti ja taidot vaihdella niin paljon, että samat lähestymistavat eivät ole mahdollisia.

9.1.1 Automaation vaikutus ohjelmiston kehitysprosessiin

Testauksen automaatiolla on mahdollista nopeuttaa ohjelmiston kehitysprosessia. Automatisoitujen testien avulla voidaan:

- Havaita nopeasti epävakaudet uudessa ohjelmistoversiossa.
- Paljastaa muutosten aiheuttamat viat niin nopeasti kuin mahdollista.
- Raportoida ongelmista nopeasti, jolloin niiden korjaaminen helpottuu.

Edellä mainitut seikat auttavat pitämään tuotantokoodin vakaana. Koodin stabiilius taas auttaa säästämään aikaa, kun useiden ihmisten ei tarvitse tuhlata aikaansa saman virheen takia, ja helpottaa koodin refaktorointia sekä muita pyrkimyksiä ohjelmiston rakenteen parantamiseksi ja spagettikoodin oikomiseksi. Kun tuotannossa on vakaa ohjelmisto ja kattava joukko automatisoituja testejä, on ohjelmoijien mahdollista tehdä suurempia muutoksia koodiin pienemmällä riskillä. Tämä taas mahdollistaa sen, että projektilla on paremmat mahdollisuudet vastata asiakkaan tai markkinoiden vaatimiin muutostarpeisiin.

Tekniikoita, jotka tukevat ohjelmiston kehitystä, ovat muun muassa yksikkötestien ja ”savutestien”, eli ohjelmistoversion verifiointitestien automatisoiminen. Automatisoiduilla yksikkötesteillä tarkoitetaan testijoukkoa, jolla on testattu ohjelmiston alimman tason funktiot tai luokat. Ne ovat mahdollisesti syntyneet TDD:n avulla, ja ne tehostavat kehitysprosessia, kun on olemassa valmis joukko testejä, joita voi käyttää regressiotestauksessa ja ohjelmiston päivitystöissä. Automatisoidut savutestit taas tarkoittavat testijoukkoa, joka kertoo, onko ajettavan ohjelmistoversion tärkeimmät ominaisuudet kunnossa vai ei. Kun savutestit ajetaan aina ohjelmiston koostamisen yhteydessä, voidaan havaita heti, jos koosteessa on ongelmia. Jos ohjelmisto ei läpäise savutestejä, ei testaustiimin tarvitse tuhlata aikaa uuden ohjelmistoversion asentamiseen ja testaamiseen. Testausta jatketaan vasta, kun ohjelmisto on korjattu siihen kuntoon, että se läpäisee savutestit.

Automatisoitujen yksikkö- ja savutestien suurin etu on se, että niitä voidaan ajaa milloin vain ja kenen toimesta hyvänsä. Ne voivat toimia testauksen lisäksi myös ohjelmoijien tukena. He voivat käyttää testejä osajärjestelmien kehityksessä, ja jos osajärjestelmä läpäisee testijoukon, on todennäköistä, että koko järjestelmän koostaminen testausta varten onnistuu myös, ja testaajat pääsevät tekemään työtään uuden version parissa. Vaikka tämän kaltaisten testien luominen vaatii aikaa, työtä, ammattitaitoa ja rahaa, ja yhteistyötä ohjelmoijien kanssa, niiden hyöty on merkittävä sekä kehitys- että testausprosessille. (Kaner ym. 2002, s. 93–95.)

9.1.2 Automaation kohteet

Edellisessä luvussa mainittujen yksikkö- ja savutestien lisäksi järkevät automatisoinnin kohteet löytyvät testeistä, joiden manuaalinen suoritus olisi hankalaa tai jopa mahdotonta. Seuraavaksi kuvataan muutamin esimerkein, minkälaisissa tilanteissa testauksen automaatiosta on eniten hyötyä:

Kuormitustestaus – mitä tapahtuu, kun 200 ihmistä yrittää käyttää sovellusta samaan aikaan? Entäpä 2000? Tämänkaltaisten tilanteiden simuloimiseen tarvitaan automaatiota.

Suorituskykytestaus – onko ohjelmiston suorituskyky parempi vai huonompi entiseen nähden? Kun samat testisarjat suoritetaan samoilla resursseilla joka kerta, voidaan tulokset koota aikajanalle ja havaita, jos ohjelmiston suorituskyky heikkenee.

Kokoonpanon testaus – toimiiko ohjelmisto erilaisilla alustoilla, konfiguraatioilla ja liitettynä erilaisiin lisälaitteisiin? Automaatiolla voidaan parantaa tämänkaltaisten testien kattavuutta, kunhan vain huolehditaan siitä, että itse testit ovat siirrettävissä eri alustoille.

Kestävyystestaus – Mitä ohjelmistolle tapahtuu, kun sitä käytetään viikkoja tai kuukausia? Esimerkiksi muistivuotojen ja osoitinvirheiden aiheuttamat ongelmat eivät välttämättä ilmene heti, mutta ajan saatossa ne aiheuttavat ongelmia. Yksi mahdollisuus on ajaa samaa testisarjaa pitkin aikavälein, niin että ohjelmisto on ollut toiminnassa ilman että sitä on käynnistetty välillä uudelleen. Tämä vaatii automaatiota.

Kilpailutilanteiden testaus - ”Race condition” tarkoittaa tilannetta jossa sama resurssi on jaettu siten, että operaatioiden suoritusjärjestys vaikuttaa lopputulokseen. Tällaista tilannetta ei voi testata kuin automaation avulla, toistaen samaa testisarjaa hienoisesti toisistaan poikkeavilla ajastuksilla.

Kombinaatioiden testaus – jotkut virheet liittyvät usean ominaisuuden yhteistoimintaan. Automaation avulla voidaan ajaa suuri määrä testejä, joissa näitä ominaisuuksia yhdistellään eri tavoin.

Kaikkien edellä mainittujen testien lähtökohtana on käyttää automaatiota uusien testien luomiseksi tai testattavan ohjelman suorituksen toistamiseksi niin, että löydettäisiin siitä uusia vikoja. Tämänkaltaisten testien kehittäminen ei ole yksinkertaista, vaan vaatii työtä testaus- ja kehitystyökalujen parissa. Se on kuitenkin vaivan arvoista, sillä nämä testit tuovat todennäköisesti huomattavasti enemmän uutta tietoa testauksen kohteesta kuin se, että toistettaisiin automatisoituja toiminnallisuustestejä uudelleen ja uudelleen. (Kaner ym. 2002, s. 96.)

9.2 Työkalujen valinta ja käyttöönotto

Tässä luvussa käsitellään testauksen automatisointiprojektin läpiviemistä ja sitä, mitä on otettava huomioon onnistuneen automatisointiprojektin aikaansaamiseksi.

9.2.1 Ennen automaatiota

Ennen kuin testauksen automatisoinnin voi aloittaa, on muistettava, että automaatiolla tarkoitetaan manuaalisen tehtävän suorittamista koneellisesti. Tämä tarkoittaa sitä, että manuaalinen tehtävä on oltava olemassa: testausjärjestelmän pitäisi jo ennen automatisointia sisältää testitapaukset, määrittely- ja suunnitteludokumentteihin perustuvat testitapausten tulokset sekä testausympäristön testitietokantoineen, jonne testitapaukset voidaan tallettaa uudelleensuoritusta varten.

Kun edellä mainitut asiat ovat kunnossa, pitää miettiä sitä, kuinka hyvä testien toistettavuus lopulta on: manuaalisen testin tekeminen on nopeampaa ja halvempaa, joten testejä, joita ei sellaisenaan kannata toistaa, ei kannata myöskään automatisoida. Jos testauksen kohde muuttuu usein, on parempi käyttää aika manuaalisten testien tekemiseen automaation sijasta. Huonosti kohdennettu automaatio syö resursseja ja heikentää testauksen tuloksia. Kun mietitään, mitkä ovat edellytykset testitapauksen automatisoinnille, kannattaa kiinnittää huomiota seuraaviin seikkoihin:

- Onko testi toistettavissa samanlaisena monta kertaa niin, että testin suorittamisesta on hyötyä testaukselle?

- Onko testitapaus mahdoton suorittaa manuaalisesti, eli laajeneeko testauksen kattavuus automatisoinnin avulla?

- Selvitäänkö automatisoimalla testattavan sovelluksen muutoksista pienemmällä testitapausten määrällä kuin manuaalisesti testatessa?

- Kuinka monta koodimuunnosta testi sietää, ennen kuin siitä tulee käyttökelvoton? Jos testi ei kestä muunnoksia, on sen oltava erittäin tehokas löytämään virheitä, ennen kuin automatisointi kannattaa.

Kuten tämän dokumentin johdannossakin todettiin, ohjelmistoprojektit epäonnistuvat usein. Testausprojektit epäonnistuvat yhtä usein kuin muutkin projektit, tai jopa useammin, koska testausvälineisiin ei välttämättä ole innostuttu panostamaan rahaa ja aikaa samalla tavalla kuin muuhun ohjelmistokehitykseen. Tähän yliolkaisuuteen kun lisätään vielä testausautomaatiota kauppaavien yritysten mainospuheet siitä, miten napinpainalluksella saadaan aikaiseksi laadukasta testausta joka kattaa ”kaiken”, ollaan vaarallisilla vesillä. Automaatiota suunnitellessa pitää miettiä tarkasti, mitkä testaustehtävistä ovat sellaisia, joiden suorittaminen automatisoidusti on nopeampaa, halvempaa ja luotettavampaa kuin ihmisvoimin tehty testaus. (Kaner ym. 2002, s. 97. Pohjolainen, 2003.)

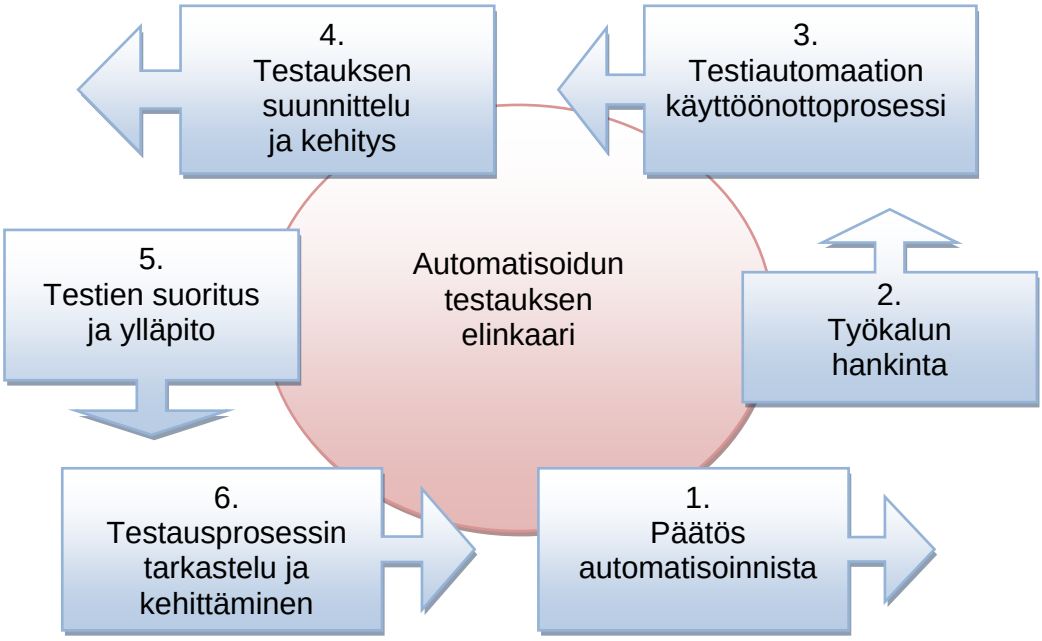
9.2.2 Projektiryhmän muodostaminen

Kuten muidenkin ohjelmistoprojektien kohdalla, myös testausautomaatioprojektissa lähdetään siitä, että muodostetaan projektiryhmä. Automatisointiprojektit ovat monimutkaisia, ja ne vaativat johtajakseen jonkun, kenellä on kokemusta suunnittelusta ja ohjelmoinnista. Ei ole mahdollista onnistua, jos testausryhmä muodostuu kokemattomista ohjelmoijista ajatuksella ”kun ei se kerta osaa koodata niin menkööt testaamaan”. Automatisoinnista ja siihen tarvittavista työkaluista päättäminen pitäisi antaa mahdollisimman kokeneelle suunnittelijalle tai ohjelmoijalle, jotta projekti onnistuisi.

Sen lisäksi, että saadaan koottua projektiryhmä, jonka ammattitaito ja kokemus riittävät projektin läpiviemiseksi, on saatava myös yrityksen johto sitoutumaan automaatiohankkeeseen. Ei riitä, että johto suostuu työkalujen hankintaan, sen pitää antaa suostumuksensa myös työkalujen ja automaation vaatimaan kouluttautumiseen ja ymmärtää se, että automaation rahalliset hyödyt alkavat todennäköisesti näkymään vasta myöhemmin kuin pilottiprojektin puitteissa. (Pohjolainen, 2003.)

9.2.2 Automatisoidun testauksen vaiheet

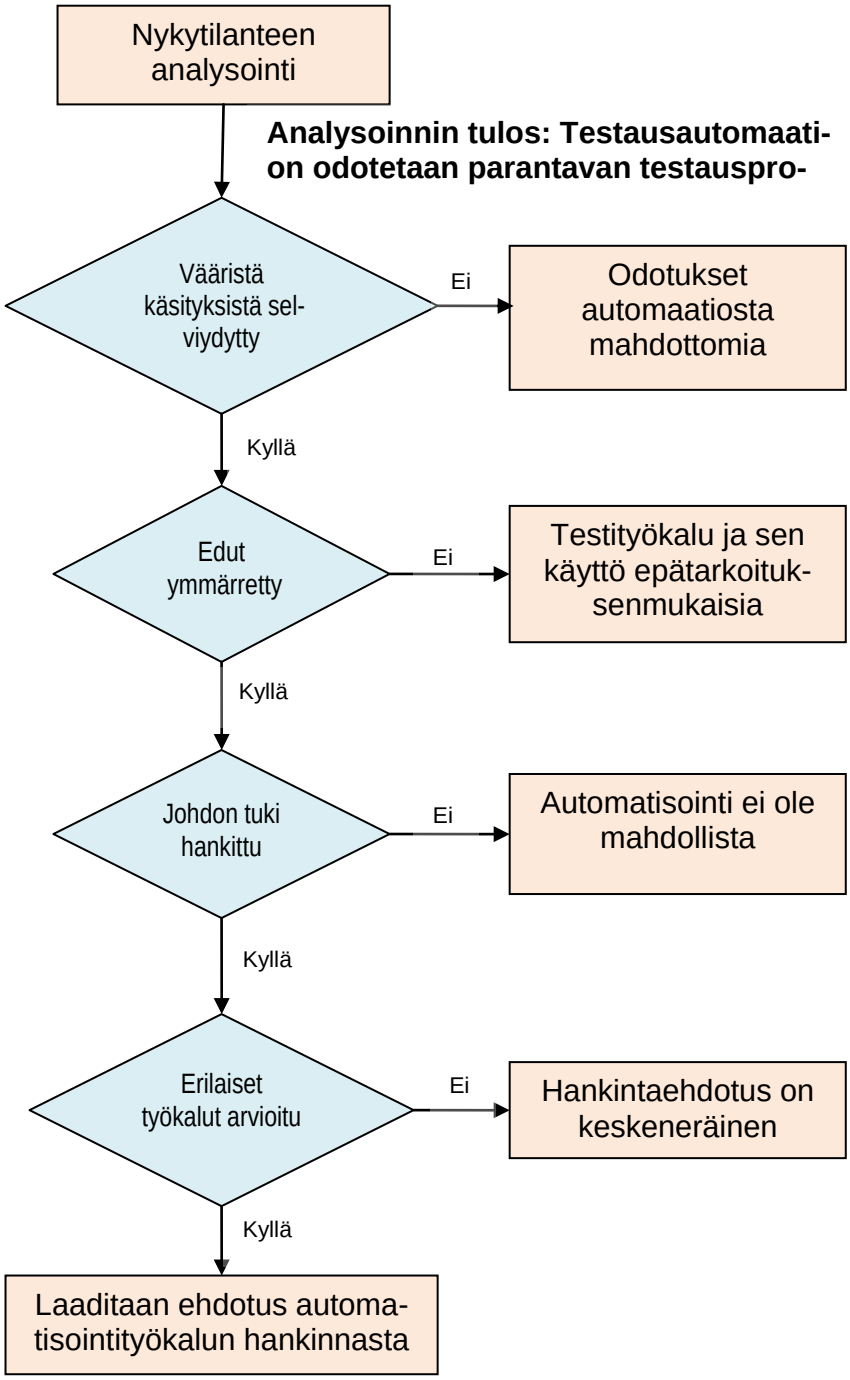
Automatisoidun testauksen elinkaari muodostuu kuudesta eri vaiheesta (kuva 9.1), joista ensimmäinen on päätös automatisoinnista. Kun päätös on saatu tehtyä, suoritetaan työkalun valinta. Seuraava vaihe on käyttöönottoprosessi, jossa analysoidaan nykyisiä testauskäytäntöjä ja mietitään, miten niitä voidaan kehittää niin, että ne tukisivat testiautomaatiota. Neljännessä vaiheessa suunnitellaan itse testaus, sen dokumentointi ja jatkokehittäminen. Seuraava vaihe on itse testien suorittaminen ja viimeisessä vaiheessa arvioidaan testauksen tulokset. Tätä prosessia voidaan toistaa uudelleen ja uudelleen. (Dustin ym. 2003.)



Kuva 9.1 Automatisoidun testauksen elinkaari (Dustin ym. 2003, s. 9)

Päätös automatisoinnista

Tässä vaiheessa päätetään, mitä aiotaan automatisoida. On esimerkiksi selvitetty, että testausprosessi ei ole riittävän tehokas, ja testauksen kehitystutkimus ehdottaa kokeiltavaksi testiautomaatiota. Vaiheittainen päätösprosessi on esitetty kuvassa 9.2.



Kuva 9.2 Automatisoinnin päätösprosessi (Dustin ym. 2003,s. 31)

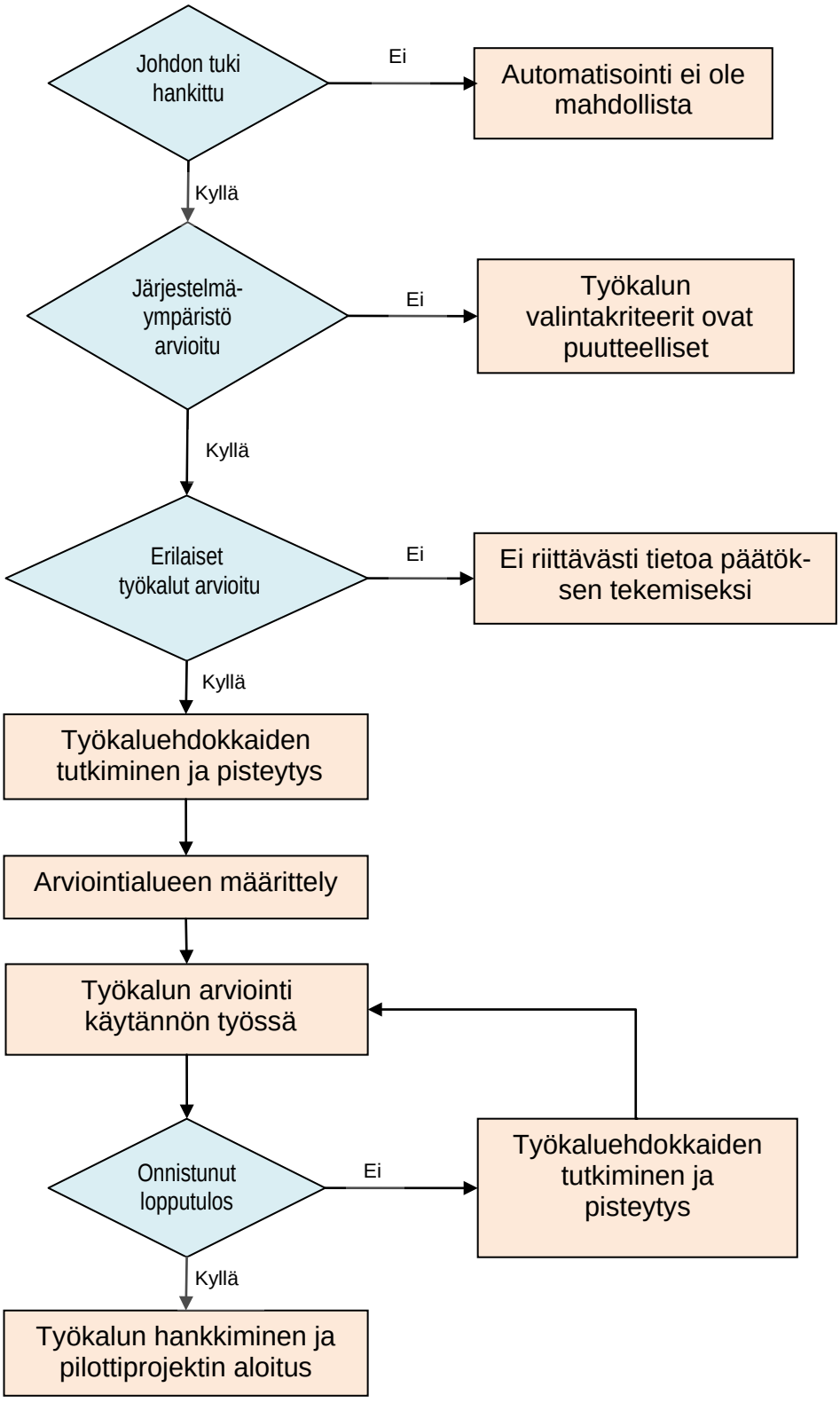
Testauksen automatisointiin liittyy usein erilaisia harhakäsityksiä. Suurimmat harhat ovat, että automatisoinnilla on mahdollista poistaa manuaalinen työ testausprosessista täysin, että testiautomaatiolla voidaan saavuttaa 100 % testikattavuus, ja että automatisoinnilla saadaan vähennettyä testauksen resurssivaatimuksia. Tosiasia on, että kaikkea ei voi automatisoida, vaan manuaalista työtä tarvitaan aina. Automaatiota ei kannata lähteä toteuttamaan kertaluontoisille testeille, vaan niissä manuaalinen testaus on nopeampaa ja halvempaa. Poikkeuksena tosin on testit, joita ei manuaalisesti pysty suorittamaan, kuten erilaiset kuormitustestit. Lisäksi automaatio ei mahdollista ”kaiken” testaamista, eivätkä testauksen resurssitarpeet vähene heti. Automatisoimalla pystytään saamaan aikaiseksi ajansäästöä, mutta vasta myöhemmissä testauksen vaiheissa, sillä suurimmat ajansäästöt testiautomaatio tarjoaa testien suorittamisessa ja raportoinnissa. (Dustin ym. 2003, s. 32–37.)

Vaikka automaatio vaatii tarkkaa harkintaa ja realiteettien ymmärtämistä, on oikein toteutetulla testiautomaatiolla (yhdistettynä manuaaliseen testaukseen) mahdollista saavuttaa lukuisia etuja: tärkeimpiä näistä ovat luotettavan järjestelmän syntyminen, testaustyön laadun paraneminen sekä testauksen ja testisuunnittelun työmäärän väheneminen. (Dustin ym. s. 2003, 37.)

Työkalun hankinta

Testaustyökalujen valinta jää yleensä tehtäväksi kauan sen jälkeen, kun testattavan sovelluksen kehitysympäristöstä ja – työvälineistä on jo päätetty. Parhaassa tapauksessa testaustyökaluksi voitaisiin valita jotain, mikä täyttäisi sekä kehitysympäristön vaatimukset että olisi sovitettavissa johonkin ohjelmistotuotantoprosessin elinkaaren alkuvaiheessa olevaan pilottiprojektiin. Todellisuudessa ohjelmistoprojektit ovat kuitenkin usein jo muilta osin yksityiskohtaisesti suunniteltuja, ennen kuin testaukseen aletaan ottaa kantaa. (Dustin ym. 2003, s. 68.)

Oli alkutilanne minkälainen tahansa, tulee testityökalun hankinnasta ja testaushenkilöstön koulutuksesta kustannuksia organisaatiolle joka tapauksessa. Parhaan vastineen saamiseksi rahoilleen tulisi testityökalun hankinnassa huolehtia siitä, että uusi työkalu on yhteensopiva koko organisaation järjestelmäympäristön kanssa. Järjestelmäympäristö käsittää ohjelmointialustat, -ohjelmistot, laitteistot ym., ja näiden huomioon ottaminen työkalun valinnassa vaatii järjestelmällistä lähestymistapaa (kuva 9.3). (Dustin ym, 2003, s. 68.)



Kuva 9.3 Automatisoidun testaustyökalun valintaprosessi (Dustin ym. 2003, s. 69)

Kun on saatu johdon tuki työkalun ja sen vaatimien resurssien hankkimiseksi, käy työkaluhankinnasta vastaava henkilö läpi järjestelmäympäristön, jotta varmistettaisiin valitun apuvälineen yhteensopivuus käytössä olevien käyttöjärjestelmien, ohjelmointikielien ja yrityksen käytäntöjen kanssa. Esiin tulevat kysymykset ja huomiot kirjataan ylös, ja laaditaan ”toivomuslista”, josta nähdään, minkälaisia ominaisuuksia työkalulta odotetaan. Työkalun hankinnasta vastaavan työnä on kohdentaa ominaisuudet ehdolla oleviin työkaluihin ja pisteyttää työkalut niiden arvioimiseksi. Pisteytettäviä piirteitä ovat esimerkiksi:

- helppokäyttöisyys
- muunneltavuus
- tuetut alustat
- monikäyttömahdollisuus
- virheenjäljitys
- työkalun toimivuus
- tulostuskapasiteetti
- suorituskyky
- yhteensopivuus eri versioiden työkalujen kanssa
- hinta
- toimittajan luotettavuus.

Työkalun, joka saavuttaa korkeimman pistemäärän, voidaan olettaa olevan kokeilemisen arvoinen, ja näin ollen voidaan tilata työkalun valmistajalta esittelytilaisuus, jossa siihen voidaan tutustua vielä paremmin. (Dustin ym. 2003.)

Vaikka työkalujen myyjät vakuuttavat tuotteidensa selviävän tilanteesta kuin tilanteesta, ei tähän kannata luottaa sokeasti, vaan työkalu kannattaa testata suunnitelmallisesti, mielellään rajatussa ympäristössä, kuten testilaboratoriossa. Työkalujen valmistajat tarjoavat tuotteistaan usein määräaikaisia kokeiluversioita, joten ostopäätöstä ei tarvitse tehdä ennen kuin tuote on käytännössä testattu. (Dustin ym. 2003.)

Työkalun rakentaminen ostamisen vaihtoehtona

Vaihtoehtona työkalun ostamiselle on kehittää työkalu itse. Kriteerit, joiden mukaan lähdetään miettimään, ostetaanko vai rakennetaanko testaustyökalu, ovat seuraavat: onko työkalua ylipäänsä saatavilla valmiina, onko tiedossa jotain merkittäviä etuja ostamisen ja rakentamisen välillä valitessa, ja kuinka päätös lopulta vaikuttaa sovelluskehityksen joustavuuteen pitkällä aikavälillä. (Pohjolainen, 2003.)

Toisinaan oletetaan, että työkalun rakentaminen on helppoa, koska siitä voidaan jättää kaupallisen sovelluksen hienoudet pois. Oletus on väärä, sillä ohjelmistoprojektit ovat aina haasteellisia, oli kyse tuotantosovelluksen tai jonkin ohjelmistotuotantoa tukevan sovelluksen kehittämisestä. On myös turha kuvitella, että parissa viikossa pystyy kehittämään sovelluksen, joka olisi parempi kuin markkinoilla oleva, kaupallinen sovellus, jonka tuotekehittelyyn on laitettu paljon rahaa ja aikaa, kun pelkästään valmiin työvälineen käytön opetteluun menee kokeneeltakin ammattilaiselta aikaa vähintään pari viikkoa. Monet ajattelevat myös, että jos työkalun kehittää itse, on helpompi vastata muutostarpeisiin joustavasti prosessin muuttuessa työn aikana. Kuitenkin, jos ei ole tietoa, mihin tarkoitukseen työkalua tullaan käyttämään, ei ole järkeä rakentaa, eikä myöskään ostaa työkalua. (Pohjolainen, 2003.)

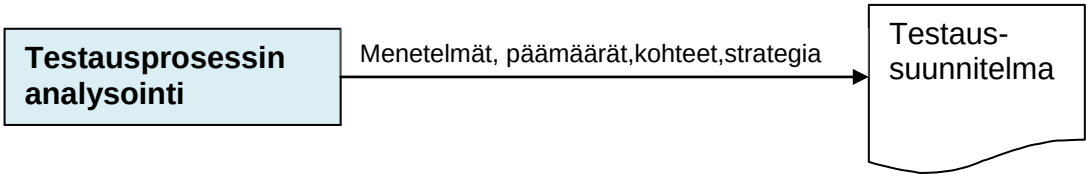
Ostaminen ei sinänsä ole sen helpompaa kuin rakentaminenkaan. Oli uusi tuote sitten itse kehitetty tai muualta ostettu, sen käyttöönotto aiheuttaa aina muutoksia, ja muutosvastarintaa tullaan kohtaamaan ihmisten taholta joka tapauksessa. Työkalun ostaminen ei myöskään automaattisesti merkitse sitä, että se tulisi käyttöön nopeammin kuin rakennettu työkalu: vaikka suunnittelu- ja ohjelmointiaikaa säästyy, menee muihin toimintoihin, kuten välineen valintaan, sijoittamiseen käyttöympäristöönsä ja käyttäjien kouluttamiseen, aikaa joka tapauksessa. On vielä mainittava, että testausvälineen ostaminen ei tarkoita sitä, ettei ylläpitotöitä tarvittaisi, sillä niitä tarvitaan aina. (Pohjolainen, 2003.)

Testiautomaation käyttöönottoprosessi

Testiautomaation käyttöönottoprosessissa (kuva 9.4) analysoidaan nykyinen testauskäytäntö ja etsitään sen kehityskohteita. Valittuun työkaluun tutustutaan lähtemällä liikkeelle testattavan sovelluksen vaatimuksista, tutkimalla sen kelvollisuus testattavan sovelluksen suhteen ja kokeilemalla työkalua käytännössä. Usein työkalu otetaan käyttöön liian vähäisellä etukäteissuunnittelulla. Se johtaa kertakäyttöisiin testiskripteihin ja tehottomaan testaukseen. Toinen yleinen virhe on työkalun ottaminen käyttöön liian myöhäisessä testauksen vaiheessa. (Dustin ym. 2003, s. 108–113.)

Kuva 9.4 Testiautomaation käyttöönottoprosessi (Dustin ym. 2003, s. 109)

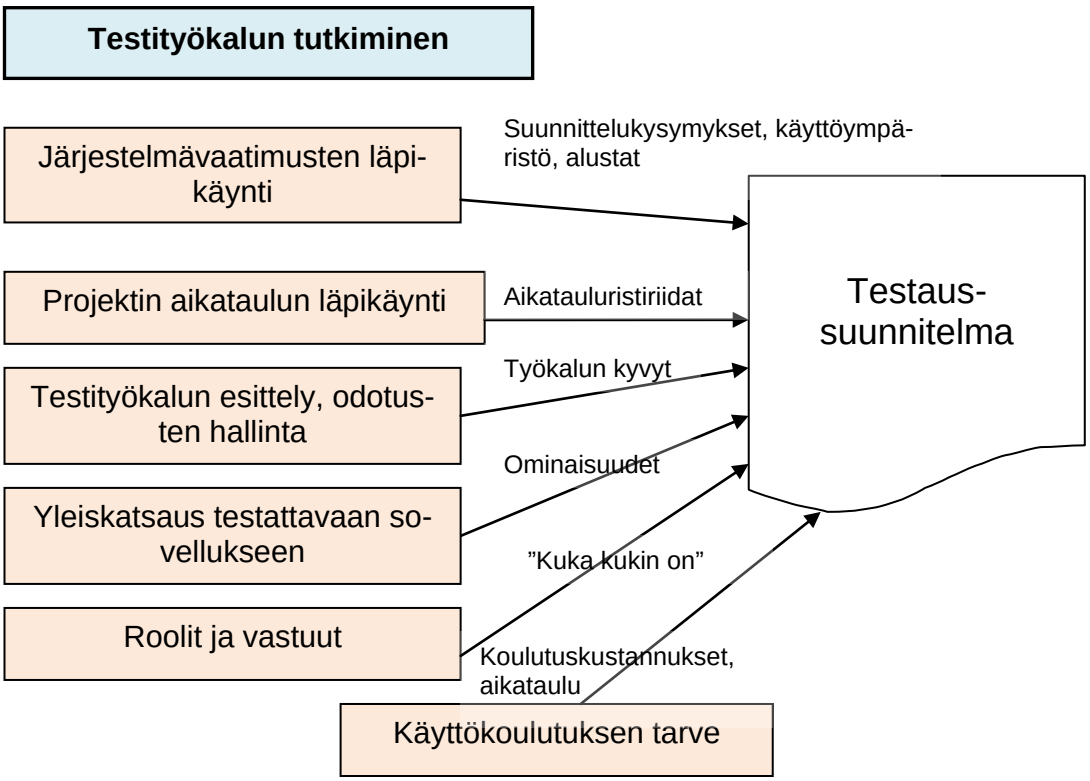
Testausprosessin analysointivaiheessa (kuva 9.5) analysoidaan käytössä olevat testauksen prosessit ja strategiat ja muokataan niitä tarvittaessa sellaisiksi, että ne mahdollistavat toimivan testiautomaation. Testausprosessin mitattavat ominaisuudet, kuten suorituskyky, toistettavuus, jäljitettävyys ja vakaus määritetään niin, että niitä voidaan käyttää hyväksi testausprosessin kehittämisessä. Testaustoiminnan tavoitteet, testauksen kohteet, testausstrategiat ja -prosessit määritellään ja ne dokumentoidaan. Myös testaustekniikat ja teknisen ympäristön vaatimukset testaustyökalulle määritellään, ja yksilöidään ne testit, joita testaustyökaluilla aiotaan suorittaa. (Dustin ym. 2003, 108-113).



Kuva 9.5 Testausprosessin analysointi (Dustin ym. 2003, s.112)

Vaihe on tärkeä, sillä testausprosessin dokumentoinnin avulla on tarkoitus selvittää testaustiimille ja muille asiasta kiinnostuneille tahoille, mitkä ovat testausprosessin päämäärät ja menetelmät. Dokumentointiin ja toteutukseen käytettävää aikaa ja kustannuksia usein kritisoidaan, mutta hyvin suunnitellun ja toteutetun testausprosessin on todettu maksavan itsensä moninkertaisesti takaisin ohjelmistotuotannon laadun paranemisena ja lyhentyneenä kehitysaikana. (Dustin ym. 2003, s. 108–113.)

Edellisessä elinkaarimallin vaiheessa testaustyökalun valinta tapahtui sen perusteella, mitkä ovat testityökalua käyttävän yrityksen yleisvaatimukset. **Testityökalun tutkimisvaiheen** (kuva 9.6) tarkoituksena on tutkia, onko valittu työkalu yhteensopiva testattavan sovelluksen kanssa. Testauspäällikön on tutustuttava testattavaan sovellukseen niin, että sen erityisvaatimukset ovat selvillä, ennen kuin johtopäätöksiä työkalun soveltuvuudesta ohjelmistoprojektiin voidaan tehdä.



Kuva 9.6 Testityökalun tutkiminen (Dustin ym. 2003, s. 134)

Työkaluun tutustuminen kannattaa suorittaa mahdollisimman ajoissa, jotta testaajat ehtisivät tutustua työkaluun kunnolla. Sovelluksen järjestelmävaatimukset selvitetään, jotta tiedettäisiin, minkälainen on käyttöliittymä ja kehitysympäristö. Testattava sovellusta tarkastellaan osa kerrallaan pohtien, minkä osien testaaminen kannattaa automatisoida. Kaikkia testauksen vaatimuksia ei voida toteuttaa automatisoinnilla, eikä kaikkea automatisoitavaa yhdellä testaustyökalulla. Vaatimukset on jaettava osajoukkoihin ja päätettävä, mikä työkalu on käyttökelpoisin juuri sen joukon vaatimuksiin. Roolitusosiossa selvitetään testausryhmän jäsenten kokemus työkaluista henkilöit-

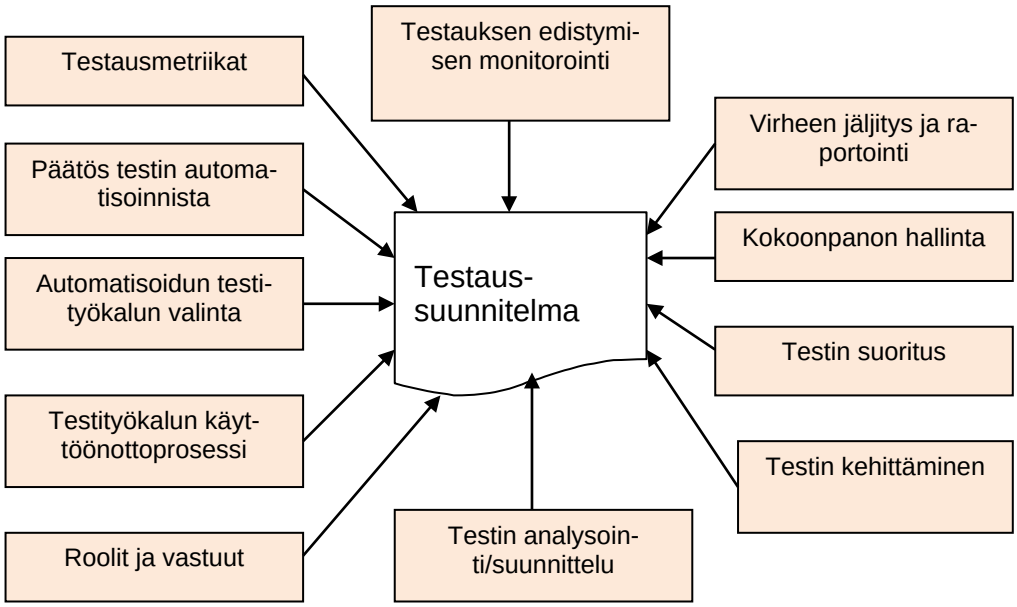
täin, se, kuinka paljon koulutusta työkalujen käytöstä kukin on saanut ja huomioidaan laajan kokemuksen omaavat henkilöt. Näiden tietojen perusteella voidaan arvioida koulutuksen ja harjoituksen tarve nykytilanteessa. (Dustin ym. 2003, 133–134.)

Testauksen suunnittelu ja kehitys

Jotta testauksen automatisointityökaluja pystyttäisiin käyttämään tehokkaasti, on testauksen suunnitteluun ja valmisteluun panostettava paljon (kuva 9.7). Testaussuunnitelman on sisällettävä paljon informaatiota, myös testauksen dokumentoinnin vaatimuksista projektissa. Testaussuunnitelman sisältönä on muun muassa:

- roolit ja vastuut
- testauksen aikataulutus
- testauksen suunnittelun ja testien suunnittelun toiminnot
- testauksen riskien kartoitus
- testauksen hyväksymiskriteerit.

Testaussuunnitelmaan voidaan liittää myös testausprosessit, nimeämiskäytännöt, testauksessa noudatettavat standardit ja testauksen jäljitysmatriisit.



Kuva 9.7 Testaussuunnitelman syötteet ja sisältö (Dustin ym. 2003, s. 193)

Myös testausympäristön asentaminen on osa testauksen suunnittelua. Ympäristön pystyttäminen vaatii suunnittelua ja hallintaa, ja testaustiimin tulee aikatauluttaa ja testausympäristön asennustehtävät, joita ovat esimerkiksi:

- testausympäristön laitteistojen asennus
- testausympäristön ohjelmistojen asennus
- tietoverkkojen asennus
- testausympäristön integrointi
- testitietokantojen hankinta
- ympäristön asennusskriptien kirjoittaminen
- testipetien skriptien kirjoittaminen.

Testauksen suunnittelun jälkeen siirrytään testien yksityiskohtaiseen suunnitteluun. (Dustin, 2001. Dustin, 2003.)

Testisuunnitelmasta käy ilmi suoritettavien testien määrä, testauksen lähestymistavat ja ehdot testien suorituksille. Tehokas testausprosessi, johon sisältyy testiautomaatiota, sisältää jo itsessään pienen ohjelmistokehityssyklin strategian ja päämäärän suunnitteluineen, testivaatimusten määrittelyineen, suunnitteluineen ja toteutuksineen. Kuten ohjelmistokehityksessäkin, pitää testausprojektissa testauksen vaatimukset olla määriteltyinä ennen kuin testejä aletaan suunnitella. Testauksen vaatimukset pitää olla selkeästi määriteltyinä ja dokumentoituina, että kaikki projektiin osallistuvat ymmärtävät, mitkä ovat testauksen tavoitteet. Jotta automatisoidut testit olisivat uudelleenkäytettäviä, toistettavia ja huollettavissa, on testien kehittämiselle määriteltävä standardit ja noudatettava niitä myös. Kun testikehityksen prosessit ovat kunnossa, voidaan siirtyä testien kirjoittamiseen. Kuvassa 9.8 on esimerkki siitä, miten testaustehtävien ja ohjelmistokehityksen tehtävät ajoittuvat. (Dustin, 2001.)

Vaihe	Kehitysprosessi	Testausprosessi
Yksikkötestaus	Kehitä moduuli vaatimusten perusteella	Suorita testauksen suunnittelu ja testiympäristön asennus
	Käännä moduuli	Kirjoita testiskriptit tai nauhoita testiskenaarioa moduulia käyttäen
	Yksikkötestaa moduuli	Käännä automatisoitu testiskripti ja testaa moduuli sillä. Käytä työkaluja, jotka tukevat yksikkötestausta
	Korjaa havaitut viat	Aja uudelleen testiskripti regressiotestin suorittamiseksi, kunhan havaitut viat on korjattu
	Suorita toiminnallisuustestaus	Verifioi, että järjestelmä on skaalattavissa ja että se toteuttaa toiminnalliset vaatimuksensa.
Integrointitestaus	Rakenna järjestelmä yhdistämällä moduulit, integrointitestaa yhdistetyt moduulit, katselmoi virheraportit	Yhdistä yksikkötestiskriptit ja lisää uusia skriptejä, jotka demonstroivat moduulien yhteistoimintaa. Käytä testityökalua joka tukee integraatiotestausta
	Korjaa havaitut viat ja päivitä virhestatus	Aja uudelleen testiskripti regressiotestauksen suorittamiseksi, kunhan havaitut viat on korjattu
	Jatka toiminnallisuustestausta	Verifioi, että järjestelmä on skaalattavissa ja että se toteuttaa toiminnalliset vaatimuksensa.
Järjestelmätestaus	Katselmoi virheraportit	Integroi automatisoidut testiskriptit järjestelmätason testausprosesseihin niiltä osin kuin se on mahdollista, ja kehitä varsinaiset järjestelmätason testausprosessit. Suorita järjestelmätestaus ja nauhoita testitulokset.
	Korjaa havaitut viat ja päivitä virhestatus	Aja uudelleen testiskriptit regressiotestauksen suorittamiseksi, kunhan havaitut viat on korjattu
Hyväksymistestaus	Katselmoi havaintoraportit	Suorita osa järjestelmätestauksen testeistä osana käyttäjän hyväksymistestausta
	Korjaa havaitut viat	Aja uudelleen testiskriptit regressiotestauksen suorittamiseksi, kunhan havaitut viat on korjattu

Kuva 9.8 Ohjelmistokehityksen ja testauksen tehtävien ajoittuminen (Dustin 2001)

Testien suoritus ja ylläpito

Kun testaussuunnitelma on valmis ja testausympäristö toimintakunnossa, on aika suorittaa testit. Testaustiimi suorittaa suunnitellut yksikkö- , integrointi-, järjestelmä- ja hyväksymistestit pyrkien pysymään suunnitellussa aikataulussa. Kun nämä vaiheet on käyty läpi, ja testaussuunnitelmassa määritellyt hyväksymiskriteerit on saavutettu, on koko järjestelmän testaus suoritettu. (Dustin, 2003.)

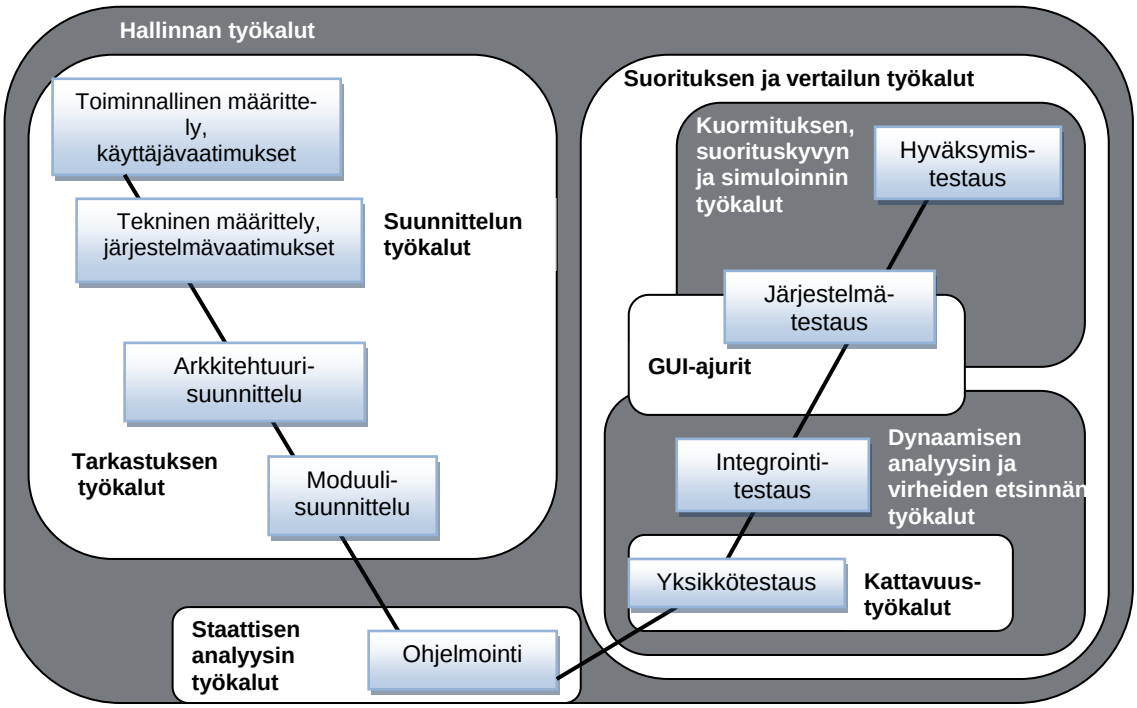
Jotta testauksen laatua voitaisiin arvioida testauksen edistyessä, on testaustiimin kerättävä ja analysoitava tietoa testauksen löytämistä vioista. Löydettyjen vikojen vertaaminen suoritettujen testien määrään on yksi tapa tutkia testauksen tehokkuutta. Jos vikojen määrä osoittautuu korkeaksi, voi olla tarpeen muuttaa testaussuunnitelmaa tai testauksen aikataulua (Dustin ym. 2003, s. 350.) Testaussuunnitelmaa saatetaan joutua muuttamaan myös testauksesta riippumattomista syistä, lähinnä testattavan sovelluksen vaatimusmäärittelyiden muutoksien takia. Nämä muutokset ovatkin yksi niitä ominaisuuksia, jotka eniten vaikeuttavat testauksen hallintaa.

Testausprosessin tarkastelu ja kehittäminen

Elinkaaren viimeisenä osana on testausprosessin arviointi ja kehittäminen. Tässä vaiheessa testaustiimin keräämät tiedot testauksen edistymisestä kerätään yhteen ja katselmoidaan sen tutkimiseksi, kuinka hyvin testaus lopulta vastasi suunnitelmia. On myös syytä kirjata ylös onnistumiset ja toimiviksi todetut käytännöt, jotta niitä voitaisiin hyödyntää tulevissa testausprojekteissa.

9.3Testaustyökalujen tyypit

Testaustyökaluja on kehitetty kaikille testauksen V-mallin tasoille. Kuvassa 9.9 on esitetty työkalujen sijoittuminen. Seuraavaksi käydään läpi, minkälaisia työkaluja näihin kategorioihin kuuluu.



Kuva 9.9 Testaustyökalujen tyypit (mukaillen Pohjolainen, 2003)

9.3.1 Hallinnan työkalut

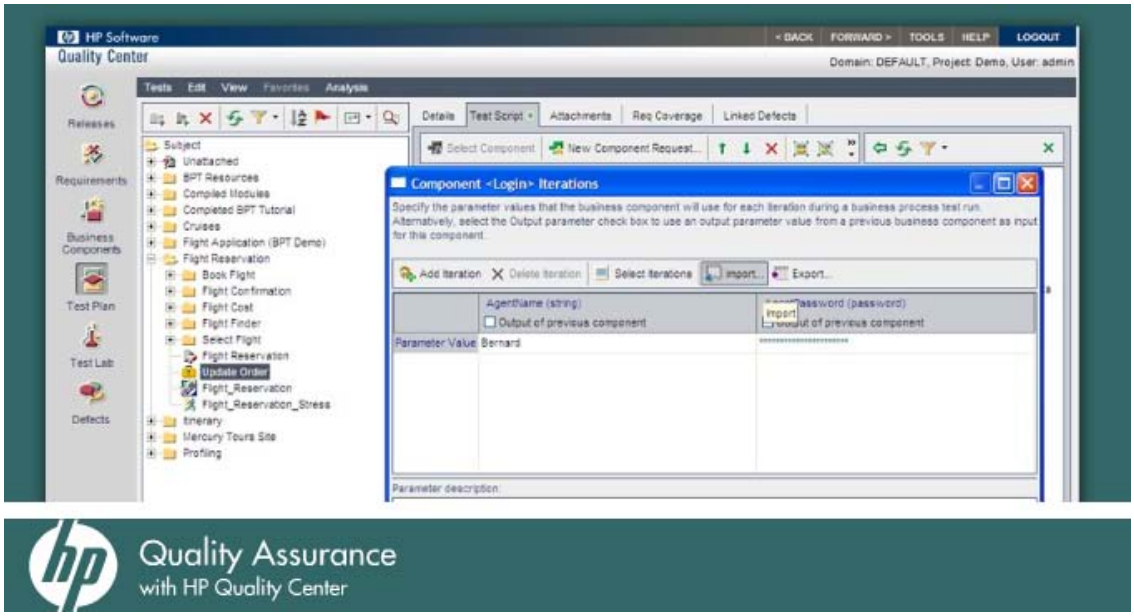
Hallinnan (management) työkaluilla tarkoitetaan kaikkia niitä työkaluja, jotka liittyvät testausprosessin hallintaan. Hallintatyökalut, jotka kattaisivat kaikki testauksen osa-alueet, ovat vasta kehitteillä, ja toisaalta testauksen hallinta voi käyttää samoja projektinhallintatyökaluja kuin muukin ohjelmistoprojekti. Näitä työkaluja ovat erilaiset versionhallintajärjestelmät ja ajanhallintajärjestelmät. WWW-pohjaisten hallintajärjestelmien käyttö mahdollistaa sen, että kaikki testausta käsittelevä tieto on ajasta ja paikasta riippumatta kaikkien projektiin kuuluvien henkilöiden käytettävissä. Testauksen hallinnan työkalut automatisoivat

testauksen suunnittelua, testitulosten analysointia, testauksen dokumentointia sekä raportointia. ***Testauksen hallinnan*** työvälineitä ovat**:**

HP Quality Center (ent. Mercury TestDirector)
Yleisin käytössä oleva kaupallinen testauksen hallinnan työkalu, joka kattaa

- vaatimusten hallinnan
- testitapausten suunnittelun
- testitapausten ajon
- virheidenkäsittelyn
- raportoinnin.

Ohjelman kotisivulta löytyy demo, joka vaatii rekisteröitymisen näkyäkseen (kuva 9.10):
https://h30406.www3.hp.com/campaigns/2007/promo/1-496MR/index.php?mcc=CUFJ



Kuva 9.10 HP Quality Center

Ohjelmasta on saatavilla myös trial-versioita sen kotisivuilta:

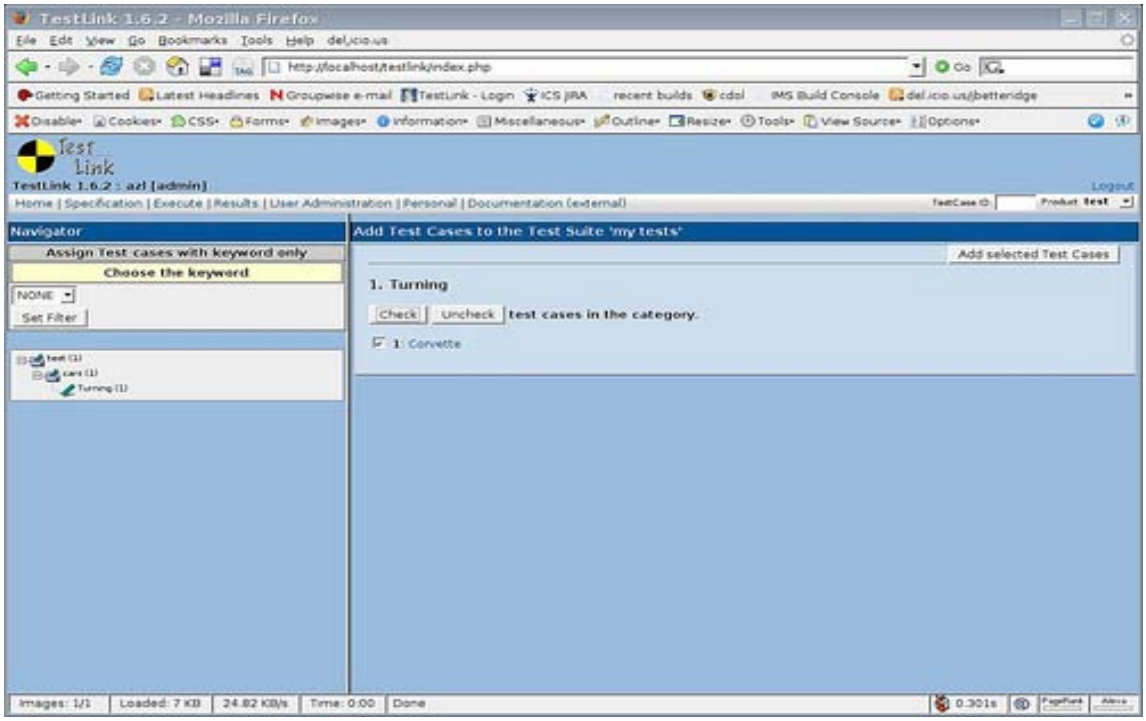
https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24%5E1131_4000_16__

TestLink

Yksi suosituimmista avoimen lähdekoodin sovelluksista, joka kattaa testien ja testitapausten

- suunnittelun
- hallinnan
- organisoinnin
- priorisoinnin
- edistymisen seurannan
- tulosten seurannan.

Ohjelman kotisivu on:
http://testlink.sourceforge.net/docs/testLink.php
Kuvassa 9.11 on TestLinkin käyttöliittymä:



Kuva 9.11 TestLink

Borland: SilkCentral Test Manager

Borlandin testauksenhallinnan ohjelmisto:
http://www.borland.com/us/products/silk/silkcentral_test/index.html

IBM Rational Quality Manager

Työvälineistö projektinhallintaan, IBM:n laajasta Rational-tuoteperheestä:.
http://www-01.ibm.com/software/awdtools/rqm/

QADirector

Microfocuksen testauksen hallintaan tarkoitettu ohjelmisto:
http://www.microfocus.com/products/QADirector/ProductPreview.asp

Vaatimusten hallintaan tarkoitettuja työvälineitä ovat:

Rational DOORS

http://www-01.ibm.com/software/awdtools/doors/productline/

Borland CaliberRM

http://www.borland.com/us/products/caliber/index.html

Tiger Pro

Olio-ohjelmien vaatimustenhallintaa. UniSA:n (University of South Australia) laatima ilmaisohjelma:
http://www.seecforum.unisa.edu.au/SEECTools.html

Softreq

Selainpohjainen ilmaisohjelma:
http://www.softreq.com/index.cfm

Havaintojen hallinnan työkalu:

Testopia, Bugzillan lisäosa, sisältää testitapausten ja virheraporttien keskitetyn hallinnan ja seurannan, ja sillä varustettuna Bugzillaa voitaisiin luonnehtia myös testauksen hallinnan työkaluksi.
http://www.mozilla.org/projects/testopia/

Kokoonpanon hallinnan työvälineitä ovat:

Rational ClearCase
http://www-01.ibm.com/software/awdtools/clearcase/

Borland StarTeam
http://www.borland.com/us/products/starteam/index.html

SmartFrog
Avoimen lähdekoodin ohjelma Javalle:
http://wiki.smartfrog.org/wiki/display/sf/SmartFrog+Home

OCS Inventory NG
Avoimen lähdekoodin ohjelma (Perl, PHP, C++)
http://www.ocsinventory-ng.org/

Seuraavat työkalut ovat ***muutoksen hallinnan*** välineitä:

Rational Change
http://www-01.ibm.com/software/awdtools/change/features/

Borland StarTeam
A Complete Software Change & Configuration Management (SCCM) Tool
http://www.borland.com/us/products/starteam/index.html

AllChange
http://www.intasoft.net/products.asp

9.3.2 Suunnittelun ja tarkastuksen työkalut

Suunnittelun työkaluja ovat työkalut, joilla voidaan mallintaa ohjelmistoja. Esimerkiksi NetBeansIDE:tä on mahdollista laajentaa UML-lisäosalla, jolla voidaan mallintaa rakenteilla oleva järjestelmä. UML-mallin perusteella voidaan generoida koodia, ja mallista voidaan johtaa testitapauksia. Muita mallinnusvälineitä:

Rational Rose
http://www-01.ibm.com/software/awdtools/developer/rose/index.html

Borland Together
http://www.borland.com/us/products/together/index.html

StarUML
Avoimen lähdekoodin mallinnustyökalu:
http://staruml.sourceforge.net/en/

Tarkastuksen työkalut sisältävät erilaisia tarkastuskäytäntöjä helpottavia, usein WWW-pohjaisia sovelluksia, jotka mahdollistavat esimerkiksi fyysisesti eri paikoissa olevien henkilöiden osallistumisen katselmointiin:

CodeCollaborator
Maksullinen ohjelma, josta on mahdollisuus saada ilmaislisenssi avoimen lähdekoodin projekteihin. Sivuilta löytyy myös demo ohjelman toiminnasta:
http://smartbear.com/docs/viewlets/CodeCollabDemo/CodeCollabDemo.html

Codestriker
Avoimen lähdekoodin työkalu katselmointien ja tarkastuksien avuksi:
http://codestriker.sourceforge.net/

9.3.3 Staattisen analyysin työkalut

Staattiselle analyysille löytyy paljon työvälineitä ohjelmointiympäristöistä, kuten Visual Studiosta, NetBeansista ja Eclipsestä. Luvussa 9.2.9 kerrotaan näistä enemmän. Edellä mainittujen lisäksi on muita, staattisen analyysin eri osa-alueisiin painottuvia ohjelmistoja:

Flawrinder
Tietoturvaheikkouksia tutkiva avoimen lähdekoodin ohjelma C- ja C++-kielille.
http://www.dwheeler.com/flawfinder/

Splint
C-kielen tarkastukseen tarkoitettu avoimen lähdekoodin ohjelma, joka etsii tietoturvaongelmia ja koodausvirheitä:
http://www.splint.org/

PC-lint
C- ja C++-kielien staattisen analyysin työkalu, joka etsii koodausvirheitä
http://www.gimpel.com/html/lintspec.htm

CodeCheck
Kaupallinen ohjelmisto, joka tutkii standardinmukaisuutta, koodin kompleksisuutta, siirrettävyyttä ja ylläpidettävyyttä:
http://www.codecheck.com/cc/index.html

Java-koodin staattinen analyysi

Eri ohjelmointikieliin on saatavissa ohjeistuksia hyvien ohjelmointitapojen noudattamiseksi. Esimerkiksi Javalle se löytyy täältä:
http://java.sun.com/docs/codeconv/

Työkalu, joka käyttää ohjeistuksia hyväkseen, on esimerkiksi **Checkstyle:**
http://checkstyle.sourceforge.net/index.html

Checkstyle:n standardikokoonpanossa se tukee Java Code Conventionsia, mutta se on muokattavissa tarkastamaan muidenkin kielien ulkoasua ja ”parhaita käytäntöjä”. Checkstyle on mahdollista asentaa lisäosana ohjelmointiympäristöihin, kuten NetBeansiin tai Eclipseen. On myös ohjelmia, jotka muokkaavat koodia hyvien ohjelmointitapojen mukaisiksi, kuten **Jacobe Code Beautifier**, jonka voi myös asentaa Eclipsen lisäosaksi:
http://www.tiobe.com/index.php/content/products/jacobe/Jacobe.html

On sovelluksia, jotka staattisen analyysin menetelmin tutkivat Java-koodista yleisiä bugeja. **FindBugs** on ohjelma, jonka myös saa asennettua Eclipsen lisäosaksi tai jota voi ajaa komentoriviltä:
http://findbugs.sourceforge.net/downloads.html

9.3.4 Dynaamisen analyysin ja virheiden etsinnän työkalut

Dynaamisen analyysin työkalut tarjoavat ajonaikaista tietoa testattavasta ohjelmistosta ja sen resurssien käytöstä. Dynaaminen analyysi on laajimmin automatisoitu testauksen osa-alue, ja työkaluja on paljon. ***Muistin, suorituskyvyn ja kattavuuksien*** analysointiin tarkoitettuja ohjelmia ovat muun muassa:

Rational Purify
Työkalu, joka etsii ajonaikaisia muistin käytön virheitä:
http://www-01.ibm.com/software/awdtools/purify/

Rational PurifyPlus
Muistivirheiden lisäksi suorituskyvyn testaaminen ja raportointi sekä koodikattavuudet ja testaamattoman koodin etsiminen:
http://www-01.ibm.com/software/awdtools/purifyplus/

DevPartner Studio ja DevPartner
Visual Studioon integroitavissa oleva dynaamisen analyysin työkalu, joka analysoi muun muassa ajonaikaisia virheitä, suorituskyvyn ongelmakohtia, tietoturvallisuutta ja kattavuuslukuja:
http://microfocus.com/products/DevPartner/StudioProfessionalEdition.asp

Project Analyzer
Kaupallinen työkalu, jossa on paljon ominaisuuksia, kuten kuolleen koodin etsiminen, muistivirheiden etsiminen, koodin optimointi, ohjelmointistandardien noudattaminen jne., ja kattavat raportointiominaisuudet:
http://www.aivosto.com/project/project.html

Coverity Integrity Center
Java-ympäristön analysointityökalupaketti, joka sisältää staattisen analyysin, arkkitehtuurin analyysin ja versioinnin analyysin lisäksi dynaamisen analyysin työkalut. Integroitavissa Eclipseen:
http://www.coverity.com/products/dynamic-analysis.html

Valgrind

Avoimen lähdekoodin ohjelma Linux-ympäristöön, sopii myös testiympäristöjen luomiseen:
http://valgrind.org/

Vaikka ohjelmointiympäristöjen debuggerit eivät ole varsinaisia testaustyökaluja, on ne syytä mainita ***virheiden etsinnän työkaluja*** listattaessa. Debuggerin avulla ohjelman suoritus on mahdollista pysäyttää tarkistuspisteeseen ja suorittaa siitä eteenpäin joko rivi kerrallaan tai tarkistuspisteestä seuraavaan tarkkaillen samalla muuttujien arvoja jne. Näiden lisäksi on paljon erilaisia virheenhallinnan ja – jäljityksen työkaluja, sekä kaupallisia että avoimen lähdekoodin työkaluja. Seuraavaksi luetellaan muutamia niistä:

Bugzilla
Avoimen lähdekoodin ohjelma, joka on virheiden raportoinnin työväline. Laajennettavissa Testopia-lisäosalla, joka on esitelty hallinnan työkalujen yhteydessä:
http://www.bugzilla.org/

Mantis
Suosittu, selainpohjainen avoimen lähdekoodin virheenjäljitysohjelmisto:
http://www.mantisbt.org/

BugRat
Java-pohjainen virheiden jäljityksen ja raportoinnin työkalu:
http://www.gjt.org/pkg/bugrat/

BUGtrack
Kaupallinen, selainpohjainen virheenjäljitysohjelmisto:
http://www.bugtrack.net/

BugTracker.NET

Ilmaisohjelma .NET-ympäristöön:
http://ifdefined.com/bugtrackernet.html

9.3.5 Kattavuustyökalut

Kattavuustyökalut voidaan jakaa kahteen ryhmään: toiset mittaavat koodin kattavuuslukuja, toiset taas testauksen kattavuutta. Erityisesti Java-ympäristöön löytyy paljon avoimen lähdekoodin kattavuustyökaluja:

Quilt

Avoimen lähdekoodin kattavuustyökalu Java-ympäristöön:
http://quilt.sourceforge.net/

Emma

Avoimen lähdekoodin kattavuustyökalu Java-ympäristöön. NetBeansille on Emmaan perustuva lisäosa CodeCoverage, Eclipselle taas varsinainen Emma-lisäosa:
http://emma.sourceforge.net/samples.html

Jester

Avoimen lähdekoodin työkalu, joka etsii koodia, jota ei ole testattu. Se siis ei ole koodikattavuus- vaan testikattavuustyökalu, ja se on tarkoitettu Java-kielelle, JUnit-testien testaamiseen. Jester käyttää automaattista virheenkylvämistä kattavuutta laskiessaan. Jesteristä on myös versiot Pythonille ja PyUnit-testeille (Pester) sekä Nester C#:lle :
http://jester.sourceforge.net/

Hansel

JUnitin laajennusosa, joka tutkii, paljonko koodista, jonka testien on ollut tarkoitus kattaa, itse asiassa on katettu:
http://hansel.sourceforge.net

Kattavuustyökaluja muille kuin Javalle, on esimerkiksi Rational PurifyPlus, joka on esitelty luvussa 9.2.4. Laajoissa, kaupallisissa ohjelmistoissa kattavuustyökalut ovatkin osa analyysityökalujen toimintoja. Näiden lisäksi on toki puhtaita kattavuustyökaluja kuten:

PartCover

Testikattavuustyökalu .Net-kielille. Käytetään yhdessä NUnit:in kanssa:
http://www.nunit.org/index.php
http://sourceforge.net/projects/partcover/

GTC

Avoimen lähdekoodin kattavuustyökalu C- ja C++-kielille:
http://www.exampler.com/testing-com/tools.html

9.3.6 GUI-ajurit

Tähän ryhmään kuuluvat työkalut, joilla voidaan automatisoida graafisen käyttöliittymän testausta. Niiden toiminta perustuu GUI-karttojen nauhoittamiseen. GUI-kartalla tarkoitetaan tiedostoa, joka kuvaa graafisen käyttöliittymän komponenttien sijainteja sovellusikkunassa. Laajat testiautomaatiosovellukset, kuten HP QualityCenter, sisältävät kuvatun kaltaisia toiminnallisuuksia. Lisäksi on olemassa sekä kaupallisia että avoimen lähdekoodin sovelluksia pelkästään graafisen käyttöliittymän testaamiseen. Nauhoitettujen käyttöliittymätestitapausten hyödyllisyydestä voidaan olla montaa mieltä: usein käyttöliittymän komponenttien sijainti muuttuu, ja testit pitää nauhoittaa uudestaan ja uudestaan, mutta toisaalta, nauhoituksia voidaan käyttää testattavan ohjelmiston toiminnan opettelemiseen. Esimerkkejä GUI-ajureista:

Selenium

Web-sovelluksen käyttöliittymän testauksen automatisointityökalu, joka on avoimen lähdekoodin ohjelma. Selenium Remote-Control mahdollistaa myös samojen testien ajamista useilla eri selaimilla:
http://seleniumhq.org/
Selenium on paljon käytetty työkalu, joten tutoriaaleja ja käyttöohjeita löytyy:
http://agilesoftwaredevelopment.com/videos/functional-testing-selenium-ide

Watin

C#:lla kirjoitettu, avoimen lähdekoodin ohjelma web-käyttöliittymän testaukseen:
http://watin.sourceforge.net/

GUIDancer

Kaupallinen ohjelma, joka on suunnattu erityisesti Java-käyttöliittymien testaukseen, mutta myös HTML-käyttöliittymien testaus onnistuu:
http://www.bredex.de/en/guidancer/first.html

Marathon

Avoimen lähdekoodin ohjelma, joka on tarkoitettu Java-ympäristöön:
http://www.marathontesting.com/Home.html

9.3.7 Kuormituksen, suorituskyvyn ja simuloinnin työkalut

Tähän ryhmään kuuluvia työkaluja käytetään testattaessa järjestelmän suorituskykyä erilaisissa stressi- ja kuormitustilanteissa. Suurin osa työkaluista on tarkoitettu web-ympäristöön. Seuraavassa luvussa esitellyt tietoturvatestauksen työkalut voitaisiin myös lukea suorituskykytestauksen työkaluiksi.

HP LoadRunner

Kaupallinen kuormitustestaustyökalu, josta on saatavilla kokeiluversio:
https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17%5E8_4000_100__

OpenSTA

Avoimen lähdekoodin kuormitustestaustyökalu web-sovelluksille. Testejä voidaan tallentaa, joten työkalua voidaan käyttää myös regressiotestien laatimisessa:
http://portal.opensta.org/index.php

Pylot

Avoimen lähdekoodin Web Service-testausohjelma, jolla voidaan tehdä toiminnallisuus- ja skaalautuvuustestauksia:
http://www.pylot.org/

JMeter

Java-sovellusten testaamiseen laadittu, avoimen lähdekoodin työpöytäsovellus, joka alun perin on tarkoitettu web-sovellusten käyttäytymisen ja suorituskyvyn testaamiseen, mutta joka on myöhemmin laajennettu muihinkin testaustarkoituksiin. Sovelluksella voidaan simuloida raskasta kuormitusta palvelimella, tietoverkossa tai muulla testauksen kohteella ja analysoida suorituskykyä erilaisen rasituksen alaisena.
http://jakarta.apache.org/jmeter/

OpenLoad

Helppokäyttöinen web-testausväline, josta on saatavilla ilmainen kokeiluversio ja käyttöohjeita videomuodossa. Mahdollistaa testien luomisen selainpohjaisesti, ilman testiskriptien kirjoittamista:
http://www.opendemand.com/openload/

Dieseltest

Helppokäyttöinen testausväline, jolla voidaan simuloida satojen tai tuhansien käyttäjien samanaikaisuutta web-sovelluksessa. Ominaisuuksina skriptien nauhoitus ja uudelleen suorittaminen sekä tulosten reaaliaikainen näyttö. Tulokset voidaan myös viedä raporttiin myöhempää analysointia varten:
http://sourceforge.net/projects/dieseltest/

Holodeck

Kaupallinen testaustyökalu, joka simuloi virheenkylvämistekniikkaa käyttäen Windows-ympäristön sovellus- ja järjestelmävirheitä. Ohjelman kotisivuilta on ladattavissa ilmainen kokeiluversio:
http://www.securityinnovation.com/holodeck/index.shtml

9.3.8 Suorituksen ja vertailun työkalut

Suorituksen ja vertailun työkalut ovat testien suorittamisen automaation työvälineitä. Toisinaan on vaikea vetää rajaa, mitkä työkalut tähän kategoriaan kuuluvat, sillä alue on hyvin laaja, kattaen koko testauksen V-mallin oikean haaran kokonaan: esimerkiksi GUI-ajurit kuuluvat osaltaan myös tähän ryhmään. Muita tähän kategoriaan kuuluvia työkaluja ovat esimerkiksi yksikkötestaustyökalut, näytön kaappaus- ja nauhoitussovellukset, SOA-testerit ja vertailuvälineet. Myös testisyötteiden valinnan avustusohjelmat, virheidenkylvämissovellukset sekä asennuksen testauksen ja tietoturvatestauksen välineet voidaan katsoa kuuluviksi tähän ryhmään. Seuraavaksi esitellään erilaisia suorituksen ja vertailun työkaluja.

Yksikkötestaustyökalut on yleisesti integroitu IDE:ihin. Työkaluja on kirjoitettu eri ohjelmointikielille, ja varsinaisten yksikkötestaustyökalujen lisäksi tähän ryhmään voidaan sisällyttää myös Mock-objektien luomiseen tarkoitetut työkalut.

Ahven
Ada-kielen yksikkötestaustyökalu:
http://ahven.sourceforge.net/

Check
C-kielen yksikkötestaustyökalu:
http://check.sourceforge.net/

CppUnit
C++-kielen yksikkötestaustyökalu:
http://sourceforge.net/projects/cppunit/files/cppunit/1.12.1/

Hippo Mocks
Mock-kirjasto C++:lle:
http://www.assembla.com/wiki/show/hippomocks

JUnit
Java-kielen yksikkötestaustyökalu:
http://www.junit.org/

JMock
Java-kielen Mock-objektikirjasto:
http://www.jmock.org/

SOA-testerit ovat työkaluja, joita käytetään Web Service-rajapintojen testaukseen. Ne automatisoivat rajapintoja ja generoivat niiden perusteella ”testihaarniskan”.

Rational Tester for SOA Quality

Kaupallinen toiminnallisen testauksen ja regressiotestauksen työkalu, joka mahdollistaa käyttöliittymättömien Web Service-rajapintojen testaamisen ilman ohjelmointia:
http://www-01.ibm.com/software/awdtools/tester/soa/

Eviware soapUI

SoapUi on suosittu ja monipuolinen ilmaisohjelma REST/WADL ja SOAP/WSDL-pohjaisten Web Service-palveluiden testaamiseen. Ohjelma on integroitavissa NetBeansiin:
http://www.eviware.com/index.php?option=com_content&task=view&id=143&Itemid=76

Testmaker

Monipuolinen avoimen lähdekoodin ohjelma, jonka yhtenä ominaisuutena on SOAP-pohjaisten Web Service-palveluiden testaus:
http://www.pushtotest.com/

Tietoturvatestauksen automatisointiin on olemassa erilaisia ohjelmia, joilla on mahdollista analysoida sovelluksen tietoturvaa. Työkalut on ensisijaisesti tarkoitettu verkkosovellusten analysointiin. Seuraavassa esimerkkejä työkaluista:

Nessus

Yksityiskäyttöön ilmainen, tehokas, ajan tasalla pysyvä ja helppokäyttöinen tietoturvaskanneri. Se tarkastaa annetun verkko-osoiteavaruuden ja selvittää, onko kohteeseen mahdollista tehdä tietomurto tai käyttää sitä muuten vääriin tarkoituksiin.

http://nessus.org/nessus/features/

Ethereal

Verkkoprotokollien analysointiohjelma, jolla on mahdollista testata verkon tietoturva-aukkoja, siepata tietoa kuten käyttäjätunnuksia, salasanoja, käytettyjä protokollia ja tietoa kulkevan datan määrästä ja laadusta. Se mahdollistaa pakettien selaamisen reaaliaikaisesti verkosta ohjelman nauhoittaessa sieppaustiedostoon avoimen verkkoyhteyden tapahtumat.

http://www.ethereal.com/

Snort

Suosittu, avoimen lähdekoodin tunkeilijan havaitsemisjärjestelmä (Intrusion Detection System, IDS)

http://www.snort.org/

Erilaisia ***vertailuvälineitä*** voidaan käyttää testauksen tukena. Niitä ovat esimerkiksi tiedostorakenteiden vertailutyökalut, dokumenttien sisällön vertailutyökalut ja tietokantojen vertailutyökalut.

ComparatorPr

Kaupallinen tiedostojen vertailutyökalu, josta löytyy shareware-versio. Helpottaa eri sijainneissa olevan datan synkronointia ja varmuuskopiointia.
https://softbytelabs.com/us/cpp/index.html

Diff Doc

Työkalu, jolla voidaan vertailla dokumenttien sisältöä. Ohjelman kotisivuilta löytyy sekä esittelyvideo että ilmainen kokeiluversio ladattavaksi:
http://www.softinterface.com/MD/Document-Comparison-Software.htm

CompareDatabase

Työkalu, jolla voidaan verrata tietokantojen rakennetta ja sisältöä. Toimii Access- ja SQL Server-tietokantojen kanssa ja vertailee niitä myös keskenään. Ohjelman kotisivuilta löytyy ohjelmasta ilmainen kokeiluversio:
http://www.softinterface.com/compare-database/compare-access.htm

Seuraavia työkaluja voi käyttää ***asennuksen testauksessa*:**

InstallShield

InstallShield on suosittu asennuspakettien luomiseen tarkoitettu ohjelma, ei testaustyökalu, mutta sen avulla voidaan tehdä asennettavuustestausta. Ohjelmalla voi nauhoittaa valitut asennusmääritykset, ja ohjelma tallettaa ne response file-tiedoston. Asennusta voidaan tämän jälkeen ajaa automaattisesti ”hiljaisena asennuksena” niin, että asennusohjelma hakee asennusmääritykset response-tiedostosta. Tiedoston avulla voidaan asennusmääritykset pitää samoina eri ympäristöihin asennettaessa. Tarvittaessa tiedostoa voidaan myös muokata helposti ja tehdä uudet testit eri määrityksillä. Ohjeita response-tiedoston luomiseksi:
http://www.direct-io.com/index.htm?http&&&www.direct-io.com/kb/q100007.htm

InstallWatch

Avoimen lähdekoodin ohjelma, joka jäljittää muutoksia, joita järjestelmässä tapahtuu asennettaessa tai poistettaessa ohjelmia tai laitteistoa. Ohjelma tallettaa muutostiedot tietokantaan.
http://www.epsilonsquared.com/installwatch.htm

Testaukselle on usein hyötyä siitä, että testattavan sovelluksen toimintaa näytöllä pystytään nauhoittamaan. ***Näytön nauhoittamista*** voidaan käyttää muun muassa virheraportoinnin tukena ja käyttäjätestauksessa.

CamStudio

Avoimen lähdekoodin näyttönauhuri. Nauhoittaminen on kätevä tapa tehdä esimerkiksi käyttäjätestausta: annetaan testausryhmälle lista tehtävistä, jotka tulee suorittaa. Testitilanteet voidaan nauhoittaa avi-videoiksi, joita myöhemmin tarkastelemalla voidaan tehdä johtopäätöksiä käytettävyydestä:
http://camstudio.org/

Wink

Avoimen lähdekoodin näyttönauhuri:
http://www.microimages.com/support/Wink.htm

Fuzz testing eli sotkeminen on testaustekniikka, jossa ohjelmalle syötetään odottamatonta, vääränmuotoista tietoa ja tarkastellaan, miten se vaikuttaa sovelluksen toimintaan. Tekniikkaa käytetään yleisimmin tiedostomuotojen ja tietoliikenneprotokollien testaamiseen, mutta mahdollista on sotkea mikä tahansa sovelluksen syöte. Esimerkkejä työkaluista:

Filefuzz

Filefuzz on graafisella käyttöliittymällä varustettu ilmaisohjelma Windows-ympäristöön. Ohjelma automatisoi sovellusten käynnistämistä ja etsii poikkeuksia, joita odottamattomat tiedostomuodot saavat aikaan.
http://labs.idefense.com/software/fuzzing.php

Testiaineiston generointiin on olemassa erilaisia työkaluja: myös hallinnan työkaluissa esitellyt HP QualityCenter ja TestLink sisältävät tällaisia ominaisuuksia. Näiden lisäksi on olemassa monen tasoisia ja –laajuisia sovelluksia erilaisen testiaineiston luomiseen, kuten:

Comformiq Qtronic

Erittäin monipuolinen, kaupallinen automaattisen testisuunnittelun työkalu. Ohjelmiston ominaisuudet on listattu täällä:
http://www.conformiq.com/features.php

AllPairs

Syötteiden parittaiseen testaamiseen tarkoitettu ilmaistyökalu. Ohjelmaan syötetään muuttujat, ja niiden perusteella se generoi testijoukon, joka kattaa muuttujakombinaatiot mahdollisimman vähillä testitapauksilla.
http://www.satisfice.com/tools/pairs.zip

AEGT

Kaupallinen web-sovellus syötteiden parittaiseen testaamiseen. Sivuilta löytyy ohjeet ilmaisen kokeilutilin luomiseksi:
http://aetgweb.argreenhouse.com/freetrial.shtml

TurboData

Työkalu, jonka kotisivuilta löytyy sekä videotutoriaali että ilmainen kokeiluversio. Ohjelma luo testitietueita ja syöttää ne tietokantaan. On myös mahdollista syöttää testattavaan tietokantaan oikeita arvoja ohjelmiston tarjoamasta tietokannasta:
http://www.turbodata.ca/

ER-Datagen

Mallipohjaisen testiaineiston luomiseen tarkoitettu kaupallinen ohjelma, josta on ladattavissa kokeiluversio käyttöohjeineen. Se on integroitavissa mm. ERwiniin.
http://www.datatect.com/er-datagen/er-datagen-info.htm

Perlclip

Pieni Perl-ohjelma, jolla voi luoda testisyötteitä. Käyttöohjeessa listatuilla komennoilla saadaan luotua syötteitä, jotka kopioituvat windowsin leikepöydälle ja ovat siitä liitettävissä esim. testattavan ohjelmiston input-kenttään.
http://www.satisfice.com/tools/perlclip.zip

DgMaster

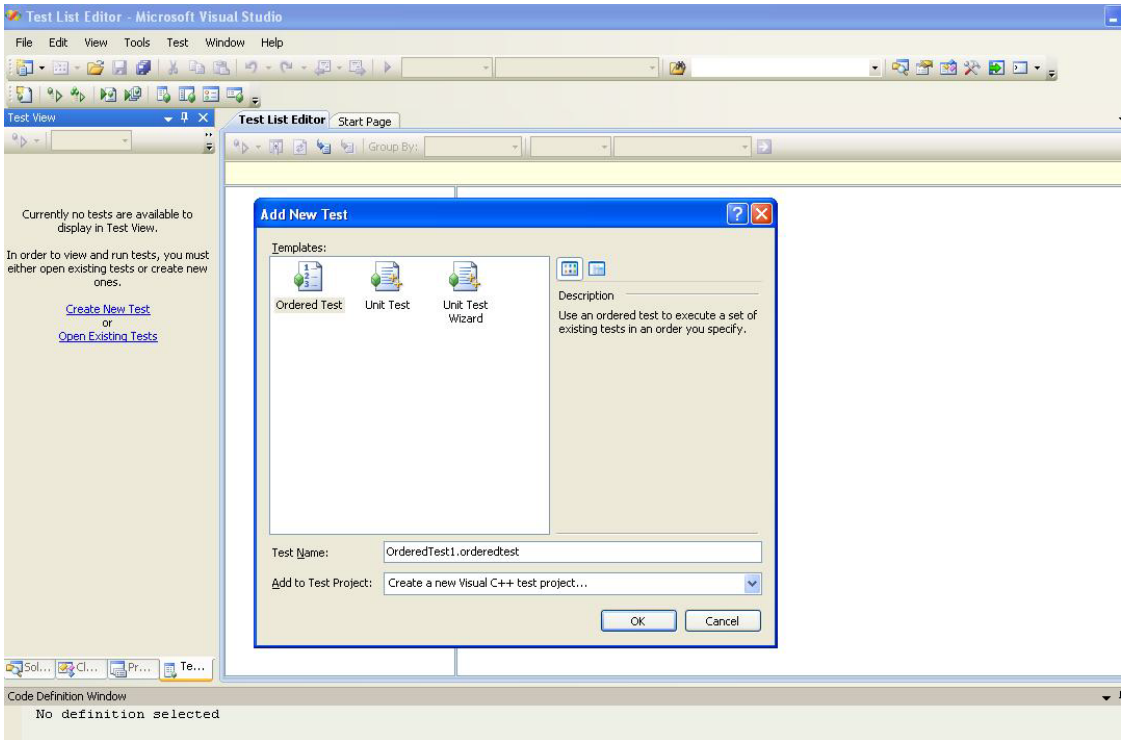
Graafisella käyttöliittymällä varustettu avoimen lähdekoodin testisyötegeneraattori. Se tukee erilaisia tietotyyppejä ja generoi dataa eri muodoissa (.txt, .xml,.db). Generaattori on laajennettavissa uusilla datageneraattori-luokilla:
http://sourceforge.net/projects/dgmaster/

9.3.9 IDE:t ja koostamistyökalut

Laajat ohjelmointiympäristöt eli IDE:t (Integrated Development Enviroment) sisältävät testaus- ja analysointityökaluja, ja niihin on mahdollista asentaa loputon määrä erilaisia testauksen automatisointiin liittyviä lisäosia. Näitä ympäristöjä ovat muun muassa Microsoft Visual Studio, NetBeans ja Eclipse. Lisäksi on olemassa koostamistyökaluja kuten Apache Ant, joka ei itsessään ole ohjelmointiympäristöjä, vaikka niissä on mahdollista tehdä samankaltaisia testaustehtäviä.

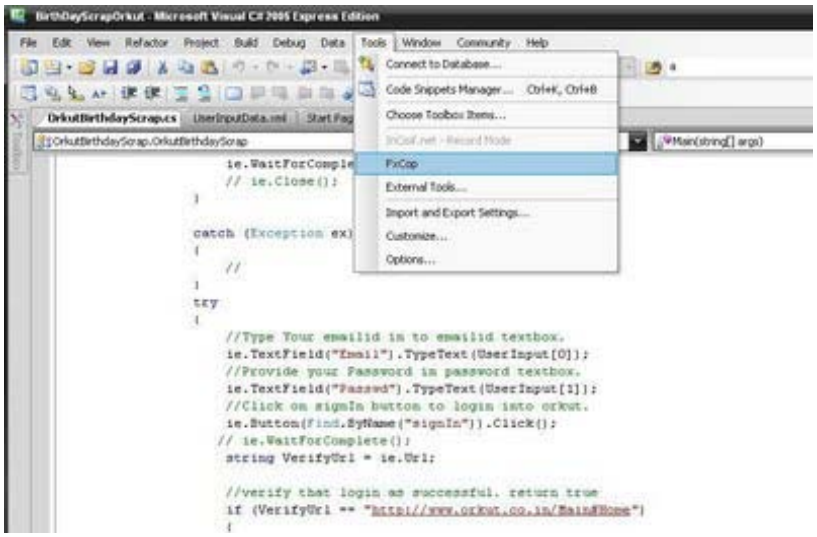
Microsoft Visual Studio

Microsoft Visual Studio 2008 Professional sisältää mahdollisuuden laatia olemassa olevista testitapauksista järjestyksessä suoritettavia testijoukkoja, sekä työkalut yksikkötestauksen automaatiota varten (kuva 9.12).



Kuva 9.12 Visual Studion testaustyökalut

Tämän lisäksi Visual Studioon on mahdollista saada lisäosana analysointityökalu, joka aiemmin oli nimeltään FxCop, mutta nykyään tunnetaan nimellä Code-Metrics. Visual Studio Team Systemin (VSTS) eri versiot sisältävät analysointi- ja testaustyökaluja, mutta Studion versioihin, joihin Team Systemiä ei pysty asentamaan, voidaan FxCop ladata lisäosana. Kuvassa 9.13 se on asennettu Visual Studio 2005 Express-versioon.(Microsoft, 2009. Kreynin, 2009.)



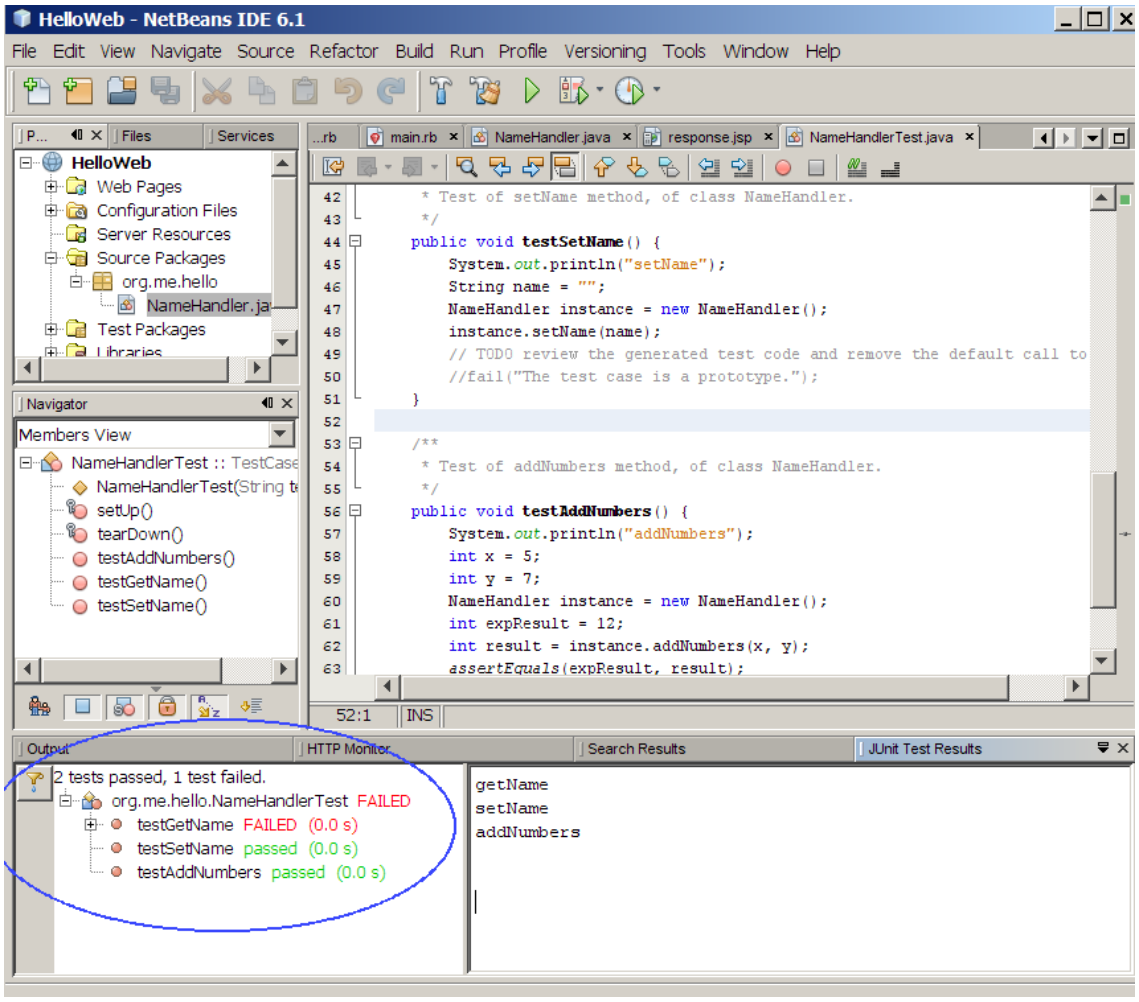
Kuva 9.13 FxCop

FxCop on ladattavissa osoitteesta
http://www.microsoft.com/downloads/details.aspx?familyid=9AEAA970-F281-4FB0-ABA1-D59D7ED09772&displaylang=en

Microsoftin kehitysympäristöihin on mahdollista integroida myös muiden toimittajien kaupallisia ohjelmistoja. Muun muassa Testwell-tuoteperhe, jonka tuotteet ovat laajasti käytettyjä ja jotka voidaan integroida myös Visual Studioon. Testwell CTC++ on kattavuusanalysaattori, C- ja C++- kielille, ja Testwell CTA++ on työkalu, joka on tarkoitettu C++:n luokkien, kirjastojen ja aliohjelmien yksikkötestaukseen. Lisäksi tuoteperhe sisältää kompleksisuuden ja staattisen analyysin työkalut; CMT++ C- ja C++-kielille sekä CMTJava Java-kielelle. Tuotetietoja löytyy valmistajan sivuilta:
http://www.testwell.fi/availabi.html

NetBeans

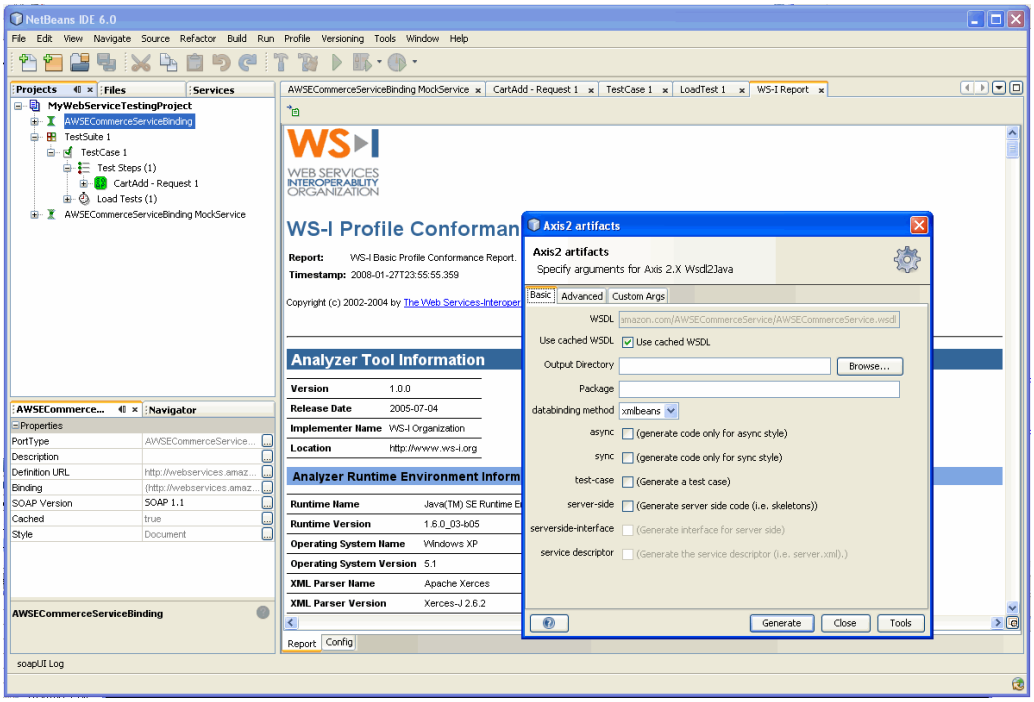
Netbeans on Java-kehitysympäristö, johon on myös integroitavissa erilaisia testausvälineitä, joista osa on kaupallisia tuotteita ja osa avoimen lähdekoodin sovelluksia. Yleisimmin käytössä olevia testaustyökaluja ovat dokumentissa aiemmin esitelty yksikkötestaustyökalu JUnit (kuva 9.14):



Kuva 9.14 NetBeans ja JUnit

Yksikkötestauksen kattavuuslukujen laskemiseksi NetBeansiin voidaan liittää Code Coverage-lisäosa, joka perustuu Emma-kattavuustyökaluun. NetBeans ja tässä luvussa esiteltävä Ant-koostamistyökalu voidaan integroida myös toisiinsa, ja näin Ant-työkaluja voidaan käyttää suoraan NetBeansista.

NetBeansiin on liitettävissä myös erilaisia web-sovellusten testausvälineitä. Yksi paljon käytetty lisäosa on jo aiemmin esitelty SOA-testeri soapUI (kuva 9.15). Lisäosa on tarkoitettu Web Service-rajapintojen testaamiseen:

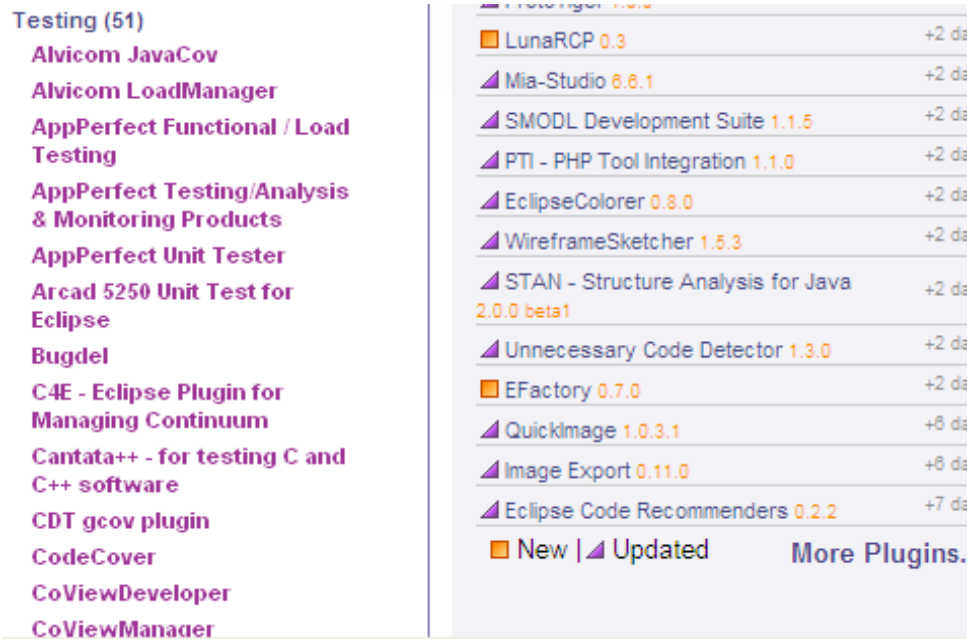


Kuva 9.15 Eviware SoapUi NetBeansissa

Lisää NetBeans-lisäosia löytyy esimerkiksi täältä:
http://plugins.netbeans.org/PluginPortal/faces/CategoryPage.jsp?categoryname=Test+Tools&orderby=DOWNLOADS

Eclipse

Java-kehitysympäristö, johon löytyy lisäosina testaustyökaluja. Latauskeskuksesta löytyy tällä hetkellä 51 erilaista lisäosaa (kuva 9.16):
http://www.eclipseplugincentral.com/



Kuva 9.16 Eclipsen testaus-lisäosat

Lisäksi Eclipse on mahdollista laajentaa TPTP-alustalla (Test & Performance Tools Platform), joka sisältää työkaluja muun muassa staattiseen analyysiin, profilointiin, yksikkötestaukseen, www-sovellusten testaukseen kuin manuaaliseenkin testaukseen. Lisäksi sen avulla voidaan luoda raportteja. Esimerkiksi täältä löytyy TPTP-tutoriaali:
http://www.vogella.de/articles/EclipseTPTP/article.html

Apache Ant

Apache Ant ei ole ohjelmointiväline, vaan ohjelmiston koostamistyökalu Java-ympäristöön. Sitä käytetään ohjelmiston automaattiseen koostamiseen, se hakee uudet ohjelmakoodit versionhallintajärjestelmästä, kääntää ne, ja ajaa määritetyt yksikkö- ja savutestit. Ant:ia voi käyttää päivittäisen koostamisen välineenä ketterissä ohjelmointimenetelmissä. Koostamisen yhteydessä on mahdollista tehdä paljon muutakin kuin edellä mainitut: Ant:issa itsessään on sisäänrakennettuna yli 80 erilaista tehtävää, ja kolmannen osapuolen lisäosia siihen on yli 100. Ant on avoimen lähdekoodin sovellus. Intro Ant:ista ja siitä, mitä sillä voi tehdä, löytyy täältä:
http://www.exubero.com/ant/antintro-s5.html

KUVAT

Kuva 1.1 Ohjelmistotuotannon osa-alueet, s. 13
Kuva 2.1 Ohjelmistotestauksen kehitysvaiheet, s. 16
Kuva 3.1 Testauksen koulukunnat (Pettichord), s. 22
Kuva 3.2 Käyttötapaus-vaatimusmatriisi, s. 24
Kuva 3.3 Testaus-käyttötapausmatriisi, s.25
Kuva 4.1 Ohjelmistotuotannon vesiputousmalli, s. 39
Kuva 4.2 Testauksen V-malli, s. 30
Kuva 4.3 Verifiointi ja validointi, s. 33
Kuva 4.4 Yksikkötestauksen sijoittuminen V-mallissa, s. 34
Kuva 4.5 Test Driven Development, s.37
Kuva 4.6 Integrointitestauksen sijoittuminen V-mallissa, s. 38
Kuva 4.7 Jäsentävä integrointi, s. 39
Kuva 4.8 Kokoava integrointi, s. 40
Kuva 4.9 Järjestelmätestauksen sijoittuminen V-mallissa, s. 42
Kuva 4.10 Hyväksymistestauksen sijoittuminen V-mallissa, s. 50
Kuva 5.1 Staattiset testausmenetelmät, s. 54
Kuva 5.2 Dynaamiset testausmenetelmät, s. 60
Kuva 5.3 Eri testaustekniikoiden käsitteelliset eroavaisuudet s. 61
Kuva 5.4 Testitapausten jako ekvivalenssiluokkiin, s. 65
Kuva 5.5 Ekvivalenssiluokan jakautuminen, s. 65
Kuva 5.6 Päätöspuu binäärimuuttujista, s. 69
Kuva 5.7 Päätöstaulu binäärimuuttujista, ensimmäinen vaihe, s. 70
Kuva 5.8 Päätöstaulu binäärimuuttujista, sääntöparien etsintä 1, s.70
Kuva 5.9 Päätöstaulu binäärimuuttujista, sääntöparien etsintä 2, s.70
Kuva 5.10 Valmis päätöstaulu binäärimuuttujista, s. 71
Kuva 5.11 Päätöstaulun avulla valitut testitapaukset, s. 71
Kuva 5.12 Digitaalikellon tilasiirtymäkaavio, s. 72
Kuva 5.13 Digitaalikellon toiminnan tilasiirtymäpuu 0-tasolla, s. 73
Kuva 5.14 Digitaalikellon 0-tason tilasiirtymäpuun testijoukot, s. 73
Kuva.5.15 Digitaalikellon tilasiirtymäpuu 1-tasolla, s.74
Kuva 5.16 Digitaalikellon 1-tason tilasiirtymäpuun testijoukot, s. 75

Kuva 5.17 Käyttötapauskuvaus, s. 77
Kuva 6.1 Testauksen vaiheet ja suorittajat, s. 84
Kuva 6.2 Vaatimuksiin liittyvän virheen suhteellinen korjauskustannus ohjelmistoprojektin eri vaiheissa (Boehm, 1981), s. 86
Kuva 8.1 Ohjelmistojen rakenteen kehittyminen, s. 100
Kuva 8.2 Luokka ja olio, s. 102
Kuva 8.3 Metodit, s. 102
Kuva 8.4 Tietojäsenet, s. 103
Kuva 8.5 Olioiden väliset viestit, s. 104
Kuva 8.6 Luokka, s. 105
Kuva 8.7 Periytyminen, s. 106
Kuva 8.8 Perintä ja kooste, s. 107
Kuva 8.9 Moniperintä (TTY ohjelmistotekniikan laitos / Timo Lehtonen 2004–2005), s. 108
Kuva 8.10 Sekvenssikaavio (TTY Pori, Arto Stenberg), s. 115
Kuva 8.11 Vuokaavio (TTY Pori, Arto Stenberg), s. 115
Kuva 8.12 Luokkakaavio (Conformiq Software Ltd.), s. 116
Kuva 8.13 Luokkatestien skaalaaminen erilaisille riskitasoille (Pöyhönen ym. 2002) s. 118
Kuva 9.1 Automatisoidun testauksen elinkaari, s. 128
Kuva 9.2 Automatisoinnin päätösprosessi (Dustin ym. 2003, s. 31) s. 129
Kuva 9.3 Automatisoidun testaustyökalun valintaprosessi (Dustin ym. 2003, s. 69), s. 132
Kuva 9.4 Testiautomaation käyttöönottoprosessi (Dustin ym. 2003, s. 109) s. 135
Kuva 9.5 Testausprosessin analysointi (Dustin ym. 2003, s. 112), s. 136
Kuva 9.6 Testityökalun tutkiminen (Dustin ym. 2003, s. 134), s. 137
Kuva 9.7 Testaussuunnitelman syötteet ja sisältö (Dustin ym, 2003, s. 193), s. 138
Kuva 9.8 Ohjelmistokehityksen ja testauksen tehtävien ajoittuminen(Dustin 2001), s. 140
Kuva 9.9 Testaustyökalujen tyypit(mukaillen Pohjolainen, 2003), s. 142
Kuva 9.10 HP Quality Center, s. 143
Kuva 9.11 TestLink, s. 144

Kuva 9.12 Visual Studion testaustyökalut, s. 164
Kuva 9.13 FxCop, s. 165
Kuva 9.14 NetBeans ja JUnit, s. 166
Kuva 9.15 Eviware SoapUi NetBeansissa, s. 167
Kuva 9.16 Eclipsen testaus-lisäosat, s.168

LÄHTEET

Ambler, Scott W. 2009. Introduction to Test Driven Design.
http://www.agiledata.org/essays/tdd.html
(Luettu 18.8.2009)

Bach, James. 2001. What is Exploratory Testing? Stickyminds weekly column.
http://www.stickyminds.com/sitewide.asp?Function=WEEKLYCOLUMN&ObjectId=2255&objecttype=ARTCOL
(Luettu 4.9.2009)

Bach, James. 2003. Exploratory Testing Explained.
http://www.satisfice.com/articles/et-article.pdf
(Luettu 4.9.2009)

Binder, Robert. 1994. Testing object-oriented systems: A status report..
http://www.rbsc.com/pages/ootstat.html
(Luettu 11.9.2009)

Binder, Robert. 1999. Testing object-oriented systems.
Addison-Wesley.

Blair, Obenski, Bridickas. 1992. Patriot missile system failure report.
http://www.fas.org/spp/starwars/gao/im92026.htm
(Luettu 19.8.2009)

Boehm, Barry W. 1981. Software Engineering Economics.
Prentice Hall, Englewood Cliffs, NJ.

Borysowich, Craigh. 2007. Testing Via Error Guessing.
http://it.toolbox.com/blogs/enterprise-solutions/testing-via-error-guessing-17139
(Luettu 25.9.2009)

Dustin. 2001. The Automated Testing Lifecycle Methodology
http://www.informit.com/articles/article.aspx?p=21468&seqNum=6
(Luettu 12.11.2009)

Dustin, Rashka, Paul. 2003.
Automated Software Testing. Introduction,Management and Performance.
Addison-Wesley.

Gelperin, Hezel. 1988. The growth of software testing.
Communications of ACM, vol. 31, No. 6. s.687 – 695
http://portal.acm.org/citation.cfm?doid=62959.62965
(Luettu 4.8.2009)

Haikala, Märijärvi. 2004. Ohjelmistotuotanto.
Talentum Media Oy.

Hecht, Herbert. 2003. Systems Reliability and Failure Prevention.
Artech House, Incorporated.

Hetzel, William. 1988. The complete guide to software testing.
John Wiley & Sons, Incorporated.

Holopainen, Juha. 2005.Regressiotestaus ja testien valintatekniikat.
Pro Gradu-tutkielma, Kuopion yliopisto.
http://www.uku.fi/tike/his/avointa/julkaisut/Regressiotestaus_ja_testien_valintatekniikat.pdf
(Luettu 27.8.2009)

IEEE 1998. 829-1998 Standard for Software Test Documentation.
http://wilma.vub.ac.be/~se1_0607/svn/bin/cgi/viewvc.cgi/documents/standards/IEEE/IEEE-STD-829-1998.pdf?revision=45 (Luettu 4.8.2009)

IEEE 2008. 829-2008 Standard for Software and System Test Documentation.
http://ieeexplore.ieee.org/servlet/opac?punumber=4578271
(Luettu 4.8.2009)

IEEE 1998. 1028-1997 Standard for Software Reviews.
http://membres.lycos.fr/benoitouellet/Peer_rev.pdf
(Luettu 31.8.2009)

ISEB 2009a. ISEB Foundation Certificate in Software Testing, overview
http://www.bcs.org/server.php?show=conWebDoc.2299
(Luettu 4.8.2009)

ISEB 2009b. ISEB Intermediate Certificate in Software Testing
http://www.bcs.org/server.php?show=nav.9609
(Luettu 4.8.200)

ISEB 2009c. ISEB Practitioner Certificate in Software Testing
http://www.bcs.org/server.php?show=nav.6956
(Luettu 4.8.2009)

ISTQB 2007. Suomennos ISTQB:n testaussanastosta ”Standard glossary of terms used in Software Testing Version 2.0”
http://ttlry-fi-bin.directo.fi/@Bin/bf55cf1cfaae232ff73073310d4cc02e/1250671363/application/pdf/14155799/istqb_sanasto.pdf
(Luettu 19.8.2009)

ISTQB 2009. Certified Tester-sertifioitu testaaja. Perustason sertifikaattisisältö suomeksi.
http://ttlry-fi-bin.directo.fi/@Bin/52462650070152296 4a01b6a3138a69d/1251706576/application/pdf/133947098/ISTQB_FL_syllabus_Finnish_20090813.pdf
(Luettu 31.8.2009)

Jaakkola, Hannu. 2006. Laatuattribuutit ja hyvä suunnittelu. Luentokalvot.
http://www.pori.tut.fi/~hj/for-guests/public/ohs/05-laatuattribuutit&suunnittelu.pdf
(Luettu 13.10.2009)

Kaner, Bach, Pettichord. 2002. Lessons learned in software testing: a context-driven approach. Wiley Computer Publishing
Koskimies, Kai. 1998. Pieni Oliokirja. Suomen Atk-kustannus Oy.

Kreynin, Vadim. 2009. Integrate FxCop 1.36 & VS 2008.
http://vkreynin.wordpress.com/2009/05/09/integrate-fxcop-1-36-vs-2008/
(Luettu 27.9.2009)

Levenson, Turner. 1993. An Investigation of the Therac-25 Accidents.
http://courses.cs.vt.edu/professionalism/Therac_25/Therac_1.html
(Luettu 19.8.2009)

Lions, J. L. 1996. ARIANE 5 Flight 501 Failure: Report by the Inquiry Board
http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html
(Luettu 19.8.2009)

Microsoft. 2009. Microsoft Visual Studion tuotetiedot.
http://www.microsoft.com/visualstudio/fi-fi/products/teamsystem/default.mspx
Luettu 27.9.2009)

Microsoft Corporation. 2008. Codezone: SQL-injektio.
http://www.codezone.fi/Tietoturva2.Codezone

Moisio, Aleksi. 2008. Digitoday: Näin XSS-aukoilla on hyökätty.
http://m.digitoday.fi/?page=showSingleNews&newsID=20089020
(Luettu 2.11.2009)

Myers, Sandler, Badgett. 2004. Art of Software Testing.
John Wiley & Sons, Incorporated

Myöhänen, Hannu. 2002. Jäljitettävyys ohjelmistotuotannon tukena.
Pro Gradu-tutkielma, Kuopion yliopisto.
http://www.plugit.fi/julkaisut/docs/myohanen-2002.pdf
(Luettu 14.8.2009)

OAMK. 2006. Software Business Competence-sivusto, ohjelmistotestaus.
http://www.oamk.fi/sbc/testaus/testausstrategiat.htm

Pettichord, Brett. 2007. Schools of software testing
http://www.io.com/~wazmo/papers/four_schools.pdf
(Luettu 11.8.2009)

Paavilainen, Juhani (toim.) 2004. Tietoturvallinen ohjelmointi.
Tekninen julkaisu, b-sarja, Tampereen teknillinen yliopisto.
http://www.cs.uta.fi/reports/bsarja/B-2004-5.pdf
(Luettu 26.8.2009)

Poimala, Heikniemi, Blåfield. 2008. Ketterät käytännöt.fi-sivusto.
http://www.ketteratkaytannot.fi/fi-FI/Menetelmat/
(Luettu 9.9.2009)

Pohjolainen, Pentti. 2003. Ohjelmiston testauksen automatisointi.
Pro Gradu-tutkielma, Kuopion yliopisto.
http://www.cs.uku.fi/tutkimus/Teho/PenttiPohjolainen_Gradu.pdf
(Luettu 5.11.2009)

Pyhäjärvi, Pöyhönen. 2009. Testauskirjan materiaalipaketti Ketterä testaus.
http://www.testauskirja.com/materiaalit.html
(Luettu 9.9.2009)

Pöyhönen, Stenberg. 2002. Laajojen oliojärjestelmien testaus-diasarja.
Nokia Research Center, SW Technology Laboratory
http://www.cs.tut.fi/tapahtumat/olio2002/poyhonen.pdf
(Luettu 15.9.2009)

Shore, James. 2005. The Art of Agile: Red-Green-Refactor.
http://jamesshore.com/Blog/Red-Green-Refactor.html
(Luettu 27.8.2009)

Software Testing Geek. 2009. Gray Box Testing.
http://www.testinggeek.com/index.php/testing-types/system-knowledge/51-grey-box-testing
(Luettu 28.9.2009)

Taina, Juha. 2004. Ohjelmistojen testaus-luentokalvot.
http://www.cs.helsinki.fi/u/taina/ohte/s-2004/luennot/Luku02.pdf
(Luettu 3.3.2010)

Taina, Juha. 2007. Ohjelmistojen testaus-luentokalvot.
http://www.cs.helsinki.fi/u/taina/ohte/s-2008/luennot/
(Luettu 11.9.2009)

TestausOSY – FAST 2009. Testauksen osaamisyhteisö
http://pcuf.fi/sytyke/kerhot/testaus/
(Luettu 4.8.2009)

Tietotekniikan liitto 2007. FiSTB, testauksen osaamisyhteisö
http://www.ttlry.fi/yhdistykset/osaamisyhteisot/fistb/
(Luettu 4.8.2009)

Vuorinen, Simo. 2005. Ohjelmistoarkkitehtuurista puoliketterästi,
Systeemityö-lehden teema-artikkeli.
http://www.pcuf.fi/sytyke/lehti/kirj/st20053/ST053-25A.pdf
(Luettu 24.8.2009)

Wallace, Ippolito, Cuthill. 1996. Reference Information for the Software Verification and Validation Process. Diane Publishing Company

Watkins, John. 2001. Testing IT: An Off-the-Shelf Software Testing Handbook. Cambridge University Press

Wikipedia. 2009. Remote File Inclusion
http://en.wikipedia.org/wiki/Remote_File_Inclusion
(Luettu 2.11.2009)