Tiina Tersa

TESTAUSMENETELMIEN KÄYTTÖ SOVELLUKSEN SYSTEEMITESTAUSVAIHEESSA

Tietotekniikan pro gradu -tutkielma

Ohjelmistotekniiikan linja

20.8.2002

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Tiina Tersa

Yhteystiedot: Sähköposti tiina.tersa@cc.jyu.fi ja puh. 040-5748488.

Työn nimi: Testausmenetelmien käyttö sovelluksen systeemitestausvaiheessa.

Title in English: Using Testing Methods in the System Testing of a Software Application.

Työ: Pro gradu -tutkielma

Sivumäärä: 137 + 2

Linja: Ohjelmistotekniikka.

Teettäjä: Jyväskylän yliopisto, tietotekniikan laitos

Avainsanat: Ohjelmistoprojekti, testaus, V-malli, yksikkötestaus, integraatiotestaus, systeemitestaus, hyväksymistestaus, testausmenetelmä, lasilaatikkomenetelmä, mustalaatikkomenetelmä.

Keywords: Software project, testing, V-model, unit testing, integration testing, system testing, acceptance testing, testing method, glass-box testing, white-box testing, black-box testing.

Tiivistelmä: Tutkielmassa perehdytään ohjelmistotestaukseen olennaisena osana ohjelmistoprojektia siten, että testauksesta saadaan kattava kokonaiskuva. Pääpaino käsittelyssä on kuitenkin testauksen vaiheissa ja menetelmissä. Tutkimuskohteena ovat erityisesti systeemitestaus ja testauksen mustalaatikkomenetelmä, joita tutkitaan taskutietokoneelle toteutetun sovelluksen testauksen pohjalta. Tutkimuksessa analysoidaan valittujen menetelmien sopivuutta testaukseen sekä pohditaan teorian ja kokemuksen kautta keinoja testauksen toteuttamiseksi paremmin.

Abstract: This thesis covers the different aspects of software testing, with emphasis on the various phases of testing and the methods that can be used. The thesis concentrates mainly on the system testing phase and the black-box testing method. They were applied during the development of an application for a PDA device. The thesis analyses how suitable the selected methods are for testing. It also considers the theoretical and practical aspects of improving the execution of testing, within a software development project.

Saatesanat

Kiitos Tietotekniikan laitoksen lehtori Jukka-Pekka Santaselle tutkielmani ohjaamisesta ja tarkastamisesta. Kiitos myös Tommi Kärkkäiselle ja kaikille heille, jotka edesauttoivat opiskeluani Jyväskylän yliopistossa.

Lisäksi haluan kiittää IBM:ltä tutkielmani tarkastajaa Sari Vainiota. Kiitos myös Ilkka Sipilälle, Hillevi Luostariselle ja Juha Hulkkoselle, jotka mahdollistivat tutkielmani tekemisen. Kiitos kuuluu myös Olli Nuorannolle ja kaikille muille, jotka ovat osaltaan auttaneet tutkielmassani.

Lopuksi vielä aivan erityinen kiitos vanhemmilleni Airalle ja Pekalle sekä siskolleni Piialle koko opiskelujeni ajan kestäneestä tuesta ja kannustuksesta.

Helsingissä 20.8.2002
Tiina Tersa

Termiluettelo

Ekvivalenssiluokka	on testidatan valintamenetelmä, jossa dataa vähennetään jakamalla se luokkiin.
Hyväksymistestaus	on asiakkaan suorittama testaus todellisessa käyttöympäristössä.
Integraatiotestaus	keskittyy komponenttien rajapintojen ja keskinäisen kommunikoinnin testaamiseen.
Jäsentävä testaus	aloitetaan koko järjestelmän toiminnan testauksesta edeten vähitellen alemmille tasoille.
Kokoava testaus	aloitetaan alemman tason pienistä komponenteista siirtyen vähitellen korkeammille tasoille.
Lasilaatikkotestaus	on testausmenetelmä, jossa käytetään koodia apuna virheitä etsittäessä.
Mustalaatikkotestaus	on testausmenetelmä, jossa syötteiden ja vasteiden avulla testataan toiminnan oikeellisuus.
Raja-arvoanalyysi	on testidatan valintamenetelmä, jossa dataan valitaan luokkien rajoilla olevia arvoja.
Regressiotestaus	varmistaa sovelluksen oikeellisuuden uudelleen sen jälkeen, kun virheitä on korjattu tai toiminnallisuutta on lisätty tai muutettu.
Systeemitestaus	keskittyy kehitetyn järjestelmän testaukseen kokonaisuudessaan käyttötarkoitusta vastaavassa ympäristössä.

Testiajuri	on käytössä yksikkötestauksessa ja jäljittelee muiden yksikköjen antamia syötteitä.
Tynkä	on käytössä yksikkötestauksessa mahdollistamaan testauksen aikana yksiköiden välisen kommunikoinnin.
Yksikkötestaus	on kehittäjän suorittama testaus yksikkötasolla.

Sisältö

1 Johdanto

Kaikissa ohjelmistoissa ja sovelluksissa on virheitä. Juuri kehitetyissä sovelluksissa on ennen testausvaiheen aloittamista yleisesti yksi virhe muutamaa kymmentä koodiriviä kohti ja arvioiden mukaan noin viittä prosenttia virheistä ei koskaan löydetä. Sovelluksen testaus ja virheenjäljitys saattavat maksaa 50-80% sovelluksen ensimmäisen toimivan version kustannuksista. Tunnettua on myös se, että mitä myöhemmin virhe havaitaan, sitä kalliimpaa sen korjaaminen on. Silti asiakkaalle toimitetaan tuotteita, joissa on jopa tuhoisia virheitä.

Ohjelmistojen testaus kuuluu olennaisesti ohjelmistokehitykseen. Testauksen pääasiallisena tarkoituksena on havaita virheitä. Monesti testauksella pyritään osoittamaan, että on kehitetty oikea, vaatimusten mukainen ohjelma, joka toimii oikein kaikissa tilanteissa. Koskaan ei testauksella kuitenkaan pystytä takaamaan ohjelman täydellistä oikeellisuutta, vaan ainoastaan osoittamaan virheitä ja antamaan todennäköisyys, kuinka paljon sovelluksessa vielä on virheitä.

Yleisenä ohjeena testaajille Boris Beizer antaa seuraavan: ”Testaaja! Tuhoa sovellus (niin kuin sinun tehtäväsi on) ja aja se äärimmilleen - mutta älä nauti kehittäjien tuskasta.”

Olen työssäni ollut testaajana kahdessa eri projektissa viimeisen vuoden aikana. Projektit erosivat toisistaan selkeästi kokonsa puolesta toisen ollessa vajaan kymmenen työntekijän ja toisen noin sadan työntekijän projekti. Testaus suoritettiin näissä eri tavoin ja erityisesti testauksen arvostus vaihteli.

Testausta ei yleisesti arvosteta, eikä sille varata riittävästi resursseja projektista. Välillä testausta suoritetaan lähes vailla järjestelmällistä toimintatapaa, vain oman ajattelun ohjaamana ja testaajiksi yleensä asetetaan vasta projektiin tulleita työntekijöitä. Parhaan tuloksen saamiseksi on testaajien kuitenkin osattava dokumentoida, kommunikoida, raportoida sekä käyttää kuhunkin testauksen vaiheeseen liittyviä testausmenetelmiä.

Testaus onkin ohjelmistoprojektin olennainen osa. Täten tutkielman teoriaosuudessa tutustutaan testaukseen laajasti esitellen testauksen suunnittelua, raportointia ja toteuttamista. Ohjelmistoprojektin tavoin testaus koostuu eri vaiheista, joita kaikkia tässä käsitellään. Tutkielman olennaisin osuus on testausmenetelmät.

Testausmenetelmät jaetaan yleisesti kahteen luokkaan ja testauksen kokonaisuuden kannalta on olennaista tutustua molempiin. Teoriaa sovellettiin käytäntöön projektissa lähinnä testauksen systeemitestausvaiheessa ja ainoastaan mustalaatikkomenetelmää soveltaen. Toisaalta testausta ei olisi voitu suorittaa irrallisena osana, joten yleinen tieto testauksesta oli olennaista tuntea etukäteen.

Tutkielmassa käsitellään ohjelmistotestausta ensin lähdemateriaalin pohjalta. Käytännön sovellustestausta tarkastellaan tutkielmassa ohjelmistotestauksessa yleisesti käytettyjen menetelmien, tasojen ja vaiheiden kautta. Luvussa 2 esitellään tutkielman tausta ja tavoitteet. Luvussa 3 kuvataan testauksen merkitystä ja sijoittumista ohjelmistoprojektin vaiheisiin. Luvussa 4 käsitellään testauksen jakautumista tasoihin aina yksikkötestauksesta asiakkaan suorittamaan koko järjestelmän testaukseen saakka. Luku 5 käsittelee erilaisia testauksen menetelmiä. Luvun alkuosassa keskitytään lasilaatikkomenetelmiin ja loppuosassa mustalaatikkomenetelmiin. Luku 6 soveltaa näitä aiemmissa luvuissa kuvattuja menetelmiä toteutettuun projektiin systeemitestausvaiheen kannalta. Luvussa 7 pohditaan, miten testaus onnistui näitä menetelmiä käyttämällä ja mitä testauksessa kannattaisi tehdä toisin.

2 Tutkielman taustaa ja tavoitteet

Testaukseen liittyy paljon harhaluuloja, eikä testausta arvosteta yleisesti. Tästä kertovat ne lukuisat projektit, jotka ovat epäonnistuneet testaukseen kohdistuvan välinpitämättömän asenteen takia.

Täysipainoinen ja oikealla tavalla toteutettu testaus on tärkeä osa ohjelmistoprojektia. Tämä on havaittavissa siitä, kuinka virheiden korjaamisen kustannukset kohoavat ohjelmistoprojektin edetessä. Määrittelyvaiheessa löydetty virhe on suhteellisen halpa korjata, kun taas jo asiakkaan käyttöön ehtineen sovelluksen korjaaminen on yleensä huomattavan kallista. Luvussa käsitellään aluksi lyhyesti testauksen historiaa, jonka jälkeen käsitellään virheen määritelmiä ja luokittelua eri vakavuusasteisiin. Lopuksi esitellään tutkielman vaatimukset ja tavoitteet. Luvun pääasiallinen lähde on [Patton].

2.1 Testauksen historiaa

Ohjelmoinnin alkuaikoina ajateltiin, että ainoastaan koneen tarvitsee lukea ja ymmärtää tietokoneohjelmia, eikä ihminen tarvitse näitä taitoja lainkaan. Ajateltiin, että kaikki tarvittava testaus voidaan tehdä koneilla, eikä koodin selkeyteen tai luettavuuteen tarvinnut kiinnittää huomiota. Melzer huomauttaakin artikkelissaan [Melzer], että koska koodin ulkoasusta ei välitetty, ajauduttiin epäselkeisiin sovelluksiin, joiden toiminnan selvittäminen koodista oli lähes mahdotonta.

Myöhemmin ohjelmoijien tehtäviin liitettiin koodin esittely muille sovelluskehittäjille, joten koodista oli pakko tehdä selkeämpää ja paremmin jäsenneltyä. Miettiessään koodin esitettävyyttä ohjelmoija samalla alustavasti testaa sovellusta ja tuottaa selkeämpiä rakenteita. Tällöin virheiden löytäminen testausvaiheessa sujuu helpommin kuin alunperin koneille suunnatuista ohjelmista. Lisäksi selkeyteen on vaikuttanut se, että ohjelmointikielet ovat kehittyneet koneläheisistä korkeamman tason kieliksi. Ohjelmointikieliä kehitettäessä niistä on muokattu ihmislähtöisempiä, joten kaikkien on helpompi ymmärtää niitä.

Drabick painottaa artikkelissaan [Drabick], että testauksen alkuaikoina testaus oli yleisesti ”ad-hoc” -tyyppistä kokeilua ohjelmistoprojektin loppuvaiheessa. Testaajat olivat kouluttautuneet lähinnä virheitten arvailuun. Vasta 1970-luvun lopulla Myers julkaisi kuuluisan kirjansa ”The Art of Software”, jossa hän esitteli toimivia testauksen tekniikoita. Näiden tekniikoiden käyttöönotto ja erityisesti hyödyllisyyden ymmärtäminen on ollut hidas prosessi. Kirjassa esitetyt tekniikat ovat olleet hyviä ja ne ovat käytössä edelleenkin, yli kaksikymmentä vuotta kirjan julkaisemisen jälkeen.

Toisaalta tietyt puutteellisuudet sovelluksien testauksessa ovat ymmärrettäviä, sillä tietotekniikka on tieteenä varsin nuori. Tietotekniikka on lisäksi omaksunut perinteisistä insinööritieteistä paljon menetelmiä, jotka eivät kuitenkaan sellaisenaan sovi sovelluskehitykseen. Joka tapauksessa tietotekniikka on alana saanut varsin erilaisen aseman, sillä sovellusten virheet hyväksytään. Päinvastoin kuin tietotekniikassa, lähes kaikilla muilla aloilla tuotteen valmistajalla on huomattavan suuri vastuu virheistä.

2.2 Virheen määritelmiä

Testauksen aikana tavoitteena on paikallistaa virheitä ja tietysti raportoida ne. Ei ole kuitenkaan itsestäänselvyys, mikä on virhe. Patton luettelee kirjassaan [Patton, luku 1] lukuisan joukon englanninkielisessä kirjallisuudessa termille virhe esiintyviä vastineita, kuten *defect*, *fault*, *error* ja *bug*. Eri nimitykset johtuvat työpaikan kulttuurista ja yhteydestä, jossa niitä käytetään. Joissain yrityksissä voidaan käyttää tunteja mietittäessä testaukselle yhteisesti hyväksyttyä sopivaa nimitystä, joka ei loukkaa kehittäjiä, mutta antaa oikeutuksen myös vakaville virheille.

Suomenkielisiä termejä esitellään kirjassa [Haikala et al., luku 15], jonka mukaan **virheenä** (engl. *bug*) voidaan pitää koodissa olevaa poikkeamaa sovelluksen toiminnallisesta tai teknisestä määrittelystä. Sovelluksen ja sen määrittelyjen tulkinnassa tosin saattaa olla erimielisyyksiä eri intressiryhmien välillä. Kun asiakkaan mielestä sovelluksen toiminta on virheellinen, saattaa se kehittäjän mielestä olla selvä toiminnallisuus. Täten kaikkia tyydyttävän ratkaisun saavuttamiseksi ja erimielisyyksien välttämiseksi dokumentaatioiden tulisi olla tarkkoja ja yksiselitteisiä.

Sovelluksessa olevan virheellisen kohdan suorittaminen voi aiheuttaa **vian** (engl. *fault*). Tästä ei välttämättä aiheudu ongelmia sovellukseen, sillä vika saattaa korjautua itsestään jonkin toisen toiminnallisuuden seurauksena tai sen vaikutus voi kumoutua toisen virheen tapahtuessa. Mutta joissain tapauksissa sovellukseen saattaa vian takia syntyä **häiriö** (engl. *failure*). Tällöin sovellus ei enää toimi toivotulla tavalla.

Suurin osa virheistä johtuu tuotteen määrittelyn (engl. *product specification*) virheellisyydestä. Määrittely kuvaa kehitettävän tuotteen, sen yksityiskohdat ja toimintavaatimukset. Projektista riippuen tämä määritelmä toteutetaan suullisesti tai tarkasti kirjallisena dokumentaationa.

Virheen määritelmä voidaan kuvata tuotteen määrittelyn kautta. Ron Patton listaa kirjassaan [Patton, luku 1] viisi vaihtoehtoa, joista yhdenkin toteutuessa virhe esiintyy. Seuraava lista ei kuitenkaan huomioi virheiden kriittisyyden järjestystä:

1. Ohjelmisto ei tee jotain, jota sen määrittelyn mukaisesti kuuluisi tehdä.
2. Ohjelmisto toimii tavalla, jonka määrittely kieltää.
3. Ohjelmisto toimii tavalla, jota määrittely ei mainitse.
4. Ohjelmisto ei tee jotain, jota määrittely ei mainitse, vaikka sen pitäisikin mainita.
5. Ohjelmisto on hankala ymmärtää, vaikeakäyttöinen, hidas tai käyttäjän mielestä selkeästi toimii väärin.

2.3 Virheiden luokittelua

Kaikki virheet eivät ole samanarvoisia, vaan ne voidaan luokitella vakavuudeltaan eri luokkiin. Virhe ohjelmakoodissa on vakava, jos se liittyy ohjelman kannalta elintärkeisiin osiin. Osa virheistä on pieniä, vain käyttäjää ärsyttäviä yksityiskohtia. Virheet voidaan luokitella esimerkiksi neljään **luokkaan**: **tuhoisaan**, **vakavaan**, **häiritsevään** ja **siedettävään**. Luokittelutavat vaihtelevat suuresti ja muun muassa Beizer luokittelee kirjassaan [Beizer, luku 2] virheet vakavuudeltaan kymmeneen eri luokkaan. Virheen vakavuus ohjaa kehittäjiä työssä heidän arvioidessaan virheiden korjaamisen aikataulua, järjestystä ja niiden vaatimia resursseja.

Onpa virheiden vakavuus mikä tahansa tuhoisasta häiritsevään yksityiskohtaan, ne aina laskevat tuotteen laatua. Jos ohjelmaa suoritettaessa tulee virheilmoitus kielellä, jota käyttäjä ei ymmärrä, hän joutuu katsomaan sanakirjasta. Tällaista virhettä voidaan pitää häiritsevänä. Toisaalta jos tuotteen kaikki käyttäjät joutuvat turvautumaan sanakirjaan, virheen vaikutukset ja siten myös vakavuus nousee.

Toinen tapa luokitella virheitä on antaa niille **prioriteetti** eli kuinka välttämätöntä ja kiireellistä virheen korjaaminen on. Patton muodostaa kirjassaan [Patton, luku 18] kyseiset neljä luokkaa seuraavasti prioriteettijärjestyksen mukaisesti:

1. Välttämätön, kiireellinen korjaus.
2. Korjattava ennen tuotantoa.
3. Korjaus ajan salliessa.
4. Mahdollista korjata tai jättää sellaiseksi.

Virheen vakavuus ja prioriteetti ovat toisiaan tukevia virheen laadusta suuntaa-antavia ominaisuuksia, joskin ne ovat testaajan subjektiivisia mielipiteitä. Beizer kirjoittaa kirjassaan [Beizer, luku 2], että virheiden vakavuuden ja prioriteetin mukaisesti virheiden korjauksesta päättäminen käy helpommin. Patton muistuttaa, että virheitä ei kuitenkaan tule luokitella sen mukaan, kuinka paljon resursseja niiden korjaaminen vaatii. Jollain tuhoisalla virheellä saattaa olla varsin nopea ja yksinkertainen ratkaisu, kun taas jonkin vain häiritsevän virheen korjaaminen saattaa vaatia suuritöisen urakan.

Virheen vakavuus riippuu myös ohjelmasta, josta se löydetään. Sovellukset voidaan jakaa äärimmäistä luotettavuutta vaativiin (kuten sairaalan ja ydinvoimalan järjestelmät) sekä vähemmän kriittisiin (kuten kalenterisovellukseen). **Ohjelman kriittisyys** vaikuttaa testaamisen suoritukseen ja erityisesti tarkkuuteen. Lisäksi päätökset testauksessa havaittujen virheiden korjauksesta tehdään aina testauksen kohteesta riippuen. Sairaalan koneen on siis ehdottomasti oltava luotettava, mutta lapsille suunnatun pelin tärkeimpiä ominaisuuksia ovat näyttävyys ja nopeus.

Ron Pattonin listassa (katso luku 2.2) virheen määritelmistä viides kohta on vaikeaselkoisempi kuin neljä ensimmäistä. Se ei liity ainoastaan virheettömyyteen, vaan myös **ohjelman laatuun**. Kautto listaa opinnäytteessään [Kautto] joitakin hyvän laadun mukaisia vaatimuksia ohjelmille. Ensinnäkin ohjelman on vastattava määrittelyään ja määrittelyn asiakkaan vaatimuksia. Käyttäjiä ajatellen ohjelman on oltava helppokäyttöinen ja yksinkertainen sekä vaivattomasti ylläpidettävä ja päivitettävä. Lisäksi sen on oltava turvallinen, suorituskykyinen sekä rasitusta kestävä. Kaikkia näitä ominaisuuksia voidaan ja pitää testata ennen ohjelman käyttöönottoa. Tosin on projekteja, joissa joitakin näistä ominaisuuksista ei arvosteta, eikä niitä mielletä virheiksi, vaan heikommiksi toiminnallisuuksiksi.

2.4 Testauksen tärkeys, harhaluulot ja vaatimukset

Ohjelmistosovellusten testaaminen ennen käyttöönottoa on eräs ohjelmistotuotannon tärkeimpiä osa-alueita, sillä puutteellisesti testattu ohjelma saattaa aiheuttaa yllättäviä kustannuksia ja tulonmenetyksiä. Tästä huolimatta testausta ei yleisesti arvosteta, vaan sitä pidetään helppona tienä tutustua projektiin ja testaajiksi asetetaan vähemmän kokeneita työntekijöitä. Testaus ei kuitenkaan automaattisesti sovi kaikille työntekijöille.

Testauksen tavoitteena pidetään virheiden löytämistä, mutta yksinään tämä ei takaa hyvää testaustulosta ja projektin onnistumista. Testauksessa virheellisyyden osoittaminen ei ole itsetarkoitus, vaan perimmäisenä pyrkimyksenä on virheiden löytämisen jälkeen saada koodi korjattua, sovellus virheettömäksi ja oikealla tavalla toimivaksi. Paakki kuitenkin painottaa luentomonisteessaan [Paakki, luku 2], että valitettavasti oikeellisuuden verifiointi ei ole mahdollista. Testaamisella ei nimittäin voida todistaa virheettömyyttä, vaan ainoastaan virheiden olemassaolo.

Testaus ei ole yksinkertainen ohjelmistoprojektin vaihe, ja Melzer [Melzer] päätyy siitä muun muassa seuraaviin johtopäätöksiin. Testauksen ensisijaisena tavoitteena on löytää virheitä. Toisaalta koskaan ei tulla löytämään kaikkia niistä, eikä näin ollen testauksella voida osoittaa ohjelman virheettömyyttä. Kaikissa vaiheissa on tuotteen määrittelyä käyttäen vertailtava, millainen sovellus on ja millainen sen pitäisi olla. Testaukseen vaikuttavat monet inhimilliset psykologiset tekijät, joten sitä suorittavilta ihmisiltä vaaditaan runsaasti kärsivällisyyttä. Hyvän testin on oltava helposti toistettavissa tarvittaessa. Projektin suorittamisen kannalta olisi etukäteen pyrittävä arvioimaan testauksen aikataulua, mutta yleisesti ei koskaan voida olla täysin varmoja, milloin testaus olisi parasta lopettaa.

Hyvältä testaajalta vaaditaan paljon. Ensinnäkin tarvitaan luovuutta, jotta mahdolliset ja mahdottomatkin virheet havaitaan. Erityisesti testien suunnittelussa tarvitaan asiantuntemusta bisneksestä ja kokemusta testaamisesta. Testauksen suunnitteluun ja toteutukseen tarvitaan riittävästi voimavaroja. Myös varsinaisessa testauksen toteutuksessa olisi hyvä olla mukana kokeneita testaajia, jotka osaavat käyttää oikeita testausmenetelmiä. Erityisesti hyvältä testaajalta vaaditaan kokeilunhalua, kärsivällisyyttä toistaa samaa yritystä, pyrkimystä sopivaan täydellisyyteen, perustelutaitoja virheiden korjaamisen takaamiseksi ja lisäksi diplomaattisuutta kertoa virheistä syyttämättä ohjelmoijaa. Ron Patton pyrkii kirjassaan [Patton, sivu 19] tiivistämään testaajan tehtävät seuraavasti: *”Ohjelmiston testaajan tavoitteena on löytää virheitä, löytää ne mahdollisimman aikaisin, ja varmistaa, että ne korjataan.”*

Tosin koko ajan projekteissa tehdään virheitä testaamisessa. Paakki listaa luentomateriaalissaan [Paakki, luku 10] joitakin näistä klassisina. Ensinnäkin testauksen suunnitteluun varataan ja käytetään aivan liian vähän aikaa, eikä ohjelmoijien oleteta testaavan omaa koodiaan. Kun sitten testaus aloitetaan, pidetään tärkeimpänä asiana havaita virheitä, vaikka pitäisi keskittyä tärkeisiin virheisiin. Käytettävyysongelmia ei pidetä virheinä lainkaan, eikä niitä varsinaisesti edes havaita, koska yhä enenevässä määrin testaus pyritään automatisoimaan eri tavoin suoritetun testauksen kustannuksia vertaamatta. Lisäksi testauksen laatua ja suoritusta arvioidaan aivan liian paljon koodin kattavuuden suhteen.

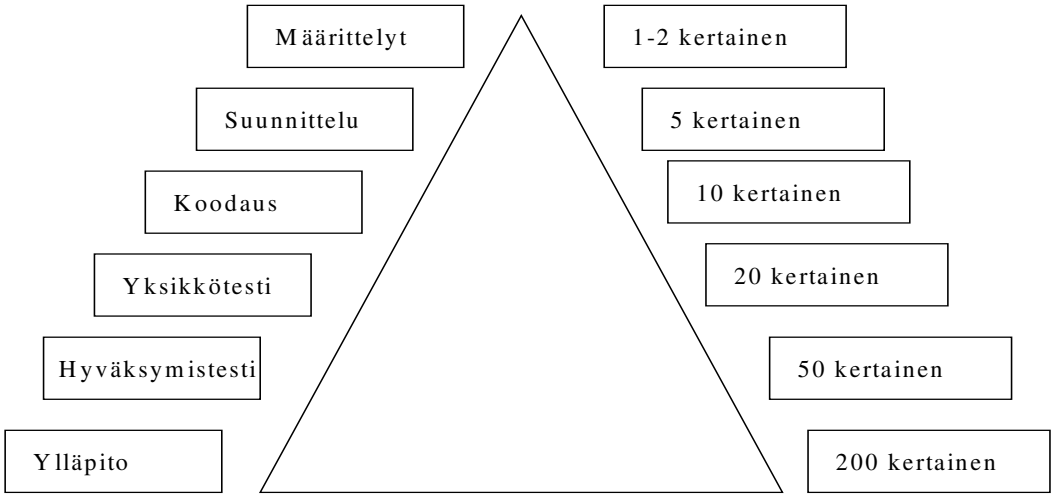
2.5 Virheiden kustannus ohjelmistokehityksen eri vaiheissa

Kaikissa ohjelmistoissa ja tuotteissa on virheitä. Beizer kuvaa kirjassaan [Beizer] tapausta, jossa sataa lausetta (engl. *statement*) kohden on yksi virhe. Yleisesti puhutaan virheiden määrästä koodiriviä kohden. Näin tekevät myös Haikala ja Märijärvi kirjassaan [Haikala et al., luku 15] ja arvelevat luvun olevan yksi virhe muutamaa kymmentä koodiriviä kohden. Jopa pitkään käytössä olleissa ohjelmissa on suunnilleen yksi virhe jokaista tuhatta ohjelmariviä kohden. Arvioiden mukaan noin viittä prosenttia virheistä ei koskaan havaita, sillä ohjelmissa olevien tilojen määrä on valtava ja joitakin funktioita ei niiden kaikilla arvoilla tulla käyttämään.

Laatuun pyrkiminen ja sen saavuttaminen on ohjelmistojen suurin kustannusten aiheuttaja. Suurin yksittäinen kustannus ovat virheiden etsintä ja korjaus sekä testien suunnittelu ja ajo. Beizer huomauttaa kirjassaan [Beizer, luku 1], että testaus ja debuggaus saattavat maksaa 50-80% tuotteen ensimmäisen toimivan version kustannuksista. Täten virheiden ennaltaehkäisyyn kannattaisi panostaa.

Mitä myöhemmässä projektin vaiheessa virhe löydetään, sitä kalliimmaksi sen korjaaminen muodostuu. Korjauskustannukset eivät nouse lineaarisesti, vaan lähinnä logaritmisesti projektin vaiheiden edetessä. Määrittelyvaiheessa virheiden kustannukset ovat alhaisimmat, kun ylläpitovaiheessa ne ovat huomattavan suuret verrattuna määrittelyvaiheeseen.

Santanen esittää luentomateriaalissaan [Santanen] kuvan 1, josta voidaan nähdä, miten kustannukset kohoavat projektin edetessä määrittelyvaiheesta eteenpäin. Vasemmalla puolella on projektin vaihe, jossa virhe havaitaan. Oikealla puolella on esitetty suhteellinen korjauskustannus. On siis mahdollista hävittää suuri osa projektin tuotosta, jos vakavat virheet pääsevät asiakkaalle asti.



Kuva 1. Virheiden kohoavat korjauskustannukset ohjelmistoprojektin eri vaiheissa [Santanen].

Kuvasta 1 voi huomata virheiden kustannusten nousevan jopa kaksisataakertaisiksi, jos ne huomataan ja korjataan vasta ylläpitovaiheessa. Patton esittää kirjassaan [Patton, luku 1] vieläkin suurempia lukuja. Kun määrittelyn kirjoitusvaiheessa havaitun virheen kustannukset ovat lähes olemattomat, niin saman virheen havaitseminen vasta koodauksen tai testauksen aikana nostaa kustannukset kymmen- tai satakertaisiksi. Jos taas virheen löytäjänä on asiakas, korjauskustannukset kohoavat jopa tuhatkertaisiksi.

2.6 Tutkielman vaatimukset ja tavoitteet

Tutkielmassa käsitellään ensin testauksen yleisiä periaatteita ja suhdetta ohjelmistoprojektiin kunnollisen yleiskuvan saamiseksi. Seuraavaksi keskitytään tutkielman varsinaiseen aiheeseen eli tarkastellaan erityisesti testauksen tasoja ja menetelmiä. Käytännön osuudessa tutkitaan eri testausmenetelmien käyttöä keskittyen systeemitestausvaiheeseen. Tutkielman tavoitteena on saada lisää tietoa testauksesta ja sen mahdollisuuksista. Teorian ja sen käytäntöön soveltamisen avulla voidaan varmasti parantaa testaajan taitoja.

Testaus käsitteenä rajataan useimmissa projekteissa testausryhmän suorittamaan tehtävään, systeemitestaukseen. Tässäkin tutkielmassa testausta käsitellään testausryhmän näkökulmasta. Tutkielmassa käytännön osuudessa käsitellään kenttämyyjien työkaluksi toteutetun järjestelmän testausta. Kyseinen projekti oli suhteellisen pieni ja testausryhmän suorittama testaus sijoittui pääasiallisesti systeemitestaukseen. Tehtävänä oli asiakas- ja palvelinsovelluksen systeemitestaus sekä koko systeemin integraatiotestaus. Testauksen alemman tason yksikkö- ja integraatiotestauksen suorittivat kehittäjät sekä hyväksymistestauksen suoritti asiakas. Täten tutkielman pääpaino käytännön osuuden käsittelyssä on systeemitestaukseen liittyvien tehtävien käsittelyssä.

Hyvä testaaja tarvitsee myös ohjelmointitaitoja. Täten kaikkien testaajien on hyvä uransa aikana toimia myös ohjelmoijan roolissa. Ohjelmoijien tehtäviin kuuluu yksikkö- ja integraatiotestauksen toteuttaminen, joten tässä tutkielmassa tutustutaan myös näihin osa-alueisiin. Hyväksymistestaus kuuluu yleisesti asiakkaan tehtäväksi, mutta yleensä asiakas tarvitsee apua sen toteuttamiseen, joten tämän testauksen tarkoituksen tunteminen on olennaista.

Tutkielman teoriaosuuden (katso luvut 2-5) tietoja sovelletaan käytäntöön järjestelmän testauksessa. Tutkielman käytännön osuudessa (katso luvut 6-7) selvitetään erityisesti testausmenetelmien käyttöä sovellusprojektissa. Tässä projektissa toteutetussa testauksessa käytettiin joitakin näistä menetelmistä. Yksi menetelmä ei ehkä vielä takaa parasta tulosta, vaan pitää soveltaa useampia menetelmiä. Tutkielman tarkoituksena onkin selvittää, miten nämä menetelmät soveltuvat projektin läpivientiin. Projektista saatujen tulosten ja kokemusten avulla pohditaan, mitä olisi kannattanut tehdä toisin ja millaisia menetelmiä tulisi käyttää paremman tuloksen saamiseksi tai ainakin virheiden löytämiseksi helpommin.

Toteutetun projektin testauksessa testitapaukset suoritettiin pääosin manuaalisesti. Testaustyökalujen käyttö onkin rajattu tutkielman aiheen ulkopuolelle ja sitä käsitellään vain lyhyesti.

3 Testaus osana ohjelmistoprojektia

Olennaisena osana ohjelmistoprojektiin kuuluu testaus, jonka päätavoitteena on löytää ohjelmistosta virheitä mahdollisimman aikaisin ja varmistaa niiden korjaaminen. Testaus koostuu periaatteessa kahdesta osasta: virheiden jäljityksestä ja löytyneiden epäkohtien korjauksesta. Testaus suoritetaan V-mallin mukaisesti monella eri tasolla, joita ovat yksikkö-, integraatio-, systeemi- ja hyväksymistestaus. Luvun pääasialliset lähteet ovat [Patton] ja [Haikala et al.].

3.1 Ohjelmistoprojektin vaiheet

Ohjelmistoprojektit läpiviedään yleensä jonkin valitun prosessimallin mukaisesti. Yleisimmissä prosessimalleissa ohjelmistoprojektin vaiheita ovat esitutkimus, määrittely, suunnittelu, toteutus, testaus sekä käyttöönotto ja ylläpito.

Ohjelmiston kehitysprosessia voidaan kuvata elinkaarella, joka esittää ohjelmiston vaiheet kehittämisen aloittamisesta aina käytöstä poistamiseen asti. Elinkaarta kuvataan vaihejako- eli prosessimalleilla. Näistä eri vaihejakomalleista on useita muunnelmia, mutta yleisinä osina kaikissa on erotettavissa määrittely-, suunnittelu- ja toteutusvaiheet. Tämä luku perustuu pääasiassa lähteisiin [Haikala et al., luku 2] ja [Pressman, luku 11].

3.1.1 Esitutkimus

Ohjelman elinkaari alkaa yleensä esitutkimuksella. Tällöin selvitetään, mitkä ovat asiakkaan ja käyttäjien vaatimukset kehitettävästä tuotteesta. Kyseisen vaiheen tarkoitus on määrittää yleiset järjestelmätason vaatimukset päätavoitteista. Tällaista määrittelyä kutsutaan asiakasvaatimukseksi, koska se huomioi asiakkaan ja käyttäjien tarpeet, mutta ei vielä mieti varsinaista toteutusta. Tavoitteena on saada vastaus kysymykseen, miksi tällainen tuote olisi toteutettava tai päinvastoin, miksi sitä ei kannata toteuttaa. Asiakkaan vaatimukset pyritään selvittämään muun muassa erilaisin haastatteluin ja aivoriihin, jotta saadaan ymmärrettyä ja kirjattua ylös asiakkaan ja käyttäjien todelliset tarpeet ja toiveet.

3.1.2 Määrittely

Esitutkimusta ohjelmiston elinkaaressa seuraa ohjelmiston vaatimusmäärittelyvaihe (engl. *software requirements analysis*). Sillä pyritään tarkentamaan esitutkimuksessa kerättyä tietoa, selventämään tavoitteita ja projektin kannattavuutta. Pressman huomauttaa kirjassaan [Pressman, luku 11.1] tämän vaiheen vastaavan kysymykseen, mitä tuotetaan.

Joskus myös esitutkimus mielletään osaksi tätä vaihetta, sillä molemmat pyrkivät omilla tavoillaan keräämään asiakas- ja käyttäjävaatimuksia. Vaatimusmäärittelyssä tiedonkeräys kohdistuu itse tuotteeseen, pyrkien muuntamaan asiakasvaatimukset täsmällisiksi ohjelmistovaatimuksiksi. Jotta ymmärrettäisiin tuotteen tarkoitus, on selvitettävä vaaditut toiminnallisuudet, tekniset vaatimukset ja rajoitukset, rajapinnat sekä käyttöliittymän ulkoasu ja toiminnot.

Kaikki määrittelyssä selvitetyt asiat dokumentoidaan toiminnalliseksi määrittelyksi, joka esitetään ja hyväksytetään asiakkaalla. Pressman huomauttaa, että yksityiskohtainen määrittely ei vielä ole mahdollista, koska asiakas saattaa olla epävarma tavoitteista ja kehittäjä tuotteen toteutuksen keinoista. Määrittelyn oikeellisuuden parantamiseen ja varmentamiseen tässä vaiheessa tulee käyttää erilaisia haastatteluja, valmiita kysymyksiä ja vaikkapa prototyypitystä. Vaatimusmäärittelyä voidaan pitää elinkaaren tärkeimpänä vaiheena, sillä huonoa määrittelyä ei voida pelastaa enää suunnittelulla tai koodauksella, eikä tuote miellytä asiakasta ja käyttäjiä tai se ei vastaa heidän tarpeitaan.

3.1.3 Suunnittelu

Määrittelyä elinkaaressa seuraa suunnitteluvaihe (engl. *design*), jossa Pressmanin [Pressman, luku 13] mukaan vastataan kysymykseen, miten sovellus toteutetaan. Tämä on moniportainen vaihe, joka on jakautunut neljään osaan: tiedon rakenteeseen, ohjelmiston arkkitehtuuriin, käyttöliittymän esittämiseen ja toiminnallisiin yksityiskohtiin eli moduulien suunnitteluun. Elinkaaren edellisen vaiheen määrittelyistä muodostetaan ohjelmiston tai järjestelmän suunnittelu ja dokumenttina saadaan tekninen määrittely.

Suunnitteluvaiheessa tuotettujen dokumenttien laatu kannattaa tarkastaa ennen toteuttamisen aloittamista. Haikala ja Märijärvi muistuttavat kirjassaan [Haikala et al., luku 3.7], että sovelluksen laadun takaamiseksi jo suunnitteluvaiheessa pitäisi huolehtia selkeydestä, ymmärrettävyydestä, tehokkuudesta, luotettavuudesta, ylläpidettävyydestä ja siirrettävyydestä. Tähän tavoitteeseen pääsemiseksi on erityisen tärkeää osittaa arkkitehtuurisuunnittelu sopivan kokoisiksi moduuleiksi.

3.1.4 Ohjelmointi ja testaus

Suunnittelun jälkeen on aika siirtyä itse toteutukseen eli ohjelmointiin, joka toteutetaan suoraan suunnitteluvaiheessa tuotettujen kuvausten perusteella. Kuvauksia tulee noudattaa ja tarvittaessa niitä voidaan muokata. Yksi suuri virhelähde on se, ettei tuotteen määrittelyyn ja suunnitteluun tutustuta riittävästi kehitystyön aikana. Tähän vaiheeseen liittyy myös virheenjäljitys, joka siis ei ole testauksen vaihe. Usein myös yksikkötestaus liitetään ohjelmoinnin osaksi.

Testaukseen voidaan periaatteessa siirtyä, kun ohjelmasta on ensimmäinen toimiva ja annetut vaatimukset täyttävä versio. Testauksessa pyritään käymään läpi ohjelman kaikki osa-alueet, havainnoimaan virhetilanteet sekä raportoimaan ja korjaamaan ne.

3.1.5 Käyttöönotto ja ylläpito

Elinkaaren lopussa alkaa käyttöönotto ja ylläpito (engl. *operation and maintenance*), joka yleensä on sovelluksen elinkaaren pisin vaihe. Asiakkaalle siirtymisen jälkeen ohjelmistoon tehdään monesti muutoksia, koska ilmenee lisää virheitä, asiakas haluaa uudistaa tiettyjä ominaisuuksia tai muuttunut ympäristö vaatii uutta toimintatapaa. Syynä voi myös olla, että asiakas ei ole ensimmäisessä versiossa ottanut huomioon loppukäyttäjiä ja heidän tarpeitaan.

Tosin Haikala ja Märijärvi huomauttavat kirjassaan [Haikala et al.], että ohjelmistotuotteissa ei tällaista varsinaista ylläpitovaihetta esiinny niin usein kuin muiden alueiden tuotteissa. Syynä on, että useat sovellukset ovat juuri tiettyä asiakasta varten tehtyjä ja muutokset tehdään sovelluksen seuraavassa versiossa.

3.2 Ohjelmistoprojektin vaihejakomalleja

Vaihejako- eli prosessimalli määrää, miten ohjelmiston koko elinkaari tai ainakin sen toteutusprosessi jaetaan vaiheisiin. Vaihejakomalleja on olemassa useita, joilla kaikilla on omat piirteensä sekä hyvät ja huonot puolensa. Tässä luvussa vaihejakomalleista esitetään kolme erilaista: big-bang, vesiputousmalli ja spiraalimalli.

3.2.1 Big-bang -malli

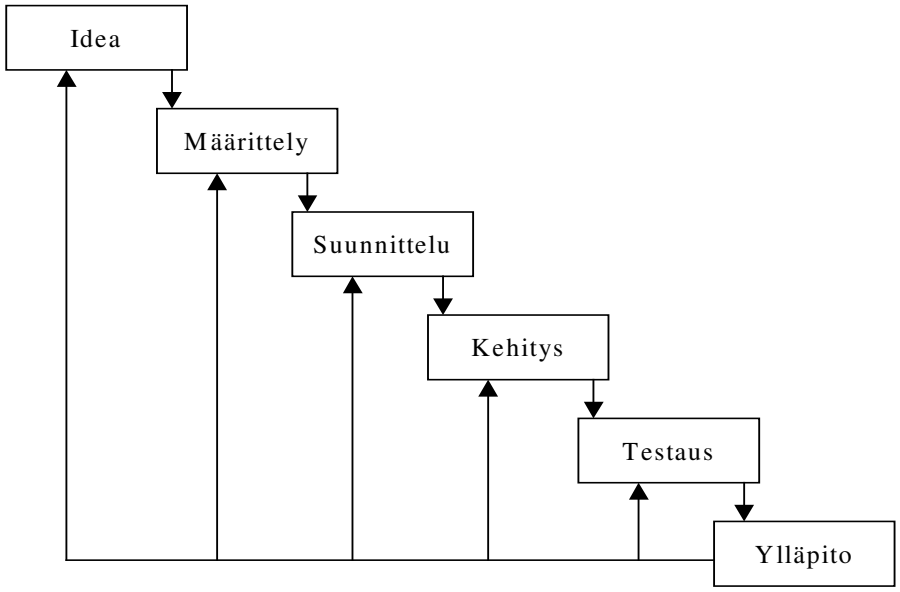
Patton esittelee kirjassaan [Patton, luku 2] yksinkertaisimpana mallina niin sanotun **big-bang** -mallin, joka kuvaa ohjelmistoprosessia maailman alkuräjähdyksen kaltaisena tuotoksena. Ohjelmisto syntyy kokoamalla ihmiset, rahaa ja aikaa yhteen, ja odottamalla räjähdystä.

Tässä ei suunnitella, dokumentoida, eikä tiedetä tulosta ennen sovelluksen valmistumista. Mallissa ei ole paluuta aiempiin vaiheisiin, joten testaus, jos sellaista suoritetaan, on vain virheiden raportointia asiakkaiden tietoon. Mallin noudattaminen on varsin yksinkertaista, mutta yleensä lopputulosta ei voida taata, joten hyvänä mallina tätä ei voida pitää.

3.2.2 Vesiputousmalli

Tavanomaisin projektin vaihemalleista on vesiputousmalli, joka on yksinkertainen ja samalla kuitenkin toimiva. Mallissa liikutaan portaittaisesti vaiheesta toiseen aina projektin alkuideasta sen elinkaaren loppuun. Jokaisen askeleen lopussa tarkastellaan, joko ollaan valmiita seuraavaan vaiheeseen vai tulisiko samaa vaihetta jatkaa ennen siirtymistä. Tämä siirtymisen tarkastelu on tärkeää, vaikka jo alkuperäisen mallin suunnittelija mahdollisti mallissa paluun takaisinpäin. Pressman kuitenkin huomauttaa kirjassaan [Pressman, luku 2], että tosiasiassa useissa projekteissa mallia sovelletaan lineaarisena, jolloin paluuta takaisin edellisiin vaiheisiin ei käytetä.

Vesiputousmallissa (katso kuva 2) suunnittelulle varataan paljon käytössä olevasta ajasta kehityksen ollessa vain yksi kuudesta askeleesta. Nämä askeleet ovat irrallisia, eikä päällekkäisyyksiä niiden tehtävissä ole. Patton huomauttaa kirjassaan [Patton, luku 2], että malli on jokseenkin rajoittava. Toisaalta se toimii hyvin projekteissa, joissa vaatimukset ymmärretään hyvin ja niitä myös noudatetaan.



Kuva 2. Vesiputousmalli.

Mallin mukaisesti toimittaessa on testaukseen saavuttaessa suunnittelu tehty ja testiryhmä näkee selkeästi, mitkä ovat virheitä ja mitkä ominaisuuksia. Tosielämässä asiakas harvoin tietää kaikkia vaatimuksiaan projektin alussa, joten valitettavasti suunnitelmat tarkentuvat vasta myöhemmin. Toisena huonona puolena tässä mallissa on testauksen suorittaminen vasta projektin lopussa, joten virheiden korjauskustannukset ovat huomattavasti suuremmat kuin määrittely- tai suunnitteluvaiheessa.

Kuitenkin tämä malli on eniten käytetty ja se ohjaa kehitysprosessia määrittäen kaikille tehtäville selkeän järjestyksen. Tämä malli on siis projektin vetäjien osalta yleensä helpoiten hallittavissa. Vesiputousmallista on johdettu monia muunnoksia ja kehitetty kokonaan uusia malleja.

3.2.3 Spiraalimalli

Vaihejakomalleista paljon käytetty on myös spiraalimalli (engl. *spiral model*), joka on tehokas malli kehitysprosessissa. Pääajatuksena on ensin suunnitella tärkeimpiä kohtia, toteuttaa ne, saada palaute asiakkaalta ja sen jälkeen aloittaa uusi kierros tarkentaen prosessia.

Mallin vaiheita toistetaan peräkkäin, kunnes tuote on valmis. Pattonin mukaiset [Patton, luku 2] askeleet ovat seuraavat:

1. Määrittele vaihtoehdot ja rajoitukset.
2. Tunnista ja ratkaise riskit.
3. Arvioi vaihtoehdot.
4. Kehitä ja testaa nykyinen taso.
5. Suunnittele seuraava taso.
6. Päätä lähestymistapa seuraavalle tasolle.

Spiraalimalli siis lainaa ominaisuuksia muilta malleilta. Testaajat ovat mukana kaikissa projektin vaiheissa nähden vaatimukset sekä halutut tulokset. Patton näkee kirjassaan [Patton, luku 2] tämän mallin hyvänä, koska sen mukaisesti toteutetussa projektissa virheet havaitaan jo aikaisessa vaiheessa, joten korjauskustannukset pysyvät alhaisina. Testaus tapahtuu elinkaaren kaikissa vaiheissa. Tällöin projektin viivästyessä aikataulua ei välittömästi tiukenneta testaukselta, vaan ainoastaan viimeinen testien läpikäynti ja oikeellisuuden varmistus jäävät kiireiseen loppuun.

3.3 Testausprosessin V-malli

Ohjelmiston laatu pyritään selvittämään ja takaamaan validoinnin ja verifioinnin (kelpoistaminen ja todentaminen) avulla. Tällainen validoinnin ja verifioinnin prosessi käydään läpi muun muassa paljon käytetyssä testausprosessin V-mallissa.

Koikkalaisen [Koikkalainen, luku 1] mukaan **validointi** pyrkii tarkastamaan, toimiiko sovellus asiakkaan alkuperäisten toivomusten ja asiakkaalle annettujen lupausten mukaisesti sekä sopiiko sovellus käyttötarkoitukseensa. Tässä arvioinnissa käytetään hyväksi vaatimuksia, jotka on toivottavasti kirjoitettu tarkasti ja yksiselitteisesti validoinnin toimivuuden takaamiseksi. Koikkalainen huomauttaa luentomonisteessaan [Koikkalainen, luku 2], että huonosti määritellyt vaatimukset antavat paljon liikkumavaraa validoinnissa, mistä johtuen ohjelma ei ehkä vastaa asiakkaan vaatimuksia.

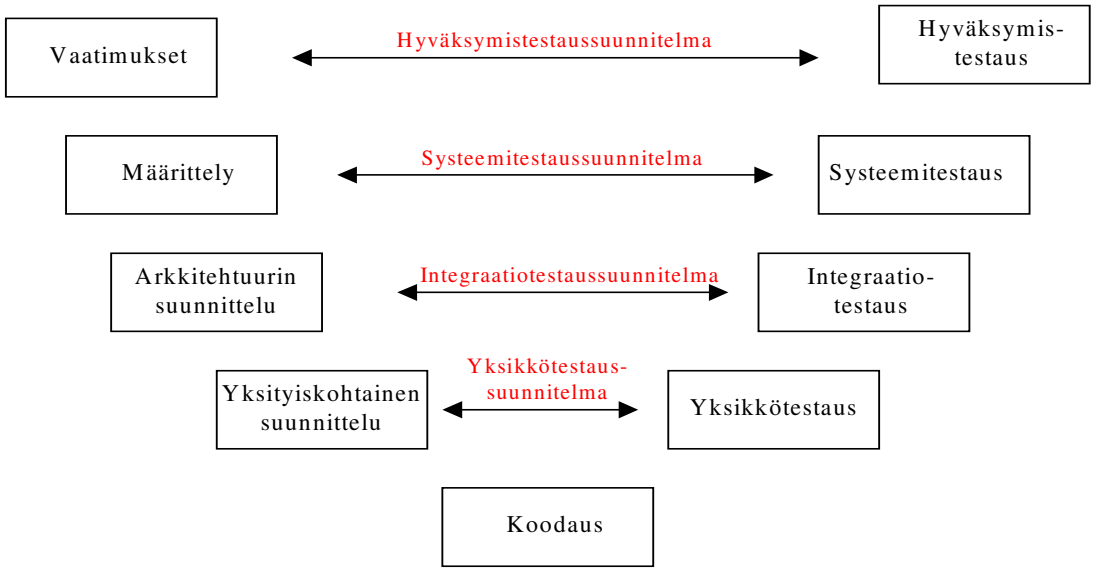
Koikkalainen määrittää luentomonisteessaan [Koikkalainen, luku 1], että **verifiointi** selvittää ohjelman toiminnan oikeellisuuden, verraten tuotetta sen spesifikaatioon. Verifiointi toteutetaan yleensä sovelluksen testauksena.

Validoinnin ja verifioinnin erot voidaan määritellä periaatteessa varsin yksinkertaisesti. Validointi pyrkii vastaamaan kysymykseen, onko kehitetty oikea tuote. Verifioinnilla taas testataan, onko tuote kehitetty oikein.

3.3.1 Perusidea

Testauksen kannalta V-malli on yksi käytetyimmistä prosessimalleista. Se soveltuu projekteihin, joissa sovelluskehitys tapahtuu vesiputousmallin mukaisesti. Ohjelmistoprojekti etenee V-mallin mukaan seuraavassa järjestyksessä tarkentuen asteittain: vaatimusten määrittely, vaatimusten analysointi, arkkitehtuurisuunnittelu, moduulisuunnittelu ja varsinainen toteutus. Itse testauksen vaiheet etenevät ohjelmistoprojektia vastaavasti suoraviivaisesti: yksikkötestaus, integraatiotestaus, systeemitestaus ja hyväksymistestaus.

Kuva 3 esittää, kuinka validoinnin ja verifioinnin vaiheet liittyvät ohjelmiston ja testauksen dokumentointiin ja toteutukseen. Oikealla puolella esitetyt testauksen vaiheet todentavat vasemmalla esitetyt toteutukset. Koska testauksen aikana suoritettavat systeemi- ja hyväksymistestaukset ovat osana asiakkaan vaatimuksia, nämä testauksen osat validoivat sovelluksen.



Kuva 3. Testausprosessin V-malli.

3.3.2 V-malli käytännössä

V-mallissa testausvaiheet on liitetty yhteen suunnitteluvaiheiden kanssa. Periaatteessa kunkin testausvaiheen suunnittelu voidaan aloittaa, kun sitä vastaavat vaatimus- ja suunnitteluvaiheet on toteutettu. Koikkalainen kirjoittaa luentomonisteessaan [Koikkalainen, luku 2], että testauksen suunnittelu tehdään kullekin tasolle sitä vastaavalla suunnittelutasolla, mutta käytännössä siinä voidaan käyttää kaikkien muidenkin vaiheiden tuotoksia hyväksi. Testauksessa saatuja tuloksia verrataan kuhunkin vaiheeseen liittyviin vaatimusdokumentteihin.

Vaatimukset kokonaisjärjestelmän toimintaan tulevat asiakkaalta, mistä johtuen nämä vaatimukset ovat melko konkreettisia. Paakki kirjoittaa luentomonisteessaan [Paakki, luku 2], että kuljettaessa alaspäin kohti yksikkösuunnittelua vaatimusten abstraktiotaso ja yksityiskohtaisuus kasvavat ja suunnittelu muuttuu teknisemmäksi.

Testauksen aluksi yksikkötasolla testaus kohdistuu yksittäisten moduuleiden toimintaan. Integraatiotasolla testataan moduuleiden väliset yhteydet ja systeemitestauksen aikana systeemin toiminta kokonaisuutena. Hyväksymistestauksessa toimitaan asiakkaan näkökulmasta. Huomattavaa on, että verrattuna tuotteen kehitykseen testauksen abstraktiotaso vaihtuu juuri päinvastaisessa järjestyksessä, aloittaen korkeilta abstraktiotasoilta aina enemmän konkretisoituen.

3.3.3 Edut

Etuna V-mallissa on mahdollisuus toteuttaa suunnitelmat ja dokumentaatiot aikaisessa vaiheessa, jo ennen varsinaista koodin kirjoittamista. Lisäksi kehittäjät yleensä kirjoittavat parempaa koodia tietäessään testaussuunnitelmasta, tulevasta testauksesta ja siten heidän koodinsa kontrolloinnista verrattuna siihen, että näitä jälkeen päin tapahtuvia tarkastuksia ei olisi.

3.3.4 Ongelmat

V-mallia käytetään runsaasti, mutta sen käytössä ilmenee ongelmia. V-mallin vaiheet vaatimusten määrittelystä koodaamiseen etenevät vesiputousmallin ohjaamassa järjestyksessä. Vesiputousmallia väheksytään monissa yhteyksissä, mutta kuitenkin se on eräs testauksen peruselementeistä. Sen ongelmat kuitenkin siirtyvät suoraan V-mallin mukaisesti suoritettavaan testaukseen.

Yhtenä ongelmana V-mallin käytössä on se, että kehittäjä voi pyrkiä kehittämään koodinsa päätavoitteenaan testitapausten läpäisy. Paakki toteaa kirjassaan [Paakki, luku 2], että ratkaisu tähän ongelmaan on helppo, jos testaajina on muita kuin kehittäjiä. Jos tällöin ei näytetä testitapauksia kehittäjille, he eivät osaa suunnata kehitystä juuri testauksessa toimivaksi. Joissain projekteissa valitettavasti kehittäjät testaavat itse kaiken koodin.

Usein V-mallin käyttäjät erottavat toisistaan testin suunnittelun ja toteuttamisen, mikä ei ole täysin tarkoituksenmukaista. V-mallissa testaus jaetaan tarkasti eri tasoille. Yksikkötestaus suoritetaan, kun yksiköt ovat valmiita. Yksiköt tarvitsevat toimiakseen yleensä tynkiä ja ajureita (katso luku 4.1.2). Hankaluutena onkin, että niiden kehittäminen lisää kustannuksia, ylläpito voi olla vaikeaa ja testauksen avuksi kehitetty järjestelmä saattaa peittää osan virheistä.

V-mallissa integrointitestaus suoritetaan, kun tarvittavat järjestelmäosiot on testattu ja yhdistetty. Toki integrointitestauksen suoritustapaa ei ole kiinnitetty, joten se voidaankin toteuttaa kokoavana tai jäsentävänä (katso luvut 4.2.3, 4.2.4). Marick huomauttaa artikkelissaan [Marick 1999], että joissain tapauksissa näiden vaiheiden yhdistäminen tai ainakin niiden lomittaminen voisi olla järkevää esimerkiksi kustannusten ja ajankäytön suhteen. Mutta hänen mukaansa tällaiseen omaan ajatteluun ja sen myötä tapahtuvaan mallin muokkaamiseen V-malli ei rohkaise.

3.4 Testausprosessi

Testausprosessi jakautuu edelleen vaiheisiin, kuten mikä tahansa ohjelmistokehityksen vaihe. Prosessia suunniteltaessa on huomioitava työntekijöiden osaamisalueet, kokemukset, roolit ja aikataulut muiden projektien suhteen. Lisäksi on mietittävä, mitkä menetelmät ovat suositeltavia ja sopivia juuri tähän projektiin sekä voisivatko työkalut tukea tätä testausprosessia. Tietysti on huomioitava myös asiakas, loppukäyttäjät, muut toimijat ja aikataulut. Tosin suunniteltaessa ei voida ottaa huomioon kaikkea, sillä testausprosessin aikana siihen vaikuttaa monia ennalta-arvaamattomia muuttujia, kuten pidentynyt kehitysaikataulu tai yllättävä määrä virheitä.

Jos testausprosessi etenee V-mallin mukaisesti, testausvaihe alkaa määrittelyllä ja suunnitelmien kirjoittamisella. Nämä suunnittelut, kirjoitettu koodi ja testiaineisto syötteenä suoritetaan testaus. Kun testauskierroksella havaitaan virheitä, ne raportoidaan ja annetaan kehittäjille korjattavaksi. Korjauksen jälkeen suoritetaan uudelleentestaus, jonka tuloksena korjaukset hyväksytään tai hylätään virhekohtaisesti. Lopulta testikierros hyväksytään tai ainakin päätetään ja päästään aloittamaan uusi kierros.

Riippuen korjauksista, uudelleentestausta tai uutta testikierrosta varten saatetaan tarvita muutoksia testitapauksiin. Patton kirjoittaa kirjassaan [Patton, luku 3], että erityisesti toistettaessa sama testaus aina uudelleen, virheet tulevat immuuneiksi näille testeille. Tällöin ainoastaan uusilla tapauksilla on mahdollista havaita vielä olemassaolevia virheitä.

3.4.1 Testauksen ajoitus

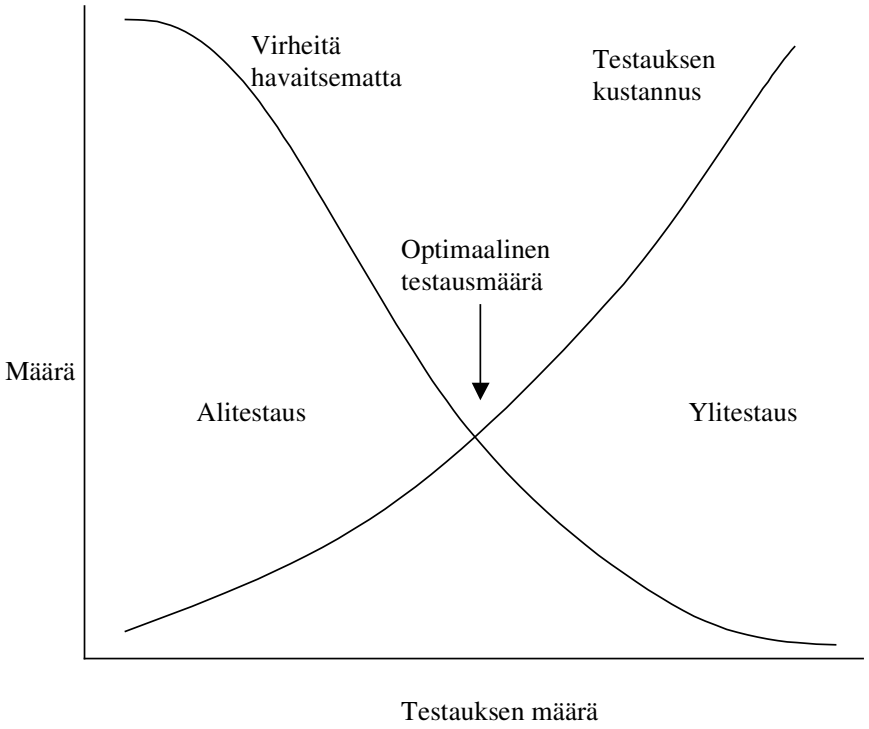
Aloittamalla testaus ja verifiointi niin aikaisin kuin mahdollista säästetään paljon vaivaa ja rahaa. Kuten luvussa 2.5 esitettiin, mitä aikaisemmassa vaiheessa virheitä löydetään, sitä helpommin ja halvemmalla ne ovat korjattavissa. Tämän takia kullekin testauksen tasolle on hyvä laatia testaussuunnitelma jo projektin alussa, samaan aikaan kuin vastaavan tason määrittely tiedetään. Tähän on monia syitä. Päällimmäisin syistä on tuotteen toiminnallisuuden ja laadun varmistaminen sekä tuotekehityksen vauhdittaminen. Toki tässä on oletettava, että tuotteen määrittely tehdään lopulliseksi jo projektin alussa. Jos tuotteen määrittelyjä muokataan myöhemmin, niistä koituu lisätyötä niin kehittäjille, testaajille kuin dokumentaatioiden ylläpitäjille. Samalla myös tuotteen laatu saattaa kärsiä.

Yleisesti aikataulun suunnittelu on tärkeää. Varsinkin projektin koon kasvaessa ja monimutkaistuessa yhä useammat ihmiset vaikuttavat projektin kulkuun. Patton ehdottaa kirjassaan [Patton, luku 2], että tällainen aikatauluttaminen voidaan tehdä yksinkertaisilla tehtävälistoilla tai varsin yksityiskohtaisella Gantt-kaaviolla. Aikataulun avulla tiedetään, mitä ollaan saatu valmiiksi, mitä on vielä toteuttamatta tai kesken ja paljonko aikaa loppuihin tehtäviin voidaan käyttää. Tosin aina on olemassa vaara ennalta-arvaamattomista muuttujista, jotka saattavat lisätä työmäärää. Usein myös projektien aikataulut arvioidaan liian optimistisesti, joten aikataulua suunniteltaessa kannattaisi mieluummin lisätä loppuun hiukan lisää aikaa kuin yrittää säästää kustannuksissa vähentämällä toteutusaikaa.

3.4.2 Testauksen lopettaminen

Testausta aloitettaessa kirjoitetussa testausstrategiassa on määriteltävä, mitkä ovat sopivat kriteerit lopettaa testaus ja todeta sovellus kelvolliseksi. Tämän määritteleminen ei ole yksinkertaista. Eräs tapa on katsoa testaus suoritetuksi, kun virheitä ei enää löydy. Yleisesti ohjelmat eivät kuitenkaan ole niin pieniä, etteikö niistä enää löytyisi lainkaan virheitä.

Patton huomauttaa kirjassaan [Patton, luku 3], että aina on olemassa ristiriita virheettömän tuotteen kehittämisen ja ajankäytön suhteen. On valittava ajankohta, jossa tuotteessa ei ole liikaa virheitä luovutettavaksi käyttöön ja toisaalta bisneksen puolesta ei viivytellä liikaa, vaan on hyvä aika luovuttaa tuote asiakkaalle. Patton kuvaa tätä kuvan 4 kaaviolla, jossa esitetään liian vähäisen ja liiallisen testaamisen suhdetta verrattuna kustannuksiin ja löydettyihin virheisiin. Patton kirjoittaa, että jossain näiden välillä on jokaisesta sovellusprojektista löydettävissä optimaalisin testausmäärä.



Kuva 4. Optimaalinen testauksen määrä [Patton, luku 3].

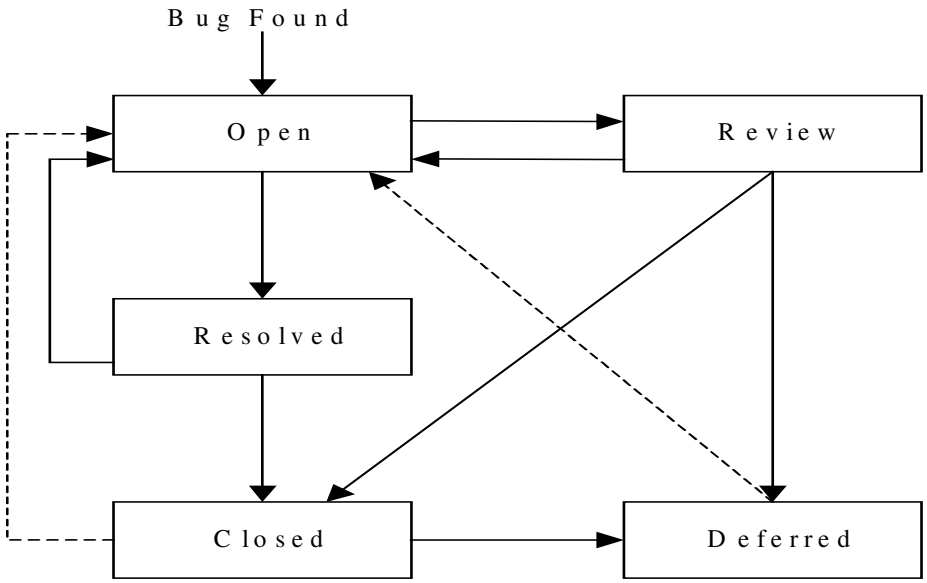
Tietyillä säännöillä voidaan pyrkiä määrittämään hyvä lopetuskohta. Melzer huomauttaa artikkelissaan [Melzer], että kriteerin lopetuskohdalle on oltava sellainen, että se kannustaa testaamaan parhaalla mahdollisella tavalla, eikä testauksen laatu kärsi. Mahdollista on pyytää testaajia lopettamaan, kun virheitä ei enää löydy säännöllisesti tai kun on testattu vähintään m päivää ja löytynyt enää n virhettä. Kysymys kuuluukin, miten määrittää ennalta löydettävien virheiden lukumäärä. Jos lukumäärä on matala, ei sen enempää virheitä halutakaan etsiä. Jos virheiden lukumäärä on liian korkea, testaajat turhautuvat, eikä tulosta synny.

Lopetuskriteerillä pitäisi Melzerin mukaan olla testauksen tasoa nostava vaikutus. Eräs mahdollisuus on tietenkin palkata testaajat yrityksen ulkopuolelta, jolloin työnlaatu voi olla parempi, mutta tähän ei toki useimmissa projekteissa päädytä. Jos saavutetaan yhteisymmärrys virheitten lukumäärästä tai käytettävistä testauspäivistä, on Pattonin mukaan huomattava, että käytetty aika ei ole mitenkään verrannollinen testauksen tulokseen. Hyvin suunnitellulla työllä voidaan testata tarkemmin samassa ajassa kuin vain umpimähkään kokeilemalla suoritetulla testauksella.

Lopetuskriteerit olisi pyrittävä löytämään kaikille testauksen vaiheille. Vasta lopetuskriteerien täytyttyä olisi mahdollista siirtyä testauksessa seuraavaan vaiheeseen. Käytännössä projekteissa usein päädytään testauksen lopettamiseen aiemmin valitun päivämäärän mukaisesti. Tosin virheellisen tuotteen myynti aiheuttaa ylläpitokustannuksia ja käytännössä tällainen tapa vain siirtää ongelman tulevaisuuteen. Ongelmaan ei tunnu olevan tarkkaa vastausta, mutta pyrkimys on käyttää käytännössä sovellettuja tapoja ja mahdollisten laskukaavojen perusteella saatuja tuloksia.

3.4.3 Virheen korjausprosessin vaiheet

Virheen elinkaari (katso kuva 5) alkaa virheen löytämisestä ja raportoinnista. Raportin mukaan kehittäjä korjaa virheen, kirjaa kommenttinsa raporttiin ja lähettää tämän testaajalle. Testaaja testaa uudelleen ja hyväksyy korjauksen. Tämä on yksinkertaisin ja ehkä tavallisin versio virheen elinkaaresta, mutta osalla virheistä on huomattavasti mutkikkaampi elinkaari. Ensin virhe on tilassa avattu (engl. *open*), josta se siirtyy mahdollisen katselmuksen (engl. *review*) jälkeen korjaukseen. Lopulta kehittäjien korjattua virheen (engl. *resolved*) se voidaan hyväksyä ja sulkea (engl. *closed*). Joissain tapauksissa virhe saatetaan asettaa odottamaan myöhempää korjausta tai hylätä kokonaan tämän projektin osalta (engl. *deferred*).



Kuva 5. Virheen elinkaari [Patton, luku 18].

Kuva 5 esittää Pattonin kuvauksen [Patton, luku 18] mahdollisista virheen tiloista sen elinkaaren aikana. Virheen elinkaari alkaa sen havaitsemisesta ja virheraportin avaamisesta, jonka jälkeen virheraportti siirretään kehittäjälle korjattavaksi. Kehittäjä saattaa kuitenkin pitää virhettä turhana korjata ja pyytää esimieheltä lupaa sulkea sen. Sulkemisen jälkeen testaaja havaitsee virheen uudelleen sovelluksen tärkeässä toiminnossa ja pyytää esimieheltä lupaa avata se uudelleen korjattavaksi. Esimies on samaa mieltä ja pyytää kehittäjältä korjausta. Näin virhe saattaa kulkea monta kierrosta eri osapuolten välillä ennen kuin kaikki hyväksyvät sen lopullisen tilan. Kuvassa 5 katkoviivalla esitetyt kulut suljetuista tiloista uudelleen avatuiksi ovat harvinaisia, mutta saattavat kuitenkin tapahtua jatkuvan virheiden etsinnän tuloksena.

Patton listaa kirjassaan [Patton, sivut 42-43] syitä sille, miksi kaikkia virheitä ei korjata. Osan korjaaminen päätetään siirtää myöhempään versioon tai muutosten toteutusvaiheeseen. Joskus virheen korjaamiseen ei yksinkertaisesti ole riittävästi aikaa, koska sovellus on saatava tuotantoon tiettynä päivänä tai muita, tärkeämpiä virheitä, on korjattava ensin. Toisaalta jotkut raportoiduista virheistä eivät olekaan varsinaisia virheitä, vaan ominaisuuksia (engl. *feature*), johtuen väärinymmärryksestä tai tuotteen määrittelyn äskettäisestä muuttumisesta.

Joskus virheiden korjaaminen on riskialtista, koska se voi tuottaa muita virheitä, joilla on kauaskantoiset seuraukset. Varsinkin projektin kiireisessä loppuvaiheessa voi olla parempi jättää tunnettu virhe sovellukseen kuin riskeerata uusien virheiden ilmaantumisella. Lisäksi Patton toteaa, että joidenkin virheiden korjaaminen ei ole vaivan arvoista. Jos ne ilmenevät vain harvoin tai sovelluksessa on olemassa vaihtoehtoinen tapa suorittaa asia, saatetaan liiketoiminnan takia jättää virhe korjaamatta.

3.4.4 Roolit testauksessa

Testausta suunniteltaessa on päätettävä rooleista. Pressman kirjoittaa kirjassaan [Pressman, sivut 505-506] joissain tapauksissa testaajina käytettävän kehittäjiä siksi, että he tuntevat koodin parhaiten. Mutta arvattavasti kehittäjien suurin tavoite on osoittaa oma koodinsa toimivaksi, eikä virheiden etsintään siten kohdistu suurta mielenkiintoa. Lisäksi omalle koodille tulee helposti sokeaksi, eikä siitä pysty havaitsemaan virheellisiä osia.

Joissain tapauksissa testaajan rooliin valitaan yksi henkilö ryhmästä. Hän on monesti työntekijä, joka ei ole hyvä kehittämisessä, eikä ehkä tunne testaamistakaan hyvin. Suurehkoihin tai kriittisiin projekteihin sopii myös kalliimpi vaihtoehto eli palkataan testaajat erillisestä testausorganisaatiosta. Tosin vieraan testaajan kanssa saattaa ongelmana olla kommunikointi, aikataulu ja mahdolliset ennakkoluuloiset asenteet.

Testaustehtävistä sovelluskehittäjän rooliin kuuluu yksikkötestauksen suorittaminen. Monissa tapauksissa kehittäjät testaavat myös integraatio-osuuden varmistaen näin osien yhteisen kommunikoinnin toimimisen. Vasta tämän jälkeen itsenäinen testausryhmä aloittaa varsinaisen testauksen, jo aiemmin tekemiensä valmistelujen ja tarvittavien suunnitelmien avulla. Pressman muistuttaa, että ryhmän mukaantulon jälkeenkin kehittäjät hoitavat tarvittavia tehtäviä ja jokaisen testikierroksen päätyttyä heidän tehtävänsä on korjata virheet.

Testauksen rooleihin tarvitaan eritasoisia työntekijöitä. Nimitykset näille tehtäville vaihtelevat asiayhteyksittäin. Tässä esitetään Pattonin mukainen jako [Patton, luku 21]. Ensimmäinen testauksen työntekijärooleista on **testiteknikko** (engl. *test technician*), joka asentaa ja konfiguroi testauksessa tarvittavat ohjelmat sekä mahdollisesti suorittaa joitakin yksinkertaisia testitapauksia. Yleisin rooleista on **ohjelmiston testaaja** (engl. *software test engineer*). Hän työskentelee uransa alkuvaiheessa samoissa tehtävissä kuin teknikko, kuitenkin kirjoittaen omat testitapauksensa. Kokemuksen kasvaessa ohjelmiston testaaja pääsee vaikuttamaan testisuunnitelmiin.

Testauksen urakehityksen seuraavana roolina Pattonin mukaan on **testiryhmän johtaja** (engl. *test leader*), joka on vastuussa suuresta osasta testausta tai pienen projektin tapauksessa koko testauksesta. Hänen tehtäviinsä kuuluu testaussuunnitelman teko ja muiden testaajien työn valvonta. **Testimanageri** (engl. *test manager*) tarkkailee koko projektia ja välittää tiedot testauksen vaiheista testiryhmän johtajalta projektin johdolle. Testimanagerin työhön kuuluu aikatauluista sopiminen, ensisijaisten tehtävien valinta, tavoitteiden suunnittelu ja vastuu testiresurssien saattamisesta projektin käyttöön.

3.5 Testauksen dokumentaatio ja raportointi

Kuten projektin muissakin vaiheissa, tulee testattaessakin suunnitelmat ja tulokset dokumentoida järkevästi, miettien dokumenttien tarve, laajuus, sisältö ja ajankohta. Testausdokumentaatioita tarvitsevat testaajat, projektin johto sekä muut työntekijät.

Raporttien kirjoittaminen ja tarkastaminen vie aikaa ja resursseja muulta toiminnalta. Kunnon dokumentaatio on kuitenkin välttämätön projektin onnistuneelle läpiviennille, uusille työntekijöille heidän tullessaan mukaan projektiin ja sovelluksen seuraavien versioiden kehittämiselle. Tämän takia dokumentaatiot pitää kirjoittaa jo alunperin kattaviksi vastamaan kysymyksiin, mitä testataan, miksi testataan ja miten testataan. Vaikka kattavan dokumentaation kirjoittamisessa voi kulua paljon aikaa, vaivannäkö maksetaan takaisin testauksen hallitulla, joustavalla ja nopeammalla suorittamisella.

Testauksessa tarvittavat dokumentit, kuten testaussuunnitelma, testitapaukset ja virheraportit, esitellään luvuissa 3.5.1-3.5.3. Dokumentaatioiden joukkoon voidaan isoissa projekteissa lisätä tarkat testausohjeet (engl. *test script*). Usein testauksen päätteeksi on tarpeellista kirjoittaa testausraportti, joka kuvaa testauksen todellisen kulun ja tulokset. Luku perustuu pääasiassa lähteeseen [Patton, luvut 16-18].

3.5.1 Testaussuunnitelma

Ensimmäinen testausvaiheen dokumentaatioista on testaussuunnitelma. Sen on tarkoitus ohjata testiryhmää sekä tiedottaa muille työntekijöille testauksen tarkoitus, tarvitut resurssit ja aikataulu. Patton huomauttaa, että testaussuunnitelman teko saattaa olla nopea tehtävä valmiin pohjan avulla, vain poimien tiedot aiempien projektien dokumentaatioista. Suunnitelman kirjoittamiseen kannattaa kuitenkin varata riittävästi aikaa, sillä päätarkoitus ei ole valmis dokumentti, vaan itse prosessi sisältäen yhteisen kommunikoinnin ja oppimisen. Dokumentissa mainitut asiat tulisi jokaisen ymmärtää ja hyväksyä sekä pystyä soveltamaan niitä käytännössä.

Testaussuunnitelmassa tulisi ensin kertoa korkeantason ominaisuudet eli suunnitelman tarkoitus, testattavan sovelluksen tiedot ja käytetty versio sekä vaatimukset yksityiskohtaisesti eriteltyinä. Patton painottaa, että testaussuunnitelmassa ei siis esimerkiksi riitä maininta kehitettävän sovelluksen tarpeesta olla nopea, vaan ominaisuus tulee tarkentaa testattavaan muotoon, esimerkiksi maininnaksi hakutoimintojen kuluttamasta ajasta.

Testauksen aikataulu olisi pyrittävä määräämään suunnitelmassa, ja tätä on käsitelty luvuissa 3.4.1 ja 3.4.2. Kullekin testikierrokselle on määriteltävä aloitus- ja lopetuskriteerit. Aloituskriteerinä voi olla se, että kehittäjät ovat saaneet kaiken tarvittavan valmiiksi sekä ohjelmat ja työkalut on asennettu. Kierroksen lopetuskriteerin määritelmä voi olla esimerkiksi kaikkien testitapausten läpikäynti havaitsematta enää yhtään virhettä.

Kaikki testauksessa tarvittavat resurssit, ihmisistä työkaluihin ja ohjelmistoihin, tulee dokumentoida. Pienenkin projektin testaussuunnitelmassa on hyvä kertoa, kuka projektissa on vastuussa mistäkin. Sovelluksen osa-alueet tulee jakaa testaajien kesken sekä määritellä ensi- ja toissijaiset tehtävät. Lisäksi vastuuosuudessa tulee kertoa muistakin testauksen suoritukseen liittyvistä osa-alueista, kuten kuka projektissa on vastuussa suunnitelmista, testaustyökaluista ja asiakkaan hyväksymistestauksesta.

Yhteisymmärryksen saavuttamiseksi olisi kaikki käytetyt termit määriteltävä. Testaussuunnitelmassa kerrotaan testaukseen liittyvästä strategiasta, käytetyistä menetelmistä ja mahdollisista testaustyökaluista. Lisäksi tulee päättää ja kirjata, testataanko kaikki itse vai annetaanko ainakin osa testauksesta ulkopuolisen testausyrityksen hoidettavaksi.

3.5.2 Testitapaukset

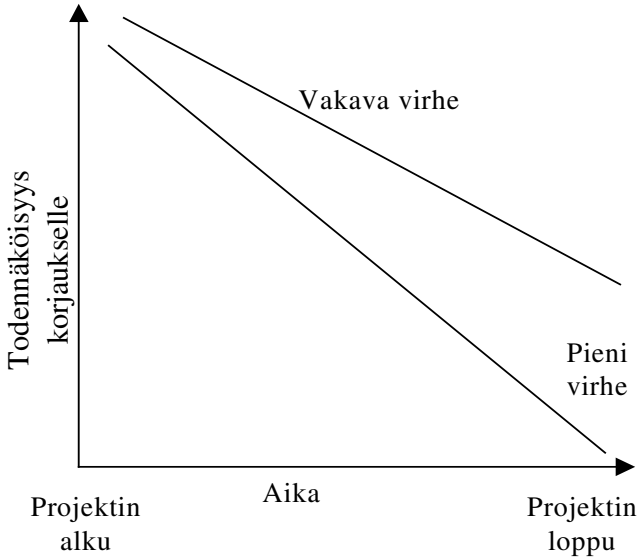
Testausprosessiin liittyy olennaisesti dokumentaatioon kirjattavat testitapaukset, jotka ovat Pattonin mukaan erityisen tärkeitä muun muassa seuraavan neljän syyn takia. Ensinnäkin projektissa saattaa olla **useita testaajia** laatimassa testitapauksia pitkän ajanjakson aikana, joten myös muiden testaajien sekä projektiin osallistuvien on tiedettävä testitapauksista ja pystyttävä käyttämään niitä. Toinen syy liittyy **toistettavuuteen** eli samat testitapaukset on pystyttävä suorittamaan uudelleen. Tällöin testauksen alussa on saatava voimaan täsmälleen samat arvot kuin aiemmalla suorituskerralla, jotta sovelluksen toiminnallisuus voidaan testata pyrittäessä hyväksymään korjattuja virheitä sekä uusien virheiden varalta.

Pattonin kolmas syy testitapausten tärkeydestä liittyy **jäljitettävyyteen**. Täten testitapaukset toimivat dokumentteina kerrottaessa, kuinka paljon testitapauksista suoritettiin, kuinka monta jätettiin suorittamatta, mitkä tapauksista läpäisivät testin ja mitkä eivät. Viimeisenä tärkeänä syynä huomioitakoon, että testitapauksia voidaan käyttää **todistamaan**, varsinkin kriittisissä sovelluksissa, tiettyjen osuuksien testauksen suorittaminen ja siten alustavasti niiden toimiminen.

Testitapausten tulee olla yksiselitteisiä ja kuvata testauksen suoritus yksityiskohtaisesti askel askeleelta. Ensin on kerrottava testauksen lähtötilanne, eli tarvitaanko erityisiä alkuvalmisteluja tai tarvitseeko tietyt testitapaukset suorittaa hyväksytysti ennen tämän aloittamista. Kaikki ohjelmalle annettavat syötteet on mainittava. Jos syötteitä on paljon, voi testauksen aikana tarvitut vaihtoehdot listata taulukkoon. Myös oletettujen tulosten tulee olla näkyvissä testitapauksissa. Yleisenä sääntönä pidetään, että kenen tahansa pitäisi ilman erityistä asiantuntemusta, vain testitapaus ohjeenaan, pystyä suorittamaan testaus.

3.5.3 Virheraportti

Dokumentaatio tarvitaan myös jokaisesta havaitusta virheestä. Raportti on syytä tehdä kunnolla, sillä huolimattomasti tehdystä raportista kehittäjä saattaa ymmärtää virheen väärin ja jättää sen huomiotta. Virheiden raportointi on tehtävä ajoissa, sillä mitä aiemmin virhe raportoidaan, sitä paremmat mahdollisuudet sillä on tulla korjatuksi. Kuva 6 havainnollistaa, että todennäköisyys varsinkin vähäisten virheiden korjaamiselle laskee lähes nollaan projektin lähetessä loppuaan. Virheraporttien elinkaarta kannattaa seurata varmistuakseen siitä, että kehittäjät korjaavat vakavat virheet.



Kuva 6. Virheen korjauksen todennäköisyys projektin kuluessa [Patton, luku 18].

Virheraportissa tulee kuvata virhe tarkasti, mutta kirjaten kuitenkin vain tarvittavat arvot ja askeleet virheen uudelleen tuottamiseksi. Välttämätöntä on raportoida, mitkä olivat muuttujien alkuarvot, suoritettavat toiminnot ja arvot virheeseen päädyttäessä. Mahdollisuuksien mukaan tulee etsiä ja raportoida syitä virheen esiintymiselle. Virheraporttiin tulee kirjata virheen vakavuus ja mahdollinen prioriteetti auttamaan kehittäjiä päättämään, kuinka nopeasti virhe tulisi korjata. Eräs esimerkki virheraportista on esitetty liitteessä 1.

3.6 Testityökalut

Testaus on suuritöinen ja kallis vaihe ohjelmistokehityksessä. Täten testityökalut ovat eräitä ensimmäisistä ohjelmistokehityksen avuksi kehitetyistä työkaluista. Patton määrittelee kirjassaan [Patton, luku 14], että testauksen työkalut on tarkoitettu helpottamaan, nopeuttamaan ja varmentamaan erityisesti rutiininomaisia työvaiheita sekä testaamaan suorituskykyä. Ei ole kuitenkaan olemassa työkalua, joka soveltuisi kaikkiin projekteihin. Mahdollisimman moneen tarkoitukseen sovellettavissa olevat työkalut ovat yleensä hakualgoritmeja käyttäviä ja varsin yksinkertaisia.

Työkalun valintaan vaikuttaa muun muassa se, millaista menetelmää testauksessa haluaa käyttää ja missä testauksen vaiheessa. Patton kirjoittaa kirjassaan [Patton, luku 14], että suurimmassa osassa testausta työkalut ovat hyödyllisiä. Tämä johtuu muun muassa siitä, että ne ovat nopeita ja toisin kuin ihmiset, voivat testata saman asian aina uudelleen ja silti yhtä tarkasti. Lisäksi niiden suorittaessa testausta ihmiselle jää aikaa suunnitella seuraavia testauksen vaiheita.

Ohjelmointikoodin virheenjäljitys eli debuggaus on hyvä esimerkki täysin työkalujen avulla suoritetusta toiminnasta. Tosin se ei kuulu osaksi testausta, vaan se suoritetaan jo ennen yksikkötestauksen aloittamista osana ohjelmointivaihetta. Patton muistuttaa kirjassaan [Patton, luku 7] debuggauksen eroavan testauksesta tavoitteensa vuoksi. Testauksen tavoitteena on löytää virhe, kun taas debuggauksen tavoitteena on paikallistaa tunnettu virhe korjausprosessin aikana. Dokumenttien oikeellisuuden testaamiseen ei ole olemassa erityisiä työtä automatisoivia työkaluja, mutta avuksi kelpaavat tekstinkäsittelyohjelmat ja muut yleiset projektin hallintaan, ajankäyttöön ja seurantaan liittyvät työkalut.

Varsinkin regressiotestausta (katso luku 4.3.5) saatetaan joutua toistamaan useasti, joten se olisi hyvä saada automatisoitua. Haikala ja Märijärvi esittelevät [Haikala et al., luku 15] avuksi testipetigeneraattorin, jolla generoidaan testipeti (engl. *test bed*) kuvaten ajettavan testin. Testipedille voidaan kuvata myös halutut testitulokset, jolloin tulosten tarkastelu voidaan automatisoida.

Järjestelmätestaukseen sopivat työkalut, joilla syötteet voidaan nauhoittaa ja käyttää seuraavalla testikerralla helposti uudelleen. Vertailuohjelmilla voidaan vertailla eri suorituskerroilla saatuja tuloksia keskenään. Tosin tulosten automatisoinnissa häiriötä aiheuttavat muuttuvat kellonajat ja muut jokaisella suorituskerralla vaihtelevat arvot.

Määriteltäessä testauksen lopetuskohtaa erilaiset testauksen kattavuusanalysaattorit ovat hyvä lisä. Ne mittaavat, kuinka monta kertaa tietty koodi on suoritettu tai montako prosenttia koodista on vielä jäänyt täydellisesti suorittamatta. Haikala ja Märijärvi kertovat kirjassaan [Haikala et al., luku 15] myös ohjelmista, joilla voidaan paikantaa suorituskykyongelmia sekä etsiä alustamattomia muuttujia tai vääriä osoittimia. Lisäksi työkaluja voidaan käyttää pienimuotoisten automaattisten raporttien tuottamiseen testauksen kulusta.

Ongelmana työkalujen käytössä on, että ne saattavat toimiessaan vaikuttaa ohjelman suorittamiseen hidastaen sen kulkua. Työkaluja käytettäessä onkin aina tunnettava niiden toiminnot ja rajoitteet, jotta tuloksia ei tulkita väärin. Kuitenkin yleisesti työkalut ovat hyvä lisä testauksessa, sillä ne lisäävät järjestelmällisyyttä, toistettavuutta ja verrattavuutta.

4 Testauksen tasot

V-mallin mukaisesti testaus jakautuu yksikkö-, integraatio-, systeemi- ja hyväksymistestaukseen. Sommerville painottaa kirjassaan [Sommerville, luku 20], että ihanteellisessa tilanteessa yksikköjen virheet löydettäisiin kehityksen varhaisessa vaiheessa ja rajapintavirheet integroinnin aikana, jolloin systeemi- ja hyväksymistestauksessa voitaisiin keskittyä vain korkeamman tason toiminnallisuuksiin. Näin ei käytännössä kuitenkaan ole, vaan yksikkötason virheitä paljastuu vielä systeemitestauksenkin aikana. Tämän takia V-mallin mukaisessa kehitysprosessissa testaus suoritetaan iteratiivisesti, jolloin uusien virheiden löydyttyä palataan takaisin aiempien tasojen testaukseen.

Työnjaon määrääminen testauksen eri tasoille kuuluu projektin johtoryhmän tehtäviin. Sommervillen mukaan useimmissa projekteissa ohjelmoijat testaavat oman koodinsa yksikkö- ja integraatiotasolla. Systeemitestauksen suorittaa erillinen ryhmä, jonka tehtäviin kuuluu kaikki systeemitestauksen moninaiset tehtävät ja koko sovelluksen integraatiotestaus. Hyväksymistestauksessa testaajina on oltava loppukäyttäjien edustajia. Kriittisimmissä sovelluksissa erillinen ryhmä saattaa suorittaa koko testauksen aina yksikkötasolta alkaen. Tällöin testaus sekä raportoidaan että suoritetaan varsin järjestelmällisesti. Luvun pääasialliset lähteet ovat [Sommerville, luku 20] ja [Pressman, luku 17].

4.1 Yksikkötestaus

Toteuttava järjestelmä koostuu useasta osasta eli yksiköstä. Testauksen ensimmäisessä vaiheessa eli yksikkötestauksessa (engl. *unit testing*) tarkastelun kohteena on kukin yksikkö vuorollaan itsenäisesti, ilman muita järjestelmän komponentteja. Testauksen tuloksia verrataan yksikkösuunnittelun ja arkkitehtuurisuunnittelun dokumentteihin, kuten tekniseen määrittelydokumenttiin. Yksikkötestauksen suorittaa yleensä sovelluksen kehittäjä itse sen jälkeen, kun virheenjäljitysohjelma ei enää havaitse virheitä sovelluksessa, vaan hyväksyy koodin syntaksiltaan virheettömänä.

4.1.1 Ohjelmayksikkö ja sen testaus

Yksikkötestauksessa testataan ohjelmiston pienimmät yksittäiset komponentit, yksiköt. Beizer kuvaa kirjassaan [Beizer, luku 4], että yleensä yksikön toteuttaa yksi kehittäjä ja se koostuu noin 100-1000 ohjelmarivistä. Sovellukset jaetaan yksiköiksi toiminnallisuuden mukaan, jolloin kunkin yksittäisen toiminnon suorittaa jokin tietty yksikkö. Määritelmän mukaan yksikön on minimissään otettava vastaan syöte, käsiteltävä sitä ja palautettava vaste. Yksikön on myös oltava tarpeellinen ja sillä on oltava rajapinta muihin yksiköihin.

Yksikkötestauksen suorittaa yleensä ohjelmoija itse. Tämä on taloudelliselta kannalta järkevää, sillä ohjelmoija tuntee yksikön toiminnan parhaiten sekä voi helpoiten valmistaa testiaineiston ja -tapaukset. Sommerville muistuttaa kirjassaan [Sommerville, luku 20], että kriittisissä sovelluksissa vaihe on syytä tehdä huolellisesti, jolloin jokaisen komponentin testauksessa tukeudutaan yksityiskohtaisiin määrittelyihin. Varsinaisia määrittelyjä ei useimmissa sovelluksissa kuitenkaan käytetä ja jo ajanpuutteen vuoksi testaus suoritetaan lähinnä omaan kokemukseen ja ajatteluun tukeutuen ilman erityistä suunnittelua.

Yksikkötestausta pidetään usein koodausta tukevana työvaiheena, joten näitä suoritetaan rinnakkain. Pressman määrittelee [Pressman, luku 17.3], että yksikkötestaus voidaan yleensä aloittaa, kun toiminnon vaatima lähdekoodi on toteutettu, se on tarkistettu päällisin puolin sekä sen syntaksi on esimerkiksi virheenjäljitysohjelman avulla tarkastettu ja korjattu. Yksikkötestaus kuuluu olennaisesti ohjelmistokehitysprosessiin, joten yksiköiden on välttämätöntä vastata määrittelyjään ennen kuin ne annetaan seuraavaan ohjelmistoprosessin vaiheeseen yhdistettäviksi toisiin yksiköihin.

4.1.2 Yksikkötestauksen toteutusperiaatteita

Yksikkötestauksessa on tarkoitus havaita virheet toteutuksesta, mutta tässä vaiheessa myös mahdollisiin vääriin määrityksiin on helppo puuttua ja korjata ne samalla. Gaon artikkelin [Gao] mukaisesti yksikkötestaus käsittää yksikön tietorakenteen, ohjelman logiikan ja ohjelman rakenteen tutkimisen yksikön kannalta. Vaiheessa testattavana ovat yksikön rajapinnat ja näkyvyys muualle sekä yksikön funktioiden oikeellisuus ja niiden suorittamat toiminnallisuudet.

Haikala ja Märijärvi huomauttavat kirjassaan [Haikala, et al., luku 15], että yksikkö ei yleensä toimi itsenäisenä ohjelmana. Testauksessa tarvitaan **testiympäristö**, joka koostuu pääohjelmasta ja riittävästä määrästä avustavaa koodia. Joskus joudutaan toteuttamaan **testipetejä** (engl. *test bed*), joilla yksikön toimivuutta kokeillaan. Testipetiin toteutetaan ympäristöä simuloivia osia eli **testiajureita** (engl. *test driver*) ja **tynkämoduuleita** (engl. *test stub*). Ajureita käytetään jäljittelemään muiden yksikköjen antamia syötteitä. Tynkiä käytetään silloin, kun kaikkia tarvittavia yksikköjä ei vielä ole koodattu, jolloin testattavasta yksiköstä voidaan antaa tyngille syötteitä ja saada niiltä takaisin vasteita.

Yksikkötestauksessa käytetään yleisesti lasilaatikkotestauksen menetelmiä eli testauksessa hyödynnetään koodia (katso luku 5.4). Pressman kirjoittaa kirjassaan [Pressman, luku 17.3] yksikkötestauksen yksinkertaistuvan, kun yksiköt suunnitellaan niin, että niillä on korkea koheesio eli mahdollisimman vähän tehtäviä. Parhaimmillaan vain yksi tehtävä suoritetaan yhdessä yksikössä. Tällöin testitapauksia ei tarvita kovin paljon ja virheiden havaitseminen ja paikallistaminen käy mahdollisimman helposti.

4.1.3 Yksikkötestauksen osa-alueet

Yksikkötestauksessa on välttämätöntä testata funktiot kaikilla mahdollisilla syötteillä. Testauksen pitää tuottaa kaikki mahdolliset vasteet. Pressman esittää kirjassaan [Pressman, luku 17.3] erityiset viisi yksikkötestauksen osa-aluetta. Testattavat osa-alueet ovat rajapinta, paikalliset tietorakenteet, raja-arvot, itsenäiset polut ja virheen käsittelypolut. Näitä Pressmanin määrittelemiä osa-alueita kuvataan tämän luvun seuraavissa kappaleissa.

Ensimmäisenä osa-alueena on **rajapintojen testaus** (engl. *interface testing*), jossa varmistetaan tiedon kulku yksikköön ja tiedon lähetys yksiköstä muualle sovellukseen. Testauksessa tarkistetaan, että syötteenä annetut ja saadut parametrit ovat virheettömiä, niiden määrä on oikea ja tyypit sopivat yhteen sekä järjestys on sama kaikkialla. Globaalien muuttujien on oltava oikeita sovelluksen läpi, eikä vakioita saa yrittää muuttaa. Erityisesti on huomioitava I/O-operaatiot, joiden virhetilanteet liittyvät esimerkiksi tiedostojen avaamiseen ennen käyttöä ja lukemiseen loppumerkin jälkeen. Testauksessa on varmistettava, että kaikki virhetilanteet käsitellään ja virheilmoitusten tekstit ovat oikein.

Paikallisten tietorakenteiden (engl. *local data structure*) testaamisella varmennetaan, että kaikki väliaikaisesti tallennettu tieto säilyy oikeana koko sovelluksen ajon ajan. Tietorakenteiden testitapausten on ensinnäkin käsiteltävä mahdolliset kirjoitusvirheet, muuttujien väärät alustukset ja oletusarvojen käyttö. Lisäksi on tarkastettava muuttujien nimien sopivuus ja oikea käyttö sekä muuttujien ali- ja ylikuormitustilanteista johtuvat poikkeukset.

Pressmanin mukaan **itsenäisten suorituspolkujen** (engl. *independent paths*) testaaminen on tarpeellista, jotta varmistutaan kaikkien mahdollisten toimintopolkujen kokeilusta ainakin kerran. Kontrollin kulkua ja erilaisia polku- ja silmukkatestauksia käsitellään dynaamisen lasilaatikkotestauksen yhteydessä luvussa 5.4.

Neljäntenä kohtana Pressmanin määritelmässä on testauksen suorittaminen **rajoilla** (engl. *boundary conditions*). Tässä testauksessa tukeudutaan ajatukseen, että rajoilla sovellus toimii kaikkein todennäköisimmin väärin. Jos sovellus toimii rajalla, voidaan päätellä sen todennäköisemmin toimivan myös rajojen sisäpuolella kaikilla sallituilla alueilla. Raja-arvojen hyödyntämistä testauksessa käsitellään tarkemmin osana mustalaatikkotestauksen menetelmiä luvussa 5.5.5.

Viidentenä osa-alueena on **virheiden käsittelyn polut** (engl. *error handling paths*). Tässä testauksessa on aikaansaatava kaikki mahdolliset virheilmoitukset ja tarkistettava niiden tarkoituksenmukaisuus, jotta epäkelpoja tekstejä ei pääse eteenpäin testauksen seuraaviin vaiheisiin. Virheet voidaan tuottaa virheen pakotuksella, joten kaikki virheilmoitukset saadaan testattua ilman sovelluksen ajamista virheellisiin tilanteisiin. Virhetekstien on oltava helposti ymmärrettäviä ja vastattava tapahtunutta virhettä. Virheilmoituksissa on oltava riittävästi tietoa, jotta ne auttavat syyn selvittämisessä. Virhetilanteet on myös pystyttävä käsittelemään oikein, eikä sovellus saisi joutua käyttäjälle vaikeaan tilanteeseen.

4.1.4 Hyvät ja huonot puolet

Huolellisella yksikkötestauksella on monia hyviä puolia. Kyseessä on vain pieni osa sovelluksesta, joten virheet löytyvät suhteellisen helposti ja testitapauksia ei tarvita paljon. Korjauskustannukset ovat pienimmät alhaisimmilla tasoilla, koska koodia on vähän, eikä korjauksella yleensä ole vaikutusta muille alueille. Testauksen suunnittelussa ja virheen jäljityksessä ei tarvitse välittää integraatio-ongelmista, koska yksiköitä ei ole yhdistetty. Pienen testausalueen etuna on myös, että voidaan käyttää raskaitakin testausmenetelmiä.

Huonona puolena yksikkötestauksessa on, että sitä varten on kehitettävä testausympäristö ja luotava riittävästi ajureita. Lisäksi yksikkötestauksella löydetään vain yksikkötason toiminnallisuuksien virheitä, kun taas monimutkaisia järjestelmään liittyviä virheitä ei voida havaita ennen kuin integraatio- ja systeemitestauksen aikana.

4.2 Integraatiotestaus

Integraatiotestauksessa (engl. *integration testing*) keskitytään erillisten komponenttien välisiin rajapintoihin sekä kommunikointiin ja toiminnallisuuteen niiden välillä. Vaihe voidaan aloittaa, kun integroitavien yksiköiden yksikkötestaus on suoritettu hyväksyttävästi ja yksiköistä on koottu osajärjestelmiä. Vaihe on tärkeä, vaikka yksikkötestaus olisi suoritettu onnistuneesti. Integraatiotestauksella havaitaan, jos tietoa katoaa rajapinnoilla, joillakin yksiköillä on epäsuotuisa vaikutus toisiin, arkkitehtuuri ei toimi kokonaisuutena tai jokin muu syy aiheuttaa ongelmia sovelluksen toiminnassa.

4.2.1 Toteutusperiaatteet

Integraatiotestaus on paljon hallitumpi ja kontrolloidumpi testauksen vaihe kuin yksikkötestaus. Useimmiten kehittäjät suorittavat integraatiotestauksen, mutta riippuen sovelluksesta vaiheen saattaa suorittaa testausryhmä. Sommerville luokittelee kirjassaan [Sommerville, luku 20.2] integraatiotestauksen tärkeimmäksi osa-alueeksi yksiköiden välisten rajapintojen toimivuuden tutkimisen. Testauksen toteutuksessa käytetään ennalta määrättyjä testitapauksia, jotka on kirjoitettu arkkitehtuurisuunnittelun ja järjestelmän määrittelyjen pohjalta. Vaiheen lopuksi kirjoitetaan raportti testauksesta ja sen tuloksista.

Integraatiotestaukseen vaikuttavat Paakin [Paakki, luku 7] mukaan sovelluksen **laadulliset tekijät**, joita ovat koheesio (engl. *cohesion*) ja kytkentä (engl. *coupling*). **Koheesio** kertoo yksikön voimasta eli sen tehtävien itsenäisyydestä. Koheesio on korkea, jos yksikkö toimii itsenäisesti suorittaen yksittäisen tehtävän ja kommunikoi vain rajallisesti muiden yksiköiden kanssa. **Kytkentä** taas mittaa eri moduulien välisiä kytkentöjä. Toisten yksiköiden toimintoja kutsuvat yksiköt ovat kytkettyjä. Integraatiotestaus on sitä helpompaa, mitä korkeampi on sovelluksen koheesio ja mitä matalampi on sen kytkentä.

Integraatiotestaus voidaan aloittaa periaatteessa heti, kun osa järjestelmän yksiköistä on koottu yhteen testattavaksi kokonaisuudeksi. Suurimpana vaikeutena tässä testauksessa Sommerville näkee löytyneiden virheiden syiden paikallistamisen, sillä kokonainen järjestelmä on monimutkainen sekä sisältää paljon rajapintoja ja monenlaista toiminnallisuutta. Jotta testaus olisi helpompaa, voidaan tiettyjä menetelmiä käyttäen järjestelmän integraatio testata aina jokaisen komponentin liittämisen jälkeen.

4.2.2 Komponenttien liittäminen yhteen

Sommerville kirjoittaa kirjassaan [Sommerville, luku 20.2], että ihanteellisessa tapauksessa testaus aloitetaan liittämällä toisiinsa ensin kaksi komponenttia ja testaamalla nämä. Kun testaus on hyväksyttävästi suoritettu, liitetään kolmas komponentti. Jos tämän jälkeen esiintyy virheitä, tiedetään virheiden melko todennäköisesti johtuvan kolmannen komponentin työskentelystä kahden ensimmäisen kanssa. Näin voidaan suorittaa koko järjestelmän integraatiotestaus ja saavuttaa virheiden paikallistamisessa mahdollisimman hyvä löytämisprosentti, koska viimeksi lisätty komponentti on aina todennäköisin virheiden aiheuttajana.

Käytännössä komponenttien liittäminen yksitellen kukin vuorollaan ei Sommervillen mukaan onnistu koko integraatiotestausvaiheen läpi, sillä järkevään toimintaan saatetaan tarvita samanaikaisesti monia komponentteja, ennen kuin niitä voidaan testata. Toisaalta lisättäessä kolmas komponentti saattaa kahden ensimmäisen komponentin keskinäinen toiminta muuttua, mikä voi johtaa uusien virheiden syntyyn. Kolmas huomioitava käytännön vaikeus on se, että virheiden korjaaminen monimutkaisessa integroidussa järjestelmässä voi aiheuttaa virheitä muissa, aiemmin hyvin toimineissa komponenteissa. Tämä johtuu siitä, että kokonaisuus on yleensä enemmän kuin osiensa summa.

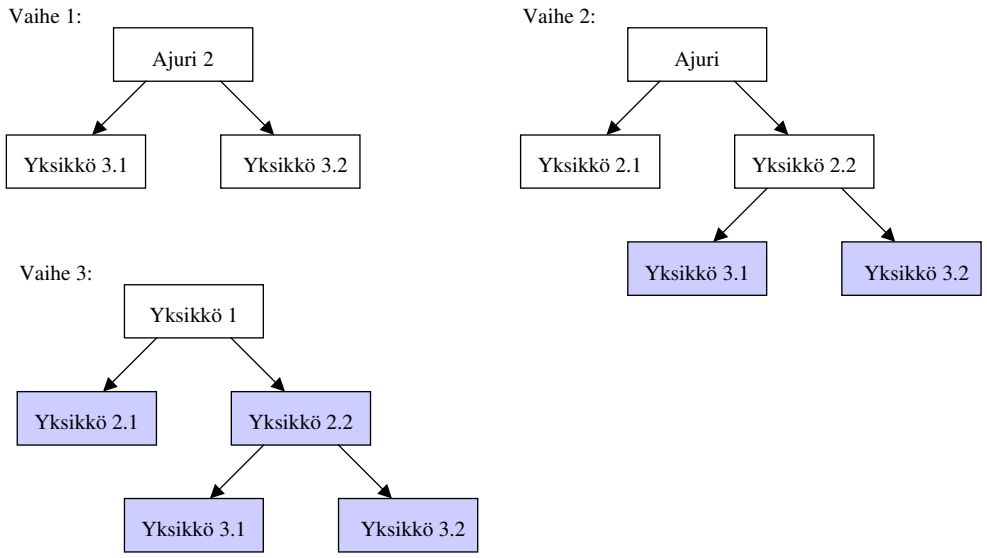
Beizer kirjoittaa kirjassaan [Beizer, luku 5.4], että integrointitestaus voidaan toteuttaa periaatteessa **kolmella eri tavalla**. Ensimmäinen tapa on käyttää niin kutsuttua ”**big-bang**” -mallia, jonka mukaisesti kaikki sovelluksen komponentit kootaan yhteen ja sen jälkeen aloitetaan testaus koko sovellukselle. Tätä ei pidetä hyvänä menetelmänä, koska virheiden selvittäminen on vaikeaa ja virheitä yleensä esiintyy paljon. Toinen tapa on **kokoava testaus**, jossa alimman testaamattoman tason komponentit yksi kerrallaan yhdistetään ja järjestelmän toimivuus testataan heti. Kolmas vaihtoehto on **jäsentävä testaus**, jossa testaus aloitetaan korkealta tasolta kokonaisen järjestelmän toiminnan testauksesta edeten vähitellen alemman tason yksityiskohtaisiin komponentteihin.

4.2.3 Kokoava testaus

Kokoava testaus tunnetaan yleisesti englanninkielisellä nimellä ”*bottom-up*” ja se on osana bottom-up -kehitysprosessia. Tässä kehitysprosessissa alemman tason komponentit kehitetään ja testataan ennen siirtymistä korkeamman tason kehitykseen. Kokoava testaus aloitetaan alemman tason pienistä komponenteista suunnaten testausta vähitellen ylemmille tasoille. Sommerville kirjoittaa kirjassaan [Sommerville, luku 20.2], että tällöin testaus voidaan aloittaa ajoissa jo ennen systeemin arkkitehtuurisuunnittelun lopullista valmistumista.

Testaus aloitetaan integroimalla alimman tason yksiköt yksitellen mukaan sovellukseen ja testaamalla nämä. Sen jälkeen integroidaan seuraavalla tasolla olevat yksiköt alimman tason kanssa. Tätä kiertoa jatketaan, kunnes kaikkein ylin taso on integroitu ja testattu. Paakki kirjoittaa luentomonisteessaan [Paakki, luku 7], että tämänkin testaamisen aikana tarvitaan ajureita korvaamaan ylempiä puuttuvia toiminnallisuuksia. Integraatiotestauksessa ajurit ovat suhteellisen yksinkertaisia koodata, koska niiden päätarkoitus on vain kutsua testattavana olevaa yksikköä, jotta sen suoritusta voidaan tarkkailla. Yleensä ajureiden tehtävänä on tarjota syötteet ja ottaa vastaan vasteet yksiköltä sekä tulostaa tulokset ruudulle testaajan tarkastettavaksi.

Kankaanpää esittää opinnäytteessään [Kankaanpää] kokoavan testauksen kulkua yksinkertaisessa vain kolmen tason komponentteja sisältävässä sovelluksessa kuvan 7 mukaisesti. Ensin testataan alimman tason yksiköt, joiden testaamisessa tarvitaan ajuria simuloimaan ylempää tasoa. Toisessa vaiheessa alimmat, jo testatut tasot, on merkityt tummalla pohjalla. Tässä vaiheessa testataan keskimmäisen tason yksiköt, missä yksiköiden apuna käytetään ajuria, joka jäljittelee ylimmän tason toiminnallisuutta. Kolmannessa vaiheessa viimeinenkin ajuri on korvattu oikealla yksiköllä ja ylimmän tason komponentti voidaan testata. Tämän jälkeen koko sovellus on valmis ja testattu.



Kuva 7. Kokoava testaus [Kankaanpää].

Hyvänä puolena kokoavassa testauksessa on se, että aloitettaessa pienistä komponenteista niiden yksittäiset virheet saadaan poistettua varhaisessa vaiheessa. Tällöin ne eivät Koikkalaisen mukaan [Koikkalainen, luku 12] ole vaikeuttamassa testaamista ylemmillä tasoilla päinvastoin kuin jäsentävässä testauksessa. Tosin, vaikka alemmat tasot on testattu tarkkaan, ei se takaa niiden toimivuutta koko järjestelmän laajuudessa.

Haittapuolena kokoavassa testauksessa on se, että sillä ei voida varmistaa järjestelmän oikeellisuutta yhtä hyvin kuin jäsentävällä testauksella. Tässä testaus suoritetaan keinotekoisessa ympäristössä ja jotkut olennaiset toiminnalliset tai perusarkkitehtuurin ongelmat eivät valitettavasti ehkä esiinny kuin vasta myöhään testauksen loppuvaiheessa. Erityisen huonona Paakki näkee [Paakki, luku 7] tässä mallissa sen, että korkean tason yksiköt tulevat testaukseen vasta viimeisinä. Joskus näiden virheellinen toiminta johtuu suunnitteluvirheistä, jotka olisi saatava korjattua mahdollisimman aikaisin.

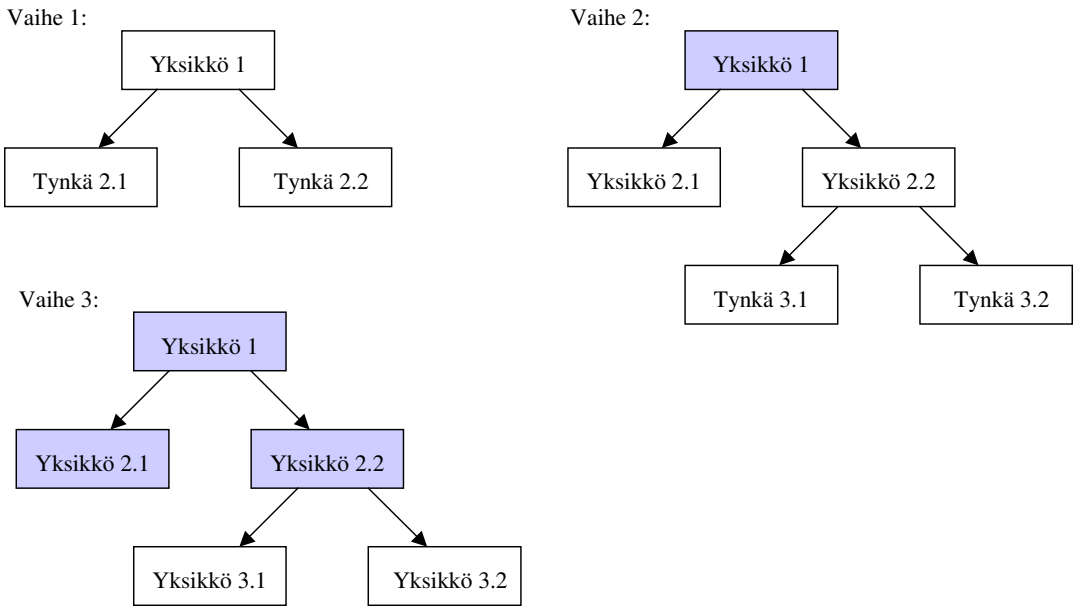
Joissakin sovelluksissa kokoava testaus saattaa olla integraatiotestauksessa ainoa vaihtoehto. Koikkalainen esittää luentomonisteessaan [Koikkalainen, luku 12], että näin on esimerkiksi sulautetuissa järjestelmissä. Tämä johtuu siitä, että kokonaisuus muodostuu osistaan, eikä kokonaisuuden testaaminen siten voi olla ensimmäinen vaihe. Tämä pätee myös järjestelmiin, joissa uudelleenkäytetään ja muokataan komponentteja muista järjestelmistä. Näissähän komponentit on jo kertaalleen toimiviksi todettuja. Tällöin ne teoriassa toimivat yksinään, jolloin ainoastaan niiden liittäminen yhteen sekä kommunikointi toisten komponenttien kanssa ovat mahdollisia virheitä tuottavia kohtia.

4.2.4 Jäsentävä testaus

Jäsentävä testaus on päinvastainen verrattuna kokoavaan. Jäsentävä testaus tunnetaan yleisesti englanninkielisellä nimellä ”*top-down*”. Sommerville kirjoittaa kirjassaan [Sommerville, luku 20.2] jäsentävän testauksen kuuluvan olennaisena osana top-down -kehitysprosessiin, jossa kehitys alkaa korkean tason komponenteista ja siirtyy vaiheittain alemmille tasoille. Korkean tason komponentin valmistuttua se testataan ja hyväksytyn testauksen jälkeen siirrytään kehittämään alemman tason komponentteja. Kehitysprosessi jatkuu näin alimmalle tasolle asti, kunnes kaikki komponentit on kehitetty ja testattu.

Jäsentävässä integraatiossa systeemin korkean tason komponentit yhdistetään ensin. Jäsentävässä testauksessa taas Koikkalaisen mukaan [Koikkalainen, luku 12] sovelluksen testaaminen aloitetaan korkeamman tason toiminnoista ja rakenteesta edeten vähitellen alemmalle tasolle yksityiskohtaisiin toimintoihin. Järjestelmää testataan aluksi kokonaisuutena ja virheitä löydettäessä korjataan ne. Kun uusia virheitä ei enää määriteltyjen ehtojen rajoissa löydetä, voidaan aloittaa alempien tasojen testataaminen. Näin käydään läpi koko järjestelmä aina yksikkötasolle asti, minkä jälkeen sovellus on täydellisesti testattu.

Integrointitestaus alkaa ylimmästä osasta, joka testataan kuten yksikkötestauksessa. Beizer kirjoittaa kirjassaan [Beizer, luku 5.4], että kaikkia kutsuttavia alaelementtejä (engl. *subelement*) jäljittelee tyngät (engl. *stub*). Nämä ovat yksinkertaistettuja ohjelmia, joilla tuotetaan samat vasteet ja toiminnallisuudet kuin oikeilla aliohjelmilla. Ohjelma esitetään yhtenä komponenttina, jossa tyngät hoitavat alakomponenttien tehtävät (katso kuva 8).



Kuva 8. Jäsentävä testaus [Kankaanpää].

Tyngät ovat vain rajapintoja muihin komponentteihin ja niiden toiminnallisuus on varsin vähäisistä. Tyngät mahdollistavat normaalin tiedon ja kontrollin kulun, aivan kuin sovellus olisi jo kokonainen ja valmis. Beizer kirjoittaa [Beizer, luku 5.4], että korkeimman tason toiminnallisuuden testauksen jälkeen sitä alemmalla tasolla olevat tyngät korvataan oikeilla toimivilla ohjelmilla ja testaus aloitetaan näille. Tätä kiertoa jatketaan jokaiselle elementille kerrallaan, kunnes koko systeemi on testattu.

Kuva 8 havainnollistaa Kankaanpään esityksessään [Kankaanpää] kuvaamaa jäsentävän testauksen vaiheitten kulkua varsin yksinkertaisessa kolmikerroksisessa sovelluksessa. Ensimmäisessä vaiheessa testataan ylin taso käyttäen apuna alemmissa tasoissa tynkiä. Vaiheessa 2 jo testattu ylin taso on merkitty tummalla pohjalla. Tässä yksikön 2.2 testaamisessa tarvitaan avuksi tynkiä. Kolmannessa vaiheessa ollaan jo sovelluksen alimmalla tasolla, jolloin kaikki tyngät on korvattu oikeilla yksiköillä. Tämän jälkeen koko sovellus on testattu.

Tyngät reagoivat yleensä vain ottamalla vastaan kaikki annetut syötteet ja antaen positiivisen, hyväksyvän vasteen. Aina sen on kuitenkin lähetettävä takaisin tarvittavat vasteet kutsujalle. Funktio tyngälle voi olla vaikkapa näin yksinkertainen:

```
int paras (int parametri) { return 0; }
```

Yleensä tilanne ei ole kuitenkaan näin yksinkertainen. Pressman luokittelee kirjassaan [Pressman, luku 17.4] tynkäfunktiot niiden vaikeustason mukaan. Yksinkertaisimmillaan tynkäfunktio ei tee muuta kuin esittää näytöllä ennalta koodatun viestin. Hieman vaikeampi taso tyngistä esittää näytöllä parametrina saadun arvon tai palauttaa halutun ennalta määrätyn arvon. Yleensä kaikkein monimutkaisimmillaan tynkäfunktio on, kun se ottaa vastaan parametrin ja hakee taulustaan sitä vastaavan arvon, jonka sitten palauttaa kutsuvalle ohjelmalle.

Luvun lopuksi tarkastellaan yhden elementtitason testauksen tapahtumia jäsentävässä testauksessa Beizerin kirjassaan [Beizer, s. 158] esittämän esimerkin mukaisesti. Ensin on oltava testattava osa valmiina ja sen on selvittävä hyväksytysti kääntäjästä ja käynnistyttävä. Seuraavaksi on korvattava yhteydet alaelementteihin tyngillä. Näiden alkuvalmistelujen jälkeen alkaa varsinainen jäsentävä testaus. Alaelementeistä testataan rajapinnat ja alaelementtien kutsun toiminta. Alaelementtejä jäljittelevistä tyngistä tulee korvata yksi kerrallaan oikeilla ohjelmilla, jotta saadaan selville mahdolliset virheiden aiheuttajat. Kun kaikki alaelementit on korvattu, voidaan tehdä elementin tasolla vielä rakenteellinen ja toiminnallinen testi asiaankuuluvasti. Tämän jälkeen voidaan siirtyä seuraavalle tasolle.

4.2.5 Voileipätestaus

Paakki esittelee luentomonisteessaan [Paakki, luku 7] voileipätestauksen (engl. *sandwich testing*), joka on yhdistelmä kokoavasta ja jäsentävästä testauksesta. Yleisesti projekteissa käytetäänkin molempia menetelmiä rinnan sopivissa määrin. Pienet komponentit voi olla hyvä testata keskenään kokoavalla testauksella, jolloin muu ohjelman kontrollin kulku ei ole liian monimutkainen.

Usein sovelluskehityksessä aikataulu eri komponenttien kehitykselle määrää, mitä osia järjestelmästä on mahdollista milloinkin testata. Tällöin sovellettu kokoavan ja jäsentävän testauksen yhdistelmä tuottanee taloudellisimman ja parhaimman tuloksen.

4.2.6 Kokoavan ja jäsentävän testauksen vertailu

Sommerville vertailee kirjassaan [Sommerville, luku 20.2] kokoavaa ja jäsentävää testausta neljän eri piirteen avulla. Ensinnäkin näitä voidaan vertailla **arkkitehtuurin validoinnin** avulla. Jäsentävässä testauksessa korkean tason suunnittelun ja arkkitehtuurin virheet löytyvät helpoiten jo kehitysprosessin varhaisessa vaiheessa, mikä auttaa pitämään kehityskustannukset kurissa. Sitä vastoin kokoavassa testauksessa arkkitehtuurivirheet huomataan vasta myöhäisessä vaiheessa, jolloin niiden vaikutukset ulottuvat kauas ja korjaaminen voi aiheuttaa suuria kustannuksia.

Paakki kertoo luentomonisteessaan [Paakki, luku 7] toisen jäsentävän testaamisen edun olevan se, että **systeemin toiminnan havainnollistaminen** on helpompaa. Jäsentävässä systeemissä rajoittunut, mutta alustavasti toimiva, systeemi on käytettävissä jo testauksen varhaisimmissa vaiheissa. Sommerville mainitsee tästä olevan systeemin kehittäjille ja testaajille hyötyä psykologiselta kannalta, sillä systeemin toimivuus on esillä ja validointi voidaan aloittaa heti systeemin ollessa käyttäjien saatavilla. Tämän lisäksi alustavasta rungosta on hyötyä loppukäyttäjille, jotka näkevät tämän prototyyppinä jo kehitysprosessin alussa. Tällöin siis käytettävyyden testaus mahdollistuu aikaisemmin. Toki kokonaisuuden havainnollistaminen voi onnistua myös kokoavassa testauksessa melko varhain niissä tapauksissa, joissa systeemi kehitetään uudelleenkäytettävistä komponenteista.

Jäsentävän testauksen haittapuolena on se, että on kehitettävä tyngät hoitamaan alemman tason tehtäviä. Pressman huomauttaa kirjassaan [Pressman, luku 17.4], että näiden tynkien on joko oltava varsin yksinkertaisia versioita oikeista alemman tason komponenteista tai testaajan itsensä on jäljiteltävä näiden käyttäytymistä antaen oikeita syötteitä ohjelmalle. Kokoavassa testauksessa avuksi on kirjoitettava testiajureita ajamaan ja ohjaamaan testauksessa olevia alemman tason komponentteja. Testiajurit simuloivat ympäristöä ja kutsuvat testauksen piirissä olevia komponentteja. Kokoavan testauksen etuna on, että yleisesti ajureiden kirjoittaminen on helpompaa kuin tynkien.

Testauksen edistymisen havainnointi voi olla ongelmallista molemmissa lähestymistavoissa. Sommerville huomauttaa, että monissa järjestelmissä korkean tason komponentit eivät anna ajettaessa minkäänlaista vastetta tai tulosta näytölle mitään. Kuitenkin testauksessa vasteet ovat tarpeellista, joten testaajan on kehitettävä keinotekoisia ympäristöjä saadakseen selville testauksen tulokset. Paakki listaa luentomonisteessaan [Paakki, luku 7] hyvänä puolena kokoavassa testauksessa olevan sen, että se soveltuu paremmin ryhmätyönä toteutettavaksi. Eri alemman tason komponentteja voi nimittäin olla samanaikaisesti testaamassa useampi henkilö, jolloin edistyminen voi olla nopeampaa.

4.3 Systeemitestaus

Systeemitestauksessa (engl. *system testing*) testataan koko järjestelmää sen käyttötarkoitusta vastaavassa ympäristössä. Tarkoituksena ei ole testata yksittäisiä funktioita, vaan pääpaino on mieluummin osoittaa poikkeavuuksia tuotteen sekä sen vaatimusten ja dokumentaation välillä testauksessa. Saatuja tuloksia verrataan määrittelydokumenttiin. Testaajiksi tulisi valita henkilöitä, jotka eivät ole osallistuneet kehitystyöhön. Systeemitestaukseen kuuluu myös systeemin teknisten ominaisuuksien (kuten käytettävyyden) testaaminen.

4.3.1 Toteutusperiaatteet ja osa-alueet

Systeemitestauksella pyritään selvittämään sovelluksen sekä siihen liittyvien osa-alueiden toiminta ja keskinäinen kommunikointi erilaisissa käytön aikana esiintyvissä tilanteissa. Systeemitestauksessa valmis ohjelmisto testataan sille määriteltyjen vaatimusten suhteen.

Systeemitestaus suoritetaan mahdollisimman aidossa ympäristössä tai erityisesti testaamista varten pystytetyssä testiympäristössä. Etuna testiympäristön käytössä on mahdollisuus tehdä erilaisia suorituskyky- ja kuormitustestejä häiritsemättä kehittäjien tai jo systeemiä käyttävien henkilöiden toimintaa. Haittapuolena on mahdollinen ympäristön luonnista ja ylläpidosta koituva lisätyö.

Testaajiksi ei saa valita ohjelmoijia, vaan testaajina on oltava henkilöitä, joilla ei ole ollut tekemistä kehitystyön kanssa. Tällöin testaajat eivät pyri tekemään testejä koodia myötäileviksi. Lisäksi heillä on suurempi kiinnostus havaita virheitä kuin itse koodin kirjoittajalla, joka mieluiten osoittaa koodinsa toimivaksi.

Systeemitestauksessa pyritään paljastamaan vikoja, jotka eivät ole tulleet esille V-mallin (katso luku 3.3) mukaisessa alemman tason ohjelmistotestauksessa. Tämä johtuu siitä, että järjestelmä on enemmän kuin osiensa summa. Lisäksi joitakin alemman tason virheitä ei voida havaita ennen kuin testattavana on koko systeemi. Haikala ja Märijärvi huomauttavat kirjassaan [Haikala et al., luku 15], että systeemitestausvaiheessa havaitut virheet ovat yleensä kriittisempiä kuin sovelluksen alemman tason testausvaiheissa havaitut, koska käsiteltävänä on koko järjestelmä. Kuten testauksen muissakin vaiheissa, myös systeemitestauksessa havaittu virhe korjataan, mutta tässä korjaus helposti heijastuu sovelluksen muihin osiin ja saattaa tuottaa uusia virheitä.

Paakki listaa luentomonisteessaan [Paakki, luku 2] systeemitestauksessa eri tarkoituksiin käytettäviä **toteutustapoja**. Näihin luetaan mukaan systeemin teknisten ominaisuuksien testaus, kuten kuormitus-, asennus- ja käytettävyystestit. Lisäksi voidaan testata systeemin turvallisuutta, luotettavuutta ja toipumiskykyä. Tarpeellinen on myös regressio- eli uudelleentestaus, koska systeemitestauksessa korjatut toiminnallisuudet voivat aiheuttaa virheitä muualla. Testaus tulee ulottaa koko järjestelmään, sillä osa asiakkaan vaatimuksista ei ole mukana määrittelyissä, vaan niitä pidetään itsestään selvyyksinä.

Luvuissa 4.3.2-4.3.7 esitellään erityiset systeemitestauksen osa-alueet, joita ovat muun muassa käytettävyys-, toipumis-, kokoonpano- ja regressiotestaus. Muut systeemitestauksessa käytetyt alueet liittyvät erityisesti mustalaatikkotestaukseen. Tätä käsitellään testausmenetelmien osana viidennessä luvussa, joten rasitus- ja käyttöliittymätestit esitetään luvussa 5.6.

4.3.2 Käytettävyystestaus

Käytettävyystestauksen (engl. *usability testing*) tarkoituksena on varmentua sovelluksen sopivuudesta, hyödyllisyydestä, helppokäyttöisyydestä ja toimivuudesta järjestelmän tuleville käyttäjäryhmille. Käytettävyystestaus on suurimmaksi osaksi käyttöliittymätestausta.

Yleensä käytettävyystestausta varten ei kirjoiteta erillistä suunnitelmaa, vaan tämä vaihe sisältyy olennaisesti jo systeemitestaukseen. Jos erillisiä testitapauksia tähän tarvitaan, ne voidaan liittää systeemitestausdokumentaatioon. Käytettävyystestaus liittyy systeemitestaukseen, mutta se voidaan Haikalan ja Märijärven mukaan osittain suorittaa myös prototyypin avulla jo määrittelyvaiheessa.

Yleisin tapa käyttäjätestaukseen on järjestää otokselle tulevista käyttäjistä käytettävyystesti, jossa uutta järjestelmää kokeillaan. Tosin jos kyseessä on olemassaolevan järjestelmän korvaaminen uudella, sen käyttäjillä on tietyt tavat käyttää vanhaa järjestelmää, mikä on haitaksi testaamisessa. Vanhojen käyttäjien keskuudessa saattaa myös esiintyä muutosvastarintaa uuteen siirtymistä kohtaan. Tällöin saattaa olla parempi käyttää ainakin osittain ihmisiä, jotka eivät ole käyttäneet aiempaa järjestelmää.

Haikala ja Märijärvi määrittelevät [Haikala et al., luku 15] **käytettävyystestauksen** alkavan sillä, että siihen osallistuville henkilöille annetaan jokin tyypillinen testitehtävä. Heitä pyydetään suorittamaan se ja ajattelemaan samanaikaisesti ääneen kaikkia suorittamiaan vaiheita ja toimintoja. Valitut tarkkailijat tekevät muistiinpanoja koko testauksen ajan ja lopuksi testaus analysoidaan. Käytettävyystestauksen voi suorittaa myös joukko siihen erikoistuneita ihmisiä. Tällöin hyödynnetään käytettävyyslaboratorioita, joissa on kunnon laitteisto käyttäjän toiminnan rekisteröintiin. Tavanomainen väline on videokamera, koska kuvattua nauhoitusta voidaan tarkastella vielä myöhemmin uudelleen.

Olennainen tarkkailtava asia käytettävyystestauksessa on se, onko sovelluksessa **helppo navigoida** eli pystyykö käyttäjä helposti syöttämään tietoa, liikkumaan ja löytämään paikasta toiseen sekä poistumaan ohjelmasta. Toisaalta on tarkasteltava, onko sovellus **helppokäyttöinen** eli voiko käyttäjä tehdä haluamansa toiminnot itselleen sopivimmalla tavalla ja onko suoritustapa selkeä. Lisäksi sovelluksesta testataan **tehokkuutta** eli käyttäjän on voitava suorittaa haluamansa toiminnot lyhyessä ajassa ja vähin askelin. Sovelluksen rakenteen, avustustiedostojen ja dokumentaation on tärkeää olla **helposti käsitettäviä**.

Kankaanpää kuvaa opinnäytteessään [Kankaanpää] mahdollisuutta toteuttaa käytettävyystestausta myös kenttätestauksena. Tällöin testaaja asettuu loppukäyttäjän rooliin ja testaa sovellusta sen todellisessa käyttöympäristössä. Kenttätestauksen tarkoituksena on varmistaa sovelluksen toimivuus sille asetettujen vaatimusten mukaisesti.

4.3.3 Toipumistestaus

Systeemitestaukseen kuuluu toipumistestaus (engl. *recovery testing*). Tätä tarvitaan, koska tietokonesovellusten tulee selviytyä virhetilanteista ja jatkaa suoritusta ennalta määritellyn ajan kuluessa. Luku perustuu pääosin lähteeseen [Pressman, luku 17].

Joissain tapauksissa sovelluksen pitää olla virhesietoinen. Tällöin mahdollisten virheiden syntyminen ei saa vaikuttaa koko sovelluksen toimintaan, vaan vian on pysyttävä paikallisena häiriönä. Tietyissä sovelluksissa toipuminen virhetilanteista ei ole automaattista, jolloin vaatimuksena on mahdollisuus ja kyky korjata virhetilanteet nopeasti, jotta taloudellista vahinkoa syntyisi mahdollisimman vähän.

Toipumistestauksessa järjestetään tilanteita, joissa sovellus saadaan kaatumaan usein ja monissa erilaisissa tilanteissa, kaikkien mahdollisten tapahtumien seurauksena. Sovelluksen kaaduttua tarkkaillaan sen toipumiskykyä. Tässä siis testataan järjestelmän toipumista normaalitilaan poikkeustilanteista. Näitä Pressmanin mukaan ovat esimerkiksi virtakatkos, muistihäiriö, levyvika, linjavika, aineistovirhe tai prosessin keskeytyminen.

Jos sovellus pystyy itse toipumaan automaattisesti, tarkkaillaan tilanteita alustuksesta, tietojen takaisin saannista ja uudelleenkäynnistyksestä. Jos taas ihmisen on itse palautettava sovellus kunnolliseen toimivaan tilaan, on arvioitava tehtävään kuluvaa aikaa, jonka tulee olla hyväksyttävissä rajoissa.

4.3.4 Kokoonpanotestaus

Systeemitestauksen yhtenä osana on kokoonpanotestaus (engl. *configuration test*), jolla pyritään testaamaan kehitetyn sovelluksen kykyä kommunikoida muiden tietokoneen sisäisten ja ulkoisten laitteiden kanssa. Tarvitut testit voivat keskittyä muutamaan asiakkaan määrittelemään kokoonpanoon. Pahimmassa tapauksessa lopullista toimintaympäristöä ei voida määritellä, vaan sovelluksen olisi periaatteessa toimittava kaikissa mahdollisissa kokoonpanoissa. Kokoonpanotestausta voidaan hyödyntää esimerkiksi WWW-sovelluksen testauksessa, jolloin testaus tulee suorittaa tarvittavilla eri selaimilla ja niiden versioilla eri käyttöjärjestelmissä. Luku perustuu pääasiallisesti lähteeseen [Patton, luku 8].

Mikäli sovellusta tullaan käyttämään erilaisissa laitteistokokoonpanoissa, sovelluksen toimivuutta kaikissa kyseisissä kokoonpanoissa täytyy testata. Testausta suunniteltaessa on mietittävä, mitä ulkoisia laitteita sovelluksen kanssa tullaan käyttämään, mitä malleja ja versioita laitteista on saatavilla sekä mitä sovelluksessa hyödynnettäviä ominaisuuksia laitteistot tukevat. Valitettavasti mahdollisia kokoonpanoja voi olla tuhansittain, joten kaikkia yhdistelmiä ei voida testata.

Ensinnäkin sovellus on testattava useammassa tietokoneessa, käyttäen erilaisia ulkoisia laitteita ja ajureita, muuttaen muistin määrää ja kokeillen eri rajapintoja. Patton kehottaa testausta suunniteltaessa miettimään, mitkä laitteet olennaisimmin liittyvät testattavaan sovellukseen. Esimerkiksi graafinen tietokonepeli vaatii runsaasti testausta kuvan ja äänen esittämisen alueilla, mutta tulostimien käytön testaus ei ole mielenkiinnon kohteena.

Välillä on vaikeaa sanoa, johtuuko virhe kokoonpanosta vai onko se vain tavallinen virhe. Tällöin kannattaa kokeilla sovelluksen toimintaa toisessa täysin erilaisessa tietokonekokoonpanossa, toistaen kaikki testauksen askeleet täysin samoina. Jos virhettä ei esiinny toisessa, se todennäköisesti on merkki kokoonpano-ongelmasta. Jos virhe taas esiintyy molemmissa, se lienee vain tavallinen koodausvirhe.

Kun kokoonpanovirhe havaitaan, seuraa kysymys, kenen se kuuluu korjata. Kuuluuko korjaaminen testattavana olevan sovelluksen kehittäjälle vai laitteiston valmistajalle? Patton huomauttaa, että jos virhe esiintyy vain tiettyjä tietokoneita käytettäessä, se luultavasti on sovelluksen vika, eikä kokoonpanovirhe. Jos se taas esiintyy esimerkiksi yritettäessä tulostaa tietylle tulostimelle miltä tahansa tietokoneelta, vika lienee laitteistossa. Tosin jos virheen aiheuttajana on markkinoiden suosituin tulostin, sen korjaaminen ei yleensä tule kysymykseen, vaan sovelluksen toimintaa on muutettava niin, että se pystyy toimimaan virheellisen tulostimen kanssa.

Erilaisia mahdollisia käyttäjien käyttämiä kokoonpanoja voi olla satoja, tuhansia tai jopa enemmän, joten kaikkia ei voida testata. On selvitettävä tapa vähentää testattavien määrää vain olennaisimpiin tapauksiin, vaikka tietysti aina on olemassa riski jätettäessä osa tapauksista testaamatta. Ratkaisuna kokoonpanojen vähentämiseen on ekvivalenssiluokkiin jako, jota käsitellään luvussa 5.5.4.

Kokoonpanoja vähennettäessä on huomioitava, mitä ominaisuuksia tietyistä laitteista on tuettava. Esimerkiksi optioista voi määritellä tulostimen testaamiseksi vain ne ominaisuudet, joita sovelluksessa tarvitaan esimerkiksi asettamalla minimivaatimuksia tulostimen värimäärälle ja resoluutiolle. Laitteistot ja sovellukset vanhenevat nopeasti, joten yleensä kovin vanhoja kokoonpanoja ei käytännössä kannata huomioida.

Patton toteaa, että mitään yleistä ohjetta testitapausten määrän vähentämiselle ei ole, vaan päätös on aina projektikohtainen. Aina olisi keskusteltava testaus- ja kehittäjäryhmän sekä projektin johtajien kanssa, mitkä laitteistot otetaan testaukseen ja millaisena testaus koetaan riittäväksi.

4.3.5 Regressiotestaus

Regressiotestauksessa (engl. *regression testing*) ohjelmisto testataan uudelleen sen jälkeen, kun siitä löytyneitä virheitä on korjattu tai toiminnallisuutta on lisätty tai muutettu. Tällä testauksella pyritään varmistamaan sovelluksen toiminta korjausten jälkeenkin alkuperäisten vaatimusten mukaisesti. Testaus tarvitaan myös takaamaan, että uusia virheitä ei ole ilmennyt aiemmin moitteettomasti toimiviin toimintoihin. Regressiotestausta suoritetaan sekä ohjelmiston kehitysvaiheessa että erityisesti ohjelmiston ylläpitovaiheessa.

Testauksessa käytetään testitapauksista osajoukkoa, joka kattaa sovelluksen päätoiminnallisuudet. Testauksen lopuksi kuuluu tehdä vielä täydellinen kaikki testitapaukset kattava uudelleentestaus. Paakki huomauttaa luentomonisteessaan [Paakki, luku 8], että regressiotestausta varten tarvitaan osittain uudet testitapaukset, koska kaikkia vanhoja testejä ei kannata ajaa uudelleen. Koska testitapaukset on suunniteltu järjestelmään ennen muutoksia, vanhat testit liittyvät osittain myös muuttamattomiin ominaisuuksiin ja uudet toiminnallisuudet eivät vielä esiinny niissä.

Pressman jakaa kirjassaan [Pressman, luku 17.4] regressiotestaukseen sopivat **testitapaukset kolmeen luokkaan**. Ensinnäkin tarvitaan joukko testejä varmistamaan sovelluksen kokonaisvaltainen toimiminen. Täydentävillä testitapauksilla pyritään testaamaan osa-alueet, joihin muutoksesta todennäköisimmin on aiheutunut haittaa. Kolmas luokka testitapauksia keskittyy muutettuihin komponentteihin.

Regressiotestauksessa tarvittavien testitapausten määrä voi kasvaa suureksi, joten Pressman suosittelee työkalujen käyttöä. Työkaluja kannattaa käyttää ainakin nauhoittamaan testitapaukset ja vasteet seuraavia testikertoja ja testauksen tasoja varten sekä erityisesti tulosten vertailuun. Paakin mukaan regressiotestauksen kustannukset voivat olla jopa 60% testauksen kustannuksista. Tosin keskimääräisesti luku on 20-30 prosentin välillä.

Regressiotestauksessa voidaan pyrkiä paikallistamaan virheiden syitä vanhempien jo toimivien versioiden avulla. Kun sovellusta on muutettu useassa toiminnossa ja virheitä ilmenee, neuvoo Paakki [Paakki, luku 8] osittamaan testauksen. Ensiksi lisätään sovelluksen viimeisimpään toimivaan versioon yksittäinen muutos (A) ja kokeillaan toimintaa. Jos virheitä esiintyy, tiedetään syyn olevan yksikön A toiminnassa. Jos virheitä ei esiinny, poistetaan muutos A ja liitetään alkuperäiseen toimivaan versioon muutos B. Näin voidaan käydä läpi kaikki muutokset ja saada selville ne, joiden kanssa sovellus ei toimi. Tosin näin yksinkertaista ei testaus aina ole, sillä virheitä saattaa ilmetä vasta muutosten C ja D yhteisestä toiminnasta. Lisäksi joskus muutosten osittaminen itsenäisiin osiin ei ole mahdollista ja joitakin muutoksista ei voi testata yksinään.

4.3.6 Suorituskyky, luotettavuus ja joustavuus

Suorituskykytestaus (engl. *performance testing*) suoritetaan usein systeemitestauksen osana, jolloin koko integroitu järjestelmä on käytössä. Toki joiltakin osin testausta suoritetaan myös testauksen aiemmissa vaiheissa lasilaatikkotestauksen aikana. Melzer määrittelee artikkelissaan [Melzer], että tarkoituksena on osoittaa, että sovellus ei selviä tehtävistään oletetussa ajassa. Tämä ei koske ainoastaan hyväksyttävää vasteaikaa, vaan myös taustalla olevien prosessien toimintaa.

Hiukan suorituskykytestauksen mukainen menetelmä on myös **luotettavuustestaus** (engl. *reliability test*), jonka aikana pyritään tarkistamaan järjestelmälle asetettujen luotettavuusvaatimuksien pitävyys. Suorituskykytestauksella pyritään selvittämään sovelluksen toiminta ympäristössään, kun taas luotettavuustestauksella pyritään tarkistamaan sovelluksen toiminta ympäristössään pitkällä aikavälillä.

Beizer määrittelee kirjassaan [Beizer, luku 9.4] erääksi luotettavuusvaatimukseksi ajan, jonka yksikkö toimii virheettömästi (engl. *MTBF, mean time between failures*). Yksiköksi voidaan tässä lukea yksittäinen komponentti tai koko sovellus. Tämän ajan määrääjänä toimii yleisesti asiakkaan vaatimukset. Toinen luotettavuusvaatimus on asennuksen jälkeen ilmenevien virheiden lukumäärä.

Luotettavuustestauksessa arvioitavana olevia ominaisuuksia ei voida aina testata, vaan täytyy käyttää matemaattisia malleja. The BCS SIGIST Standards Working Partyn julkaisussa [BCS] muistutetaan, että on tärkeää kuitenkin kuvata kaikki halutut luotettavuusvaatimukset käyttäjän ymmärtämällä laskettavissa olevalla tavalla. Luotettavuustestaus suoritetaan yleensä kustannusten kannalta tehokkaasti lyhyessä ajassa oloissa, jotka ovat tavanomaista käyttöä ankarammat. Sovellukselle annetaan runsas määrä tavanomaisia syötteitä ja luotettavuus lasketaan siitä, kuinka monta väärää vastetta saadaan. Näiden syötteiden on vastattava todennäköisiä syötteitä, jotta saadaan luotettavampi arvio.

Koikkalainen määrittelee yhdeksi testaukseksi luentomonisteessaan [Koikkalainen, luku 12] **säietestauksen**. Se tunnetaan myös tapahtumatestauksena ja se sopii erityisesti sulautettujen ja reaaliaikajärjestelmien testaamiseen. Testejä tehdään generoimalla palvelupyyntöjä tai keskeytyksiä.

Eräs välttämätön ominaisuus sovelluksilla on **joustavuus** (engl. *resilience*). Tämä on oleellista, jotta sovellusta tai sen osia voidaan helposti käyttää uudelleen muualla. Toinen joustavuudella tavoiteltu piirre on se, ettei sovellus vanhene heti ympäristöjen muututtua, vaan sitä voidaan ainakin pienten muutosten jälkeen käyttää uudelleen. Joskus testaus suoritetaan vain vähäisillä tiedoilla ja muutamilla samanaikaisilla käyttäjillä, vaikka ohjelman on toimittava oikeasti tuhannen käyttäjän sovelluksena. Joustavuustestauksen tarkoituksena onkin pyrkiä luomaan oikeat olot ja testaamaan sovellusta niissä.

4.3.7 Tietoturvatestaus

Tietoturva (engl. *security*) on merkittävä ominaisuus monissa järjestelmissä, koska niissä liikkuu luottamuksellista tietoa. Beizer esittelee kirjassaan [Beizer, luku 7.4], että tietoturvatestauksen tarkoituksena on selvittää, kuinka hyvin sovellus suojautuu epäkelpoja sisäisiä tai ulkoisia sisäänpääsypyrkimyksiä sekä tarkoituksellista vahingontekoa vastaan. Turvallisuustestauksella pyritään kattamaan koko sovellus suunnittelusta ja matalan tason tehtävistä aina tärkeimpiin toimintoihin asti. Tähän testaukseen ei varsinaisesti tarvita uusia testitapauksia, vaan muun testauksen aikana tutkitaan, ettei tietoturvasta tingitä.

Tietoturvatestien aikana testaaja jäljittelee ihmistä, joka haluaa tunkeutua sovellukseen. Testaaja pyrkii hankkimaan tietoonsa salasanoja, rikkomaan sovelluksen tietosuojan tarkoitukseen sopivilla työkaluilla tai varaamaan systeemin kaiken kapasiteetin estäen käytön muilta. Hän voi myös tahallisesti aiheuttaa systeemiin virheitä, koska toipumisaikana sovellus saattaa olla helpoiten haavoitettavissa.

Pressman kirjoittaa kirjassaan [Pressman, sivu 524], että riittävillä resursseilla sovelluksen tietoturva saadaan yleensä rikottua. Sovelluksen kehittäjien tehtävänä onkin tuottaa niin hyvä sovellus, että siihen tunkeutumiseen tarvitut resurssit ovat suuremmat kuin tunkeutumalla sovelluksesta saatava hyödynnettävä tieto.

4.4 Hyväksymistestaus

Hyväksymistestauksessa (engl. *acceptance testing*) ohjelmiston oikea toiminta varmistetaan yhdessä asiakkaan kanssa. Testauksessa on käytettävänä tuotteen alustava versio, joka ei ehkä sisällä kaikkia tuotteen lopulliseen versioon suunniteltuja ominaisuuksia. Paakki kirjoittaa luentomonisteessaan [Paakki, luku 2], että hyväksymistestauksessa ei tuotetta välttämättä verrata mihinkään määrittelyyn, vaan asiakas testaa omalla tavallaan sovelluksen käytettävyyttä ja sopivuutta itselleen.

Yleensä sovellus käy läpi kaksi vaihetta, ennen kuin se todetaan valmiiksi ja hyväksytään lopullisena. Ensimmäistä vaihetta kutsutaan alfa-testaukseksi, jonka suorittavat yleensä tulevat käyttäjät kehittäjien johdolla kehittäjäyrityksen tiloissa. Toista vaihetta kutsutaan beta-testaukseksi. Se sisältää yleensä rajallisen määrän ihmisiä myös kehittäjänä olevan yrityksen ulkopuolelta ja se suoritetaan asiakkaan tiloissa.

4.4.1 Tarkoitus

Hyväksymistestauksen tavoitteena on selvittää järjestelmän soveltuvuutta asiakkaalle ja käyttäjille. Järjestelmän tulisi olla muun muassa helppokäyttöinen, vastata käyttäjien tarpeisiin sekä olla hyödyllinen ja toimia käyttäjäryhmiensä mielestä oikealla tavalla.

On välttämätöntä, että vaiheen suorittavat oikeat käyttäjät, sillä muiden vaiheiden testaajilla on varsin erilainen näkökulma testaukseen. Kehittäjillä ja asiakkaalla ei yleensä ole käytännön kokemusta aihealueesta, eikä loppukäyttäjien yleisestä toimintatavasta. Sovelluksen tulevat käyttäjät ehkä väärinymmärtävät käyttöohjeita, käyttävät vääränlaisia syötteitä, eivätkä ymmärrä testaajien mielestä varsin selkeää vastetta. Tällaiset käyttäjien toiminnat paljastavat sovelluksen huonon käytettävyyden ja dokumentaatioiden puutteellisuuden. Pressmanin [Pressman, luku 17] mukaan hyväksymistestaus saatetaan suorittaa asiakkaan mielipiteestä riippuen nopeana testiajona epämuodollisesti tai pitkällä aikavälillä tarkkoihin testitapauksiin nojautuen.

Käyttäjätestaus eroaa paljon aiemmista testaustasoista ja itse asiassa se on sekoitus muita testejä. Yleensä käyttäjä saa koko sovelluksen ja siihen liittyvät dokumentaatiot testattavakseen. Sovelluksessa olisi oltava käytössä todellinen aineisto, eikä enää aiempia testejä varten luotua alustavaa testiaineistoa tulisi käyttää. Päätarkoitus hyväksymistestauksessa on todistaa, että sovellus sopii määriteltyihin vaatimuksiin.

Kuten Melzer artikkelissaan [Melzer] huomauttaa, niin joskus voidaan ihmetellä tällaisen testauksen hyödyllisyyttä, koska normaalit käyttäjät eivät osaa testausta, eivätkä ohjelmointia. Toisaalta tietämättömyys voidaan laskea käyttäjätestaajan eduksi, sillä hänellä on aivan eri näkökulma sovellukseen. Yleensä kuitenkin loppukäyttäjällä on paras tietämys aihealueesta ja alan toimintatavoista. Erityisesti lukiessaan dokumentaatiota ja ohjetiedostoja, voi käyttäjä olla suurena apuna virheiden havaitsemisessa. Näiden tutkiminen voi nimittäin jäädä vähälle huomiolle ohjelmoijilta ja testaajilta, joille toiminnallisuus on jo tuttua.

4.4.2 Alfa-testaus

Hyväksymistestauksen ensimmäistä vaihetta kutsutaan alfa-testaukseksi. Sen suorittavat todelliset käyttäjät yleensä sovelluksen kehittäneessä yrityksessä. Yleensä nämä käyttäjät ovat asiakkaan henkilökuntaa.

Testauksessa käytetään asiakkaan omaa, tuotantoympäristöä vastaavaa aineistoa, eikä enää käytetä aiempaa testausta varten luotua testidataa. Aineiston vaihdoksesta johtuen voidaan havaita uusia virheitä, koska yleensä todellinen aineisto on suurempi ja monipuolisempi kuin muissa testausvaiheissa käytetty.

Pressmanin [Pressman, luku 17] mukaan alfa-testauksessa sovelluksen aiempien vaiheiden testaajat ovat mukana tarkkailemassa testausta koko ajan, tarvittaessa neuvoen ja käyttäjän kaikki toiminnot rekisteröiden. Näin saadaan selville toiminnassa tapahtuvat virheet ja käyttöongelmat. Alfa-testauksessa voidaan myös havaita toiminnallisuuksia, jotka eivät vastaa asiakkaan toivomuksia tai käsityksiä sovelluksesta. Tämä testausvaihe jatkuu niin kauan, kunnes sovelluksen kehittäjä ja asiakas ovat yhtä mieltä sovelluksen hyväksyttävyydestä verratessaan tuloksia sovelluksen määrittelyyn.

4.4.3 Beta-testaus

Beta-testaus on testauksen vaiheista viimeisin ja sillä pyritään takaamaan, että sovellus on valmis julkaistavaksi. Beta-testauksen suorittavat loppukäyttäjät omassa käyttöympäristössään, eikä kehittäjä enää ole paikalla tarkkailemassa testausta. Vapaasti käytettäviä ohjelmistoja kehitetään usein niin sanottujen beta-versioiden kautta.

Pressman määrittelee kirjassaan [Pressman, luku 17] beta-testauksen olevan ulkoista testausta, jossa sovellus lähetetään lopullisia käyttäjäryhmiä mahdollisimman hyvin edustavalle joukolle testattavaksi todellisessa ympäristössä. Rajattu käyttäjäryhmä kokeilee sovellusta ja raportoi säännöllisin väliajoin sovelluksen toiminnasta ja mahdollisesta toimimattomuudesta, jotka ovat jääneet aiemmissa testeissä huomaamatta. Osa raportoiduista virheistä on usein käyttäjien omista epäilyistä ja väärinkäsityksistä johtuvia. Kehittäjien tulee arvioida raporttien joukosta todelliset virheet. Tarvittavat muutokset tehdään sovellukseen, mahdollisesti käyttäjille lähetetään uusi beta-versio testattavaksi ja lopulta tämän vaiheen jälkeen sovellus voidaan julkaista tuotantoon.

Patton listaa kirjassaan [Patton, luku 15] beta-testauksen tavoitteita, jotka saattavat vaihdella lehdistön antaman julkisuuden hakemisesta viimeiseen yritykseen havaita virheitä. Beta-testausta suunniteltaessa tulee ensinnäkin miettiä, ketkä ovat sopivia testaajiksi. Jos esimerkiksi vaiheen tärkein tavoite on havaita käytettävyysongelmia, ei testaajiksi kannata valita teknillisesti alemman tason toimintoihin suuntautuneita ihmisiä. Yleisesti pitäisi valita riittävä edustus kustakin loppukäyttäjäryhmästä.

Ehdottoman tärkeää on varmistaa, että testaajat todella käyttävät sovellusta. Patton korostaa, että usein testaajat saattavat käyttää sovellusta vain rajatun ajan ja kokeilevat vain pientä määrää toimintoja testausvaiheen lopussa, eivätkä siten oikeasti voi havaita virheitä. Kannattaakin asettaa vastuuhenkilö valvomaan, että testaajat viitsivät ja osaavat käyttää sovellusta suunnitelmien mukaisesti.

Erityisen hyödyllistä beta-testaus Pattonin mukaan on kokoonpano- ja yhteensopivuusongelmien havaitsemisessa. Jos käyttäjäryhmä valitaan järkevästi, saadaan testauksen käyttöön erilaisia konekokoonpanoja ja sovelluksia.

Tämä testausvaihe on hyvä myös käytettävyystestaukseen, sillä testaajat näkevät sovelluksen ensimmäistä kertaa. Lisäksi käyttäjät ovat eritasoisia käyttötaidoiltaan, joten he havaitsevat ominaisuudet, jotka ovat hämmentäviä tai vaikeita käyttää. Lisäksi käyttäjien erilaiset käyttötavat ja -tarpeet ohjaavat heidän sovelluskäyttöään eri suunnille. Joissain tapauksissa ne myös johtavat uusien ominaisuuksien ja toiminnallisuuksien toivomiseen luoden siten pohjaa uuden version kehittämiselle.

Patton huomauttaa kirjassaan [Patton, luku 15], että muita virheitä beta-testauksella on vaikea havaita varsin rajallisen testauksen takia ja toisaalta mahdollisia virheitä ei enää ehditä korjata kovin hyvin. Beta-testaukseen ei kannata luottaa liikaa, sillä se ei sovi oikeaksi testausmenetelmäksi. Hyvin suunniteltuna ja ohjattuna se antaa kuitenkin arvokkaan lisän muihin testeihin.

5 Testausmenetelmät

Testausmenetelmä määrittää tavan testata ja se ohjaa koko testausprosessin läpi. Se määrittää testauksen suunnittelun ja ajoituksen, antaa tarkoituksen ja päämäärän testaamiselle sekä tiedot, jotka tarvitaan koko testausprosessiin. Näin siis ainakin teoriassa hyvä testausmenetelmä tukee ohjelmistoprojektia. Luvun pääasialliset lähteet ovat [Patton, luvut 4-8] ja [Paakki, luvut 5-6].

5.1 Testausmenetelmien jako

Valittu testausmenetelmä vaikuttaa testitapausten valintaan. Usein testitapauksia valittaessa on käytettävissä kaksi päästrategiaa: musta- ja lasilaatikkotestaus. **Mustalaatikkotestauksessa** (engl. *black-box testing*) tiedetään vain määrittelyiden avulla, mitä sovelluksen kuuluisi tehdä sekä pyritään syötteiden ja vasteiden avulla päättelemään toiminnan oikeellisuus. **Lasilaatikkotestauksessa** (engl. *white-box testing, glass-box testing*) taas hyödynnetään koodia, jonka avulla pyritään havaitsemaan virheellisiä kohtia ja toiminnallisuuksia. Näiden lisäksi voidaan käyttää niiden yhdistelmää, harmaalaatikko-testausta, jolla pyritään hyödyntämään molempien menetelmien parhaat ominaisuudet.

Testitapausten valintamenetelmä määräytyy yleensä testauksen tason mukaisesti siten, että yksikkötestauksessa käytetään lasilaatikkomenetelmää, mutta siirryttäessä suurempiin kokonaisuuksiin kohti systeemi- ja hyväksymistestausta alkaa "laatikon väri" muuttua tummemmaksi. Tämä johtuu siitä, että kun yksiköt on integroitu systeemiin, rakenteelliset lasilaatikkomenetelmät tulevat mahdottomiksi käyttää, eikä niillä muutenkaan enää saavuteta haluttua tulosta. Lasilaatikkomenetelmillä keskitytään yksityiskohtiin, kun taas integroidussa systeemissä halutaan testata sen kokonaisvaltaista toimintaa.

Musta- ja lasilaatikkotestauksen ohella menetelmät jaetaan staattisiin ja dynaamisiin (engl. *static and dynamic*). **Dynaaminen testaus** yleisesti mielletään testaukseksi, sillä siinä ajetaan sovellusta ja pyritään käyttämällä havaitsemaan virheitä. Sitä vastoin **staattisessa testauksessa** ohjelmaa ei lainkaan ajeta. Testaaminen perustuu suunnitelman, arkkitehtuurin ja koodin tarkkailuun tarkoituksena löytää virheitä. Joskus tätä kutsutaan rakenteelliseksi testaukseksi (engl. *structural testing*).

5.2 Virheiden etsintää puutetestauksen avulla

Puutetestaus (engl. *defect testing*) on yleinen nimitys testaukselle, jonka tarkoituksena on paljastaa virheitä ennen sovelluksen julkaisemista. Puutetestaus onnistuu, jos se saa systeemin toimimaan väärin ja tuottamaan virheellisiä vasteita. Sommerville määrittelee kirjassaan [Sommerville, luku 20.1] puutetestauksen vastakohdaksi validointitestaamisen, jonka tarkoituksena on osoittaa järjestelmän toimivan oikein ja määrittelyjensä mukaisesti. Validointitestaus suoritetaan yleensä hyväksymistestitapausten avulla.

Patton luokittelee kirjassaan [Patton, luku 5] testaustavat kahteen erilaiseen luokkaan. Ensinnäkin on **hyväksymistestejä** (engl. *test-to-pass*), joissa pyritään osoittamaan sovelluksen toimivuus. Toisaalta on **hylkäämistestejä** (engl. *test-to-fail*), joilla sovelluksen virheet pyritään osoittamaan. Testaus kannattaa aloittaa aina helpommilla testeillä pyrkimällä osoittamaan sovelluksen toimivuus tavanomaisimmissa tapauksissa. Tällaisilla hyväksymistesteilläkin sovelluksesta saatetaan havaita runsaasti virheitä.

Kun sovelluksen tavanomainen toiminta on saatu varmistettua, voidaan siirtyä testaukseen, jonka tarkoituksena on pyrkiä kaatamaan sovellus. Tätä voidaan kutsua myös **virheiden pakotukseksi** (engl. *error forcing*). Tässä testaus kohdistetaan sovelluksen heikkouksiin ja testataan sovellusta sen äärirajoilla, suurella määrällä tietoa, yhtäaikaisilla tapahtumilla ja etsien mahdollisuuksia käyttää sovellusta jopa määrittelyjen vastaisesti.

5.3 Staattinen lasilaatikkotestaus

Lasilaatikkotestauksessa testaajalla on mahdollisuus tarkastella koodia ja käyttää sitä saadakseen vihjeitä testaukseensa sovelluksen heikoista paikoista. Lasilaatikkotestauksessa testaaja siis tavallaan näkee koodia sisältävän laatikon sisään, joten hän voi tarkastella koodin rakennetta ja toimintaa. Havaitsemansa rakenteen ja toiminnan avulla hän voi päätellä, millä arvoilla ja missä kohdissa testaus kannattaa suorittaa. Tosin testausta ei saa suorittaa koodin ehdoilla eli ei saa sovittaa testitapauksia koodin mukaisiksi. Erityisen hyödyllistä lasilaatikkotestauksessa on mahdollisuus löytää virheitä aikaisin. Toisaalta lasilaatikkotestauksella voidaan havaita virheitä, joita mustalaatikkotestauksella ei havaita.

Staattisessa lasilaatikkotestauksessa testataan sovellusta ajamatta sitä, siis tarkastuksin ja katselmoinnein (engl. *examine*, *review*). Useissa projekteissa ei kuitenkaan suoriteta staattista testausta, mikä johtuu lähinnä vääristä ennakko-oletuksista. Sitä pidetään aikaa vievänä ja kalliina, eikä muutenkaan tuottavana vaiheena. Ohjelmoijien halutaan vain kirjoittavan koodia, eikä käyttävän aikaansa muuhun.

5.3.1 Staattinen lasilaatikkotestaus on tiimityötä

Staattista lasilaatikkotestausta voidaan suorittaa ryhmätyönä **tarkastuksin** (engl. c*ode inspection*) ja **läpikäynnein** (engl. w*alk through*). Haikala ja Märijärvi [Haikala et al., luku 14] huomauttavat, että tarkastus, läpikäynti ja katselmointi termeinä sekoitetaan eri kirjallisuuksissa ja niiden terminologia ei ole kaikin puolin vakiintunutta. Tarkastuksessa on tarkoitus löytää virheitä dokumentaatiosta. Näin on myös läpikäynnissä, joka on pääasiassa koodiin keskittyvä sovellus tarkastuksesta. Katselmuksella taas tarkoitetaan vaiheen lopussa järjestettävää kokoontumista, jonka tarkoituksena on hyväksyä vaiheessa tuotetut dokumentit ja muut vaihetuotteet. Tämä ei siis varsinaisesti ole testausta, ja siinä virheiden löytäminen ei ole toivottua, toisin kuin testaukseen liittyvissä vaiheissa.

Tarkastus ja läpikäynti kuuluvat staattisen lasilaatikkotestauksen piiriin, mutta niitä voidaan käyttää koko ohjelmistoprojektin läpi kaikkiin dokumentteihin. Sen sijaan dynaamiset lasilaatikkomenetelmät liittyvät lähes pelkästään koodin tarkastukseen.

Staattinen lasilaatikkotestaus tehdään **muodollisina katsauksina** (engl. *formal review*). Patton listaa kirjassaan [Patton, luku 6] neljä sääntöä, jotka näitä katsauksia pidettäessä on muistettava. Ensinnäkin tarkoituksena on **havaita ongelmia**, kuten virheitä ja puuttuvia ominaisuuksia. Kaikki huomiot on kohdistettava koodiin, eikä sen tekijään. Toiseksi on **seurattava sääntöjä** katsauksen kestosta ja kommentoitavien aiheiden rajauksesta. Kolmanneksi jokaisen osallistujan olisi **valmistauduttava** tilanteeseen etukäteen lähetetyn materiaalin pohjalta, sillä suurin osa ongelmista löydetään ennen tapaamista. Lisäksi katsauksen tuloksista tulee **kirjoittaa raportti** kaikkien saataville.

5.3.2 Tarkastus- ja läpikäyntimenetelmät

Melzer [Melzer] kirjoittaa koodin tarkastuksen ja läpikäynnin olevan varhaisimpia niistä testausmenetelmistä, joissa ei käytetä tietokonetta apuna. Nämä ovat staattisia, perusidealtaan toisilleen läheisiä menetelmiä ja niiden käyttö on mahdollista jo hyvin varhaisessa vaiheessa sovelluskehitystä. Niillä mahdollistuu virheiden löytäminen aikaisin, joten korjauskustannukset ovat kohtuulliset. Käsitteet eroavat toisistaan siinä, että läpikäyntiä käytetään lähinnä koodin tarkastamiseen, kun taas tarkastusta voidaan käyttää myös dokumenttien tutkimiseen.

Tarkastusta toteutettaessa on ensin muodostettava ryhmä, jolle annetaan materiaalia tutustuttavaksi ja valmistautumista varten. Materiaali sisältää dokumentaation, joka on tarkoitus käydä läpi pari viikkoa myöhemmin pidettävässä kokouksessa. Materiaaliin tutustuminen etukäteen on olennaista, jotta tarkastustilaisuus saadaan toimimaan kunnolla. Tarkastuksessa on tarkoituksena käydä dokumentaatio läpi ja listata kaikki mahdolliset löytyneet viat. Paakki ohjeistaa luentomonisteessaan [Paakki, luku 4], että tarkoituksena ei ole etsiä ratkaisuja, vaan ainoastaan pyrkiä löytämään mahdollisimman paljon virheitä.

Tarkastusryhmä muodostuu yleensä vähintään viidestä henkilöstä, Paakin [Paakki, luku 4] mukaan sopivin koko on 6-8 henkilöä. **Johtajan** (engl. *manager*) roolissa on tärkeää olla asiantunteva henkilö, joka johtaa tapaamista, on vastuullinen prosessista ja myöhemmin kontrolloi, että löytyneet virheet korjataan. **Lukija** (engl. reader) käy dokumentaation läpi muille osallistujille valitsemansa loogisen järjestyksen mukaisesti.

Tarkastusryhmään valittu **suunnittelija** (engl. *design engineer*) on suunnittelun asiantuntija, mutta hän ei kuitenkaan saa olla suunnittelijana testattavassa sovelluksessa. **Ohjelmoijan** tehtävään valitaan henkilö, jolla on laaja tietämys ja tuntemus testattavan sovelluksen ohjelmointikielestä ja yleisesti ohjelmoinnista. Hän ei saa olla testattavan ohjelman kehittäjä. **Testaajan** roolissa on henkilö, jolla on kokemusta tällaisesta sovelluksesta. Melzer selvittää kirjoituksessaan [Melzer], että testaajan tehtävänä on kontrolloida tarkastusprosessin kulkua tavalla, jolla löytyy eniten virheitä.

Tarkastusryhmään tarvitaan myös **sihteeri** kirjaamaan tarkastuksen aikana ilmaantuneet ongelmat. Lisäksi mukana on ohjelman **tekijä** (engl. *author*), joka selvittää muille koodissa mahdollisesti olevia epäselviä kohtia. Joskus myös **tavallinen käyttäjä** osallistuu edustaen tavallisen käyttäjän näkökulmaa. Hän ehkä saavuttaa kommentoinnillaan sen, että lopullinen tuote on muidenkin käyttäjien oletuksien kaltainen.

Koodin tarkastuksessa ohjelman esittelijä esittää koodin ja selittää sen ryhmälle. Ryhmä voi kysyä kaikkia epäselviä asioita esityksen aikana. Patton huomauttaa kirjassaan [Patton, luku 6], että esittäjänä ei saa olla itse koodin tekijä. Tällöin ainakin joku toinen lukee koodin ja joutuu todella ymmärtämään sen. Esitettäessä koodia ryhmälle kaikki osallistujat yleensä löytävät yllättävän paljon virheitä. Lukija ohjaa keskustelua siinä määrin, että ryhmä todella etsii virheitä, muttei ratkaisuja niihin. Ratkaisut ohjelman tekijä miettii itse myöhemmin ja esittää ne ryhmälle mahdollisessa myöhemmässä palaverissa.

Koodin tarkastus on suunniteltava huolellisesti. Haikala ja Märijärvi huomauttavat kirjassaan [Haikala, luku 14], että tarkastuksen ajankohta on olennainen ja sen kesto ei saa ylittää kahta tuntia. Isoissa projekteissa on järjestettävä useampi palaveri tai tauotettava palaveri kunnolla. Paakki määrittelee luentomonisteessaan [Paakki, luku 4] yleisen tarkastusvauhdin dokumentaatioille olevan korkeintaan 20 sivua tunnissa. Koodirivejä voidaan huolellisesti tarkastaa korkeintaan 90 riviä tunnissa. Tarkastuksissa ilmapiirin on oltava hyvä, eikä virheitä saa suunnata moitteena tekijää kohtaan. Kun virheet pidetään ohjelman osana, on katselmuksella hyvät mahdollisuudet tuottoisiin tuloksiin.

Läpikäynti on tekniikkana samantyylinen koodin tarkastuksen kanssa, joskin se on epämuodollisempi. Läpikäynnin aikana kuitenkin tarvitaan monen ihmisen läsnäoloa, joten siinä on syytä seurata yleisiä sääntöjä. Patton esittää kirjassaan [Patton, luku 6], että läpikäynnissä koodin kehittäjä esittelee koodin muille osallistujille selittäen, mitä koodi tekee ja miksi. Muut osallistujat seuraavat esitystä sekä esittävät kysymyksiä ja kommentteja kaikista epäselvistä tai arveluttavista kohdista sovelluksen kehittäjälle. Melzer taas toteaa artikkelissaan [Melzer], että itse sovelluskehittäjä ei saisi osallistua läpikäyntiin, vaan koodin ja siihen liittyvän dokumentaation olisi oltava selkeää ja muiden ymmärrettävissä olevaa ilman lisättyjä selityksiä.

5.3.3 Menetelmien hyvät ja huonot puolet

Yleisesti staattisen lasilaatikkotestauksen menetelmien pääperiaatteena on koodin esittely muille sovelluskehittäjille. Melzer [Melzer] pitää hyvänä asiana sitä, että ohjelmoija tietää jo koodia kirjoittaessaan, että se on myöhemmin esitettävä muille. Tällöin ohjelmoija pyrkii jo alunperin toteuttamaan selkeän ja helposti luettavissa olevan rakenteen ohjelmaan. Tämän jälkeisessä esityksessä ohjelman sisäinen rakenne on vielä uudelleen tarkkailtavana, joskin päätarkoituksena on havaita virheitä.

Melzer näkee staattisen testauksen menetelmät hyvinä, koska ne voidaan suorittaa projektin varhaisessa vaiheessa, jolloin virheiden korjauskustannukset ovat alhaiset. Ne myös soveltuvat lähes kaikkiin projekteihin. Yhteistyössä suoritettu testaus opettaa koko ryhmää ja ehkä vastaavilta virheiltä pystytään paremmin välttymään vastaisuudessa. Toisaalta tarkastusten ja läpikäyntien aikana myös mustalaatikkotestaukseen saadaan vihjeitä mahdollisista virhepaikoista.

Patton mainitsee kirjassaan [Patton, luku 6] vielä kaksi muuta staattisen testauksen hyvää puolta. Ensinnäkin tämän testauksen toteuttamisen aikana testaajat ja ohjelmoijat työskentelevät yhdessä. Tällöin he voivat oppia paremmin arvostamaan ja ymmärtämään toistensa työtä ja taitoja. Toiseksi tässä testauksessa voidaan löytää ratkaisuja olennaisiin ongelmiin. Yleensä tämän testauksen aikana ei kuitenkaan haluta etsiä ratkaisuja, vaan ainoastaan virheitä. Ratkaisut voidaan selvittää jälkikäteen käyttämättä kaikkien testaukseen osallistuvien ihmisten aikaa.

Staattisen lasilaatikkotestausmenetelmän haittana on se, että sitä käytettäessä vaaditaan paljon valmisteluja ja monen ihmisen läsnäoloa, josta aiheutuu suuria kustannuksia. Melzer muistuttaa raportissaan [Melzer], että tässä menetelmässä osallistujilta vaaditaan tarkkaa keskittymistä ja kurinalaisuutta sekä ymmärrystä kohdistaa kritiikki ohjelmaan, ei sen tekijään. Haittapuolina on lisäksi se, että näiden testien toistaminen voi olla vaikeaa ja aina ei testauksen aikana ole lähdekoodia saatavilla.

5.4 Dynaaminen lasilaatikkotestaus

Lasilaatikkotestausta kutsutaan rakenteelliseksi testaukseksi, koska testaus perustuu järjestelmän sisäisen rakenteen tutkimiseen. Koska testattava ohjelma tunnetaan kooditasolla, voidaan arvioida mahdollisia virhealttiita paikkoja ja keskittää testaus näihin. Dynaamisessa lasilaatikkotestauksessa ohjelman sisäisen rakenteen avulla rakennetaan testitapaukset, jotka suoritetaan ja tuloksia analysoidaan suhteessa odotettuihin tuloksiin.

5.4.1 Dynaamisen lasilaatikkotestauksen tavoitteet ja ongelmat

Lasilaatikkotestaus on välttämätöntä, koska koodin ominaisuuksien takia kaikkia virheitä ei mustalaatikkotestauksella havaita. Mustalaatikkotestauksella taataan sovelluksen toiminta tuotteen määrittelyyn nähden. Pressman listaa kirjassaan [Pressman, luku 16] monia muita ominaisuuksia, joiden toiminta dynaamisella lasilaatikkotestauksella voidaan taata. Ensinnäkin ihanteellisessa tilanteessa dynaamisella lasilaatikkotestauksella taataan kaikkien toimintapolkujen kokeilu vähintään kerran. Lisäksi testataan loogisten päätöslauseiden suorittaminen sekä arvolla tosi että epätosi. Ihanteellisessa tapauksessa lasilaatikkotestauksella voitaisiin taata myös silmukoiden toiminta raja-arvoilla ja arvoalueen sisälle mahtuvilla syötteillä. Lisäksi tässä testauksessa kokeillaan sisäiset tietorakenteet niiden oikeellisuuden varmistamiseksi.

Testausta vaikeuttaa virheen monimutkaisuus, sillä virhe muodostuu usein monen tekijän vaikutuksesta. Testaukseen annettavan syötteen X lisäksi testaustulokseen vaikuttaa sovelluksen sisäinen tila Z. Näiden perusteella saadaan vaste Y, josta sovelluksen oikeellisuutta mustalaatikkotestauksessa arvioidaan. Lasilaatikkotestauksessa päästään tarkastelemaan myös sisäisten tilojen muuttumista. Mahdollisuus tarkastella sisäisiä tiloja nostaa tilojen määrää vielä runsaasti. Kaikkia mahdollisia tiloja ei voidakaan testata, joten on syytä miettiä, milloin testaus on riittävää.

Dynaamista lasilaatikkotestausta voidaan suorittaa lähes samoin kuin debuggausta. Patton muistuttaa kirjassaan [Patton, luku 7], että näiden tavoitteet ovat kuitenkin varsin erilaiset. Dynaamisella lasilaatikkotestauksella pyritään havaitsemaan virheitä, kun taas debuggauksen avulla pyritään paikallistamaan ja korjaamaan jo havaittuja virheitä.

5.4.2 Testauksen kattavuus ja hyöty verrattuna resursseihin

Testauksen laatua arvioidaan usein sillä, miten täydellisesti ohjelma on testattu. **Kattavuusmitoilla** voidaan mitata, kuinka perusteellisesti lasilaatikkotestit kattavat ohjelman toiminnot. Testaamisen kattavuuden arviointiin sopivia mittausmenetelmiä ovat esimerkiksi lausekattavuus, polkukattavuus ja erilaiset ehtokattavuudet, joita käsitellään jäljempänä luvussa 5.4.4. Käytännössä sadan prosentin kattavuutta ei saavuteta edes pieniä ohjelmia testattaessa.

Vaikka lasilaatikkotestausta käytetään vain testauksen alemmilla tasoilla, ei kaikkea voida testata suoraviivaisesti. Pressman esittää kirjassaan [Pressman, sivu 470] esimerkin pienestä aliohjelmasta, jossa on kaksi sisäkkäistä silmukkaa, jotka molemmat tulee suorittaa arvoilla 1-20. Tässäkin polkujen määrä niin korkea kuin 10^{14}, jolloin jopa nopealta tietokoneelta kuluisi näiden täydelliseen suorittamiseen aikaa 3170 vuotta. Täten on ymmärrettävää, että testausaikaa pyritään lyhentämään, mutta kuitenkin säilyttäen riittävä takuu sovelluksen oikeellisuudesta.

Testikattavuus on helpointa saada korkeaksi yksikkötasolla. Harvoin sadan prosentin kattavuutta saavutetaan sielläkään, sillä esimerkiksi lausekattavuudessa todellinen kattavuus nousee harvoin yli 80 prosenttiin. Projektin edetessä siirryttäessä V-mallin mukaisesti yksikkötasolta systeemitestauksen suuntaan testikattavuus vähenee. Haikalan ja Märijärven kirjan [Haikala et al., luku 15] mukaan kattavuus asiakasprojekteissa ei yleisesti nouse 50 prosentin yläpuolelle. Testikattavuus ei takaa sovelluksen toimivuutta, mutta sitä pidetään kuitenkin tärkeänä. Se nimittäin tarjoaa suuntaa-antavan ja johdonmukaisen arvon siitä, miten järjestelmällistä ja johdonmukaista sovelluksen testaus on ollut.

Marick pitää artikkelissaan [Marick, 1997] kaikkien kattavuusmittojen perustavana ongelmana sitä, että ne varmistavat vain niille annetun koodin oikeellisuutta. Jos siis koodista puuttuu esimerkiksi ehtolauseissa jokin oleellinen ehtoarvo, mikään kattavuusmitoista ei yleensä huomaa arvon puuttumista, vaan tällaiset asiat ovat yksin testaajan vastuulla. Tällöin osan koodista puuttuessa ei tietenkään kattavuusmitalla enää saavuteta oikeaa tulosta.

Mitään kattavuusmitoista ei koskaan tulisi käyttää ainoana takuuna testauksen ja siten välillisesti ohjelmiston laadusta. Kattavuustyökalujen voidaan ajatella olevan hyödyllisiä, jos niitä käytetään lisäämään ajattelua, ei korvaamaan sitä.

5.4.3 Dynaamisia lasilaatikkotestausmenetelmiä

Staattisessa lasilaatikkotestauksessa testausta tehdään sitoutuneena ryhmätyöhön, mutta käytössä on muitakin ja nimenomaan yksin suoritettavissa olevia testausmenetelmiä. Melzerin [Melzer] mukaan staattisen lasilaatikkotestauksen menetelmät vaativat suuresti keskittymistä ja kontrollointia. Ihmiset usein haluavatkin käyttää helpompia tapoja ja menetelmiä, joilla voidaan parantaa testauksen systemaattisuutta. Dynaamisen testauksen menetelmät antavat huomattavasti systemaattisemman lähestymistavan.

Dynaamisen testauksen menetelmät voidaan jakaa kolmeen ryhmään: kontrollin kulku, silmukkatestaus ja tietovirtatestaus. Näistä kerrotaan luvuissa 5.4.4, 5.4.6 ja 5.4.7. Käyttäen luetelluista menetelmistä tiettyä voidaan samalla varmistaa sovelluksen kattavuus kyseisellä osa-alueella. Esimerkiksi kontrollin kulun menetelmään kuuluvaa polkutestausta käytettäessä taataan polkukattavuus eli sovelluksen toiminta jokaisella suorituspolulla.

5.4.4 Kontrollin kulku

Ensimmäinen dynaamisen lasilaatikkotestauksen tekniikoista on kontrollin kulku (engl. *control flow*). Tässä pakotetaan sovellus kulkemaan erilaisia polkuja pitkin sitä suoritettaessa. Eri kontrollin kulun menetelmissä kuljettavat polut valitaan kullekin menetelmälle ominaisten piirteiden mukaisesti. Kontrollipolkujen läpikäyntiin keskittyvät tekniikat ovat lausekattavuus, erilaiset ehtokattavuudet sekä polkukattavuus. Näillä on olemassa keskinäinen järjestys siitä, mikä on vahvempi menetelmä kuin toinen. Näistä polkutestaus käsitellään laajimmin, ja on siksi esitetty omassa luvussaan 5.4.5.

Yleisin kattavuusmitoista on **lausekattavuus** (engl. *statement coverage*). Sataprosenttista kattavuus on, kun jokainen ohjelman lause on suoritettu ainakin kerran. Sadan prosentin lausekattavuus kuulostaa yksinkertaiselta vaatimukselta. Todellisessa projektissa tätä on todella vaikeata saavuttaa, sillä yksinkertaisesti lauseiden määrä on valtava. Tarkkailemalla lauseiden käyttöä voidaan löytää lauseet, joita ei koskaan suoriteta. Näissä saattaa siis olennaisimmin olla virheitä. Patton huomauttaa kirjassaan [Patton, luku 7], että tämä testaus kuitenkin takaa vain, että jokainen lause on suoritettu kerran. Mutta se ei kuitenkaan takaa, onko kaikki ohjelman polut käyty läpi.

Toinen kattavuusmitoista on **polkukattavuus** (engl. *path testing*), jossa pyritään kattamaan kaikki sovelluksen polut. Tätä käsitellään luvussa 5.4.5. Kattavuustestauksen kolmannen menetelmän, **ehtotestauksen** (engl. *condition testing*), Pressman esittelee kirjassaan [Pressman, luku 16] menetelmänä, jossa suoritetaan sovelluksen osan loogiset ehdot. Tarkoitus olisi suorittaa sovelluksen jokainen mahdollinen ehto. Ehtojen testaukseen kehitettyjä menetelmiä ovat päätös-, ehto- ja moniehtokattavuus.

Haikala ja Märijärvi määrittelevät kirjassaan [Haikala et al., luku 15], että **päätöskattavuudessa** (engl. *decision coverage*) jokainen ehto saa testattaessa molemmat arvonsa. Tämä on tavallaan yksinkertaisin ehtojen joukossa, sillä testaus suoritetaan lausekkeen ollessa sekä tosi että epätosi. **Ehtokattavuudessa** (engl. *condition coverage*) päätöksen kaikkien ehtojen on saatava kaikki arvonsa.

Selvennetään kattavuusmittoja seuraavan esimerkkiohjelman avulla. Tämä esimerkki eri kattavuusmitoille on esitetty taulukoituna liitteessä 2.

```
void paivays( int kk, int pv){
if ( kk=2 and pv=29) {
      println("On karkauspäivä. "); }
println("Tänään on " + pv + "." + kk + ".");
}
```

Jos testataan lausekattavuutta, on ainoastaan jokainen lause suoritettava. Siis sataprosenttinen kattavuus saadaan suorittamalla rivit 1, 2, 3, 4 ja 5. Tämä onnistuu muuttujien arvoilla `kk=2` ja `pv=29`. Päätöskattavuutta käytettäessä kaikkien ehtojen on saatava molemmat arvonsa. Tällöin siis tarvitaan tapaukset, joissa ehto saa ensin totuusarvon ”tosi” ja toisella kertaa arvon ”epätosi”. Tässä tapauksessa tarvittaisiin kaksi testiä eli ensin suorittaen rivit 1, 2, 4, 5 ja sitten 1, 2, 3, 4, 5. Ehtokattavuudessa sitä vastoin vaaditaan kaikkien ehdossa olevien muuttujien arvojen läpikäynti. Tässä tapauksessa tämä saavutetaan esimerkiksi arvopareilla `kk=2` ja `pv=1` sekä `kk=1` ja `pv=29`. Nämä molemmat tuottavat arvon epätosi, jolloin molemmilla kerroilla suoritetaan rivit 1, 2, 4, 5.

Ehtokattavuuksien luokitteluun kuuluu myös **moniehtokattavuus** (engl. *multiple condition coverage*). Moniehtokattavuudessa testaus on suoritettava kaikkien ehtojen kaikilla yhdistelmillä. Edellisessä esimerkissä siis on suoritettava neljä eri testiä. Yhdessä testissä molemmat ehtomuuttujat ovat totta ja toisessa molemmat ovat epätotta. Sen lisäksi tarvitaan testit, joissa toinen arvoista on vuorollaan tosi ja toinen epätosi. Tähän siis tarvittavat mahdolliset tapaukset ovat (`kk=2`, `pv=29`), (`kk=1`, `pv=1`), (`kk=2`, `pv=1`) ja (`kk=1`, `pv=29`).

Paremmuusjärjestys kattavuusmitoille määritellään yleisesti sen mukaan, mikä mitoista on vahvin eli minkä voimassa ollessa myös muut ovat voimassa. Jos siis tietyllä mitalla tehty testi takaa myös toisella tehdyn testin, on ensimmäinen vahvempi. Päätöskattavuudesta seuraa lausekattavuus. Ehtokattavuudella ja päätöskattavuudella ei kahdestaan ole varsinaista suhdetta, kuten käsitellystä esimerkistä selvisi. Moniehtokattavuudesta sen sijaan seuraa sekä päätös- että ehtokattavuus. Täten moniehtokattavuuden voi sanoa olevan vahvin mitta.

5.4.5 Polkutestaus

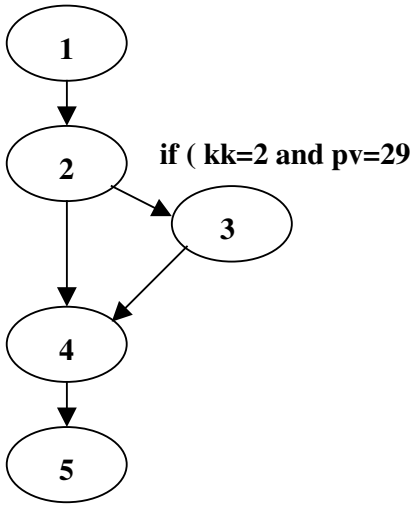
Yksi kontrollin kulun testauksen tekniikoista on polkutestaus (engl. *path testing*). Kyseinen rakenteellinen testausmenetelmä on yksi vanhimmista menetelmistä ja sitä käytetään lasilaatikkotestauksen lisäksi myös mustalaatikkotestauksessa. Melzer kirjoittaa artikkelissaan [Melzer] polkutestauksen olevan samankaltainen läpikäynnin (luku 5.3.2) kanssa. Ensin valitaan testattava polku, määritellään syötteet ja oikeat vasteet. Ohjelma suoritetaan manuaalisesti ja tuloksia verrataan alussa määritettyihin, minkä jälkeen mahdolliset virheet raportoidaan.

Kun komponentit integroidaan kokonaisuudeksi, rakenteellisten testausmenetelmien käyttö ei enää ole mahdollista. Ensinnäkin polkujen määrä kasvaa valtavaksi ja toisaalta integroinnin jälkeen testauksen luonne ja tavoitteet muuttuvat. Polkutestausta käytetäänkin yleisesti vain yksikkötestauksessa. Polkutestauksessa tarkoituksena on käydä läpi jokainen komponentin tai ohjelman polku. Tämä takaa, että jokainen ohjelman lause tulee suoritettua vähintään kerran. Ohjelman ehtolauseet testataan sekä arvolla tosi että epätosi.

Polkujen valinta voidaan tehdä eri tavoin. Optimaalista olisi käydä läpi kaikki eri vaihtoehdot, mutta se ei ole mahdollista, sillä esimerkiksi silmukkarakenteet aiheuttavat polkujen määrän suuren kasvun. Jos esimerkiksi luvussa 5.4.4 esitetty sovellus toteutetaan silmukan sisällä ja silmukka tulee suorittaa 20 kertaa, on kaikkien mahdollisten polkujen määrä 2^{20} eli yli miljoona. Täten polkutestauksessa yleensä suoritetaan ainoastaan **itsenäiset polut** (engl. *independent path*), joita esitetyssä esimerkissä on kaksi eli `if`-lauseen arvolla tosi ja epätosi. Tietysti silmukkarakenne vaatii lisää testaamista, ja sitä käsitelläänkin luvussa 5.4.6.

Polkutestauksen ensimmäiseksi tehtäväksi Sommerville määrittää kirjassaan [Sommerville, luku 20] **vuokaavion** (engl. *flow graph*) määrittämisen. Vuokaaviossa ohjelma esitetään yksinkertaistettuna rakenteena siten, että päätöskohtia esittävät solmut (engl. *node*) ja ohjelman kulkua solmujen väliset linkit eli reunat (engl. *edge*). Ehtolauseet tosi ja epätosi sekä silmukat aiheuttavat kaavioon uuden polun syntymisen. Vuokaavion avulla voidaan tarkistaa, että kaikki polut huomioidaan ja ne tulevat testatuiksi vähintään yhden kerran. Täten jokainen lause ja ehtolauseet molemmilla arvoillaan tulevat testatuiksi.

Kuva 9 esittää vuokaavion luvun 5.4.4 esimerkkiohjelmalle. Solmuissa on aina esitetty rivin numero. Tämä kaavio on varsin yksinkertainen, koska esimerkki ei sisältänyt silmukoita. Tässä on vain kaksi itsenäistä polkua, jotka ovat 1, 2, 4, 5 ja 1, 2, 3, 4, 5.



Kuva 9. Esimerkki vuokaaviosta.

Vaadittujen testitapausten vähimmäismäärä saadaan selville käyttämällä vuokaavion (engl. *flow graph*) avulla laskettavissa olevaa **kompleksisuutta** CC (engl. *cyclomatic complexity*). Sommerville määrittelee kirjassaan [Sommerville, luku 20] kaaviolle K seuraavan laskukaavan:

$$CC(K) = \text{Linkkien lukumäärä} - \text{Solmujen lukumäärä} + 2$$

Vähimmäismäärä testitapauksille on yhtä suuri kuin CC. Kyseisen arvon löydyttyä on suunniteltava sopivat testaukset kullekin näistä poluista. Kuvassa 9 esitetyssä esimerkkisovelluksessa sekä solmuja että linkkejä on viisi, joten CC saa arvon 2. Täten testauksessa on käytettävä vähintään kahta testitapausta. Sommerville määrittelee toisen yleisen ohjeen laskea CC ilman vuokaavion piirtämistäkin. Jos ohjelmassa ei ole `goto`-lausetta, CC on yhtä suurempi kuin ehtojen määrä. Esimerkiksi ohjelmalle, jossa on viisi `if`-lausetta ja kaksi `while`-silmukkaa, kompleksisuus on kahdeksan.

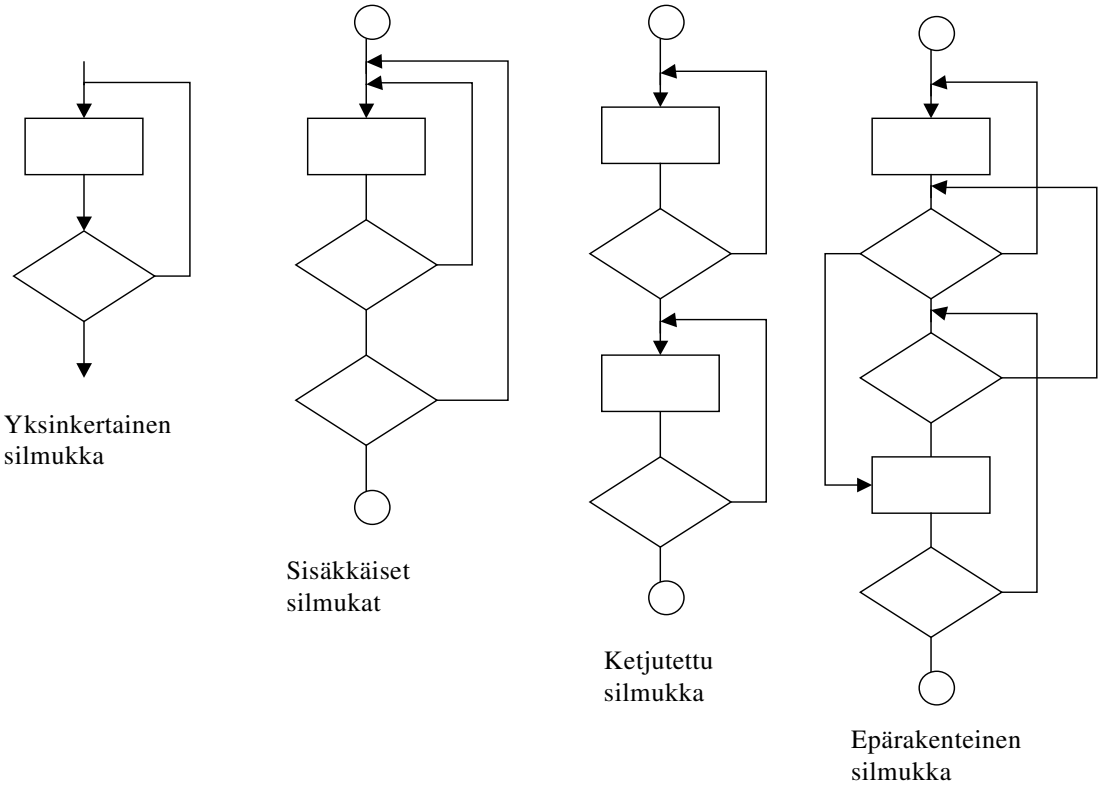
Sommerville kehottaa kirjassaan [Sommerville, luku 20] polkutestausta sovellettaessa käyttämään apuna **dynaamisia ohjelman analysoijia**. Kun ohjelma käännetään, analysoija lisää koodiin tiettyjä ohjeita. Nämä laskevat, montako kertaa kukin ohjelman lauseista on suoritettu ajon aikana. Ohjelman ajon jälkeen näistä laskelmista saadaan suoritusprofiili. Lisäksi analysoija listaa lauseet, joita ei ole suoritettu. Näin tällä profiililla voidaan helposti saada arvio testikattavuudelle.

5.4.6 Silmukkatestaus

Toinen dynaaminen lasilaatikkomenetelmä on silmukkatestaus (engl. *loop testing*). Silmukoiden testaaminen on haastavaa, koska kaikkien mahdollisuuksien läpikäynti on usein mahdotonta. Paakki myös huomauttaa luentomonisteessaan [Paakki, luku 6], että kehittäjät tekevät helposti paljon virheitä silmukoissa, joten näiden testaaminen on todella tärkeää. Beizer kirjoittaa [Beizer, luku 3] tavanomaisimpien näistä virheistä johtuvan alustusvirheistä sekä toisaalta silmukan maksimikierrosmäärän ylityksestä.

Pressman jakaa kirjassaan [Pressman, luku 16] silmukat neljään eri luokkaan. Näitä luokkia ovat **yksinkertaiset silmukat** (engl. *simple loops*), **sisäkkäiset silmukat** (engl. *nested loops*), **ketjutetut silmukat** (engl. *concatenated loops*) ja **epärakenteiset silmukat** (engl. *unstructed loops*). Silmukoiden eri tyypit on esitetty kuvassa 10.

Melzerin tutkielman [Melzer] mukaan **ohjelman tila ja oikeellisuus tulisi testata** kolmessa eri vaiheessa: **silmukkaan tultaessa, silmukan suorituksen aikana ja siitä poistuttaessa**. Vaikeimpia ovat silmukat, jotka ovat osittain sisäkkäisiä. Sisäkkäisyyden vuoksi muun muassa `goto`-lauseen käyttö aiheuttaa suuria ongelmia ja se onkin kielletty monissa yhteyksissä.



Kuva 10. Silmukoiden tyypit [Pressman, luku 16].

Yksinkertaisten silmukoiden testaamiseen Pressman antaa kirjassaan [Pressman, luku 16] seuraavat vaiheet suoritettavaksi. Tässä *n* on silmukan suorituskertojen maksimi.

1. Hyppää silmukan yli eli suorita se arvolla nolla.
2. Suorita silmukka kerran eli arvolla yksi. Tällä suorituksella havaitaan kaikki yleisimmät alustusvirheet.
3. Suorita silmukka kahdesti eli arvolla kaksi.
4. Suorita silmukka arvolla *m*, joka on tavanomaisin silmukan suorituksessa.
5. Suorita silmukka arvoilla *n-1*, *n* ja *n+1*. Tässä sovelluksen olisi estettävä silmukan suoritus arvolla *n+1*.

Tähän listaukseen Beizer lisää kirjassaan [Beizer, luku 3] vielä yhden kohdan. Jos joku arvo silmukan suorituskertojen minimi- ja maksimiarvojen välillä on kielletty, niin silmukka tulee testata myös tämän arvon *a* läheisyydessä. Tämä tarkoittaa arvoilla *a-1*, *a* ja *a+1*.

Laajennettaessa yksinkertaiset silmukat **sisäkkäisiksi** mahdollisten suorituspolkujen määrä kasvaa runsaasti ja samalla tietysti vaadittujen testitapausten määrä. Jotta suorituskertojen määrää saadaan vähennettyä, Pressman kehottaa testaamaan seuraavien vaiheiden mukaisesti:

1. Aseta kaikki silmukkamuuttujat pienimpään arvoonsa ja aloita sisimmästä silmukasta.
2. Suorita testaus sisimmälle silmukalle käyttäen yksinkertaiselle silmukalle annettuja testausohjeita.
3. Siirry seuraavaksi sisimpään silmukkaan. Pidä sisempien silmukoiden arvot tavanomaisissa arvoissa ja ulompien arvot minimissä.
4. Jatka tätä kunnes kaikki silmukat on testattu.

Beizer lisää tähän listaukseen vielä tapauksen, jossa testaus suoritetaan kaikkien silmukoiden ollessa maksimiarvossaan.

Pressman kirjoittaa kirjassaan [Pressman, luku 16], että **ketjutetut** silmukat voidaan testata yksinkertaisten silmukoiden menetelmää käyttäen, jos ketjuun kuuluvat silmukat ovat kukin itsenäisiä. Jos taas näillä silmukoilla on yhteisiä muuttujia, ne eivät ole itsenäisiä ja ne tulee testata sisäkkäisten silmukoiden menetelmiä käyttäen.

Epärakenteisten silmukoiden testaaminen on todella vaikeaa. Epärakenteisiin silmukoihin yleisesti ottaen tulee helpommin virheitä koodausvaiheessa ja niitä varten on huomattavan vaikeaa suunnitella testejä. Paakki ohjeistaakin luentomonisteessaan [Paakki, luku 6], että aina mahdollisuuksien salliessa tällaiset silmukat tulisi ennen testausta kirjoittaa uudelleen helpommiksi silmukkarakenteiksi testauksen mahdollistamiseksi.

5.4.7 Tietovirtatestaus

Sovellukseen kuuluvat luvuissa 5.4.4 ja 5.4.6 kuvatut ehto- ja toistorakenteet, mutta siihen kuuluu myös data. Eräs rakenteisen testauksen alue on tietovirtatestaus (engl. *data flow testing*), jossa tarkastellaan tiedon kulkua sovelluksessa.

Patton määrittelee kirjassaan [Patton, luku 7], että yksikkötasolla tämä tarkoittaa tiedon kuljettamista yksittäisen moduulin läpi. Systeemitestaustasolla se tarkoittaisi tiedon kuljettamista koko ohjelman läpi, mikä olisi varsin aikaa vievää. Lasilaatikkotestauksessa tiedon kulkua voidaan tarkkailla syötteistä ja vasteista sekä sovelluksen sisäisten muuttujien arvojen muutoksista debuggerilla sovellusta ajettaessa.

Tietovirtatestauksessa tarkastellaan tietorakenteiden tilamuutoksia ohjelman suorituksen aikana. Tilamuutoksina huomioidaan arvon asettaminen ja arvon hyväksikäyttö. Paakki huomauttaa luentomonisteessaan [Paakki, luku 6], että koska tietovirtoja tutkittaessa tarvitaan tietoa kontrollipoluista, niin kontrollin kulun testaus on aina suoritettava ensin.

Tietovirtatestausta varten on laadittava tietovirtakaavio. Paakki kuvaa tämän olevan kontrollin kulussa käytettyä vuokaaviota vastaava (katso luku 5.4.5), joskin sen laajennettu versio. Tietovirtakaavioon on lisätty erityisiä tiedon muutoksiin liittyviä osia. Tämän testauksen menetelmät perustuvat tämän kaavion läpikäyntiin. Testattaessa tarkastellaan erityisesti tilanteita, kun muuttuja on alustettu, muuttujan arvo poistetaan tai muuttujaa käytetään. Menetelmän käyttöä on tarkemmin esitetty lähteessä [Paakki, luku 6.3].

Erityisesti koodissa käytettyjen kaavojen tarkastelu kuuluu Pattonin mukaan [Patton, luku 7] tietovirtatestauksen piiriin. Koska koodi on saatavilla, voi testaaja perustaa testinsä erityisiin kaavojen kannalta kriittisiin arvoihin.

Tiedon testauksessa voidaan suorittaa erityinen virhetestaus. Sovelluksen arvot ovat näkyvissä koko testauksen ajan, joten sovellukselle voidaan syöttää arvoja, joilla saadaan tuotettua vuorotellen kukin virhetilanne. Tällaista erityistä virheiden tuottamista suoritettaessa on huomioitava, ettei yritä pakottaa sovellusta toimimaan arvoilla, joita se ei oikeasti ikinä voi saada. Tämä on kuitenkin hyvä tapa testata kaikkien virheviestien kunnollisuus.

5.4.8 Hyötyjä ja haittoja

Dynaamisen lasilaatikkotestauksen menetelmien hyvinä puolina on niiden soveltuvuus useimpiin ohjelmiin. Lisäksi niiden ajankulutus on pieni, koska ne on mahdollista ottaa käyttöön jo sovelluksen varhaisessa vaiheessa. Melzer on artikkelissaan [Melzer] tyytyväinen siihen, että näiden menetelmien avulla tulee käytyä läpi kaikista tavanomaisimmat virhetilanteet.

Haittapuolien Melzer mainitsee olevan samankaltaisia kuin staattisen lasilaatikkotestauksen yhteydessä esitetyissä ryhmätyönä suoritettavissa testeissä. Suuri haittapuoli on lähdekoodin välttämättömyys, sillä aina koodia ei testausvaiheessa ole saatavilla. Lisäksi testejä on vaikea suorittaa uudelleen ja niitä käytettäessä onnistuminen riippuu paljon suorittajan kärsivällisyydestä ja itsekontrollista.

5.5 Mustalaatikkotestaus

Mustalaatikkotestauksessa sovelluksen sisäistä toimintaa ei testata, vaan ainoastaan sen syötteet ja vasteet ovat testauksen piirissä. Ohjelmalle annetaan syöte käsiteltäväksi ja sovelluksen toiminnan oikeellisuus testataan vertaamalla vastetta määrittelyjen mukaiseen oikeaan tulokseen. Tässä testauksessa päästään eroon lasilaatikkotestauksen haittapuolesta eli lähdekoodin tarpeesta testaamista varten. Luku keskittyy pääasiassa dynaamiseen mustalaatikkomenetelmään ja staattista mustalaatikkotestausta käsitellään vain luvussa 5.5.2. Luvun pääasialliset lähteet ovat [Patton, luvut 4-5] ja [Sommerville, luku 20].

5.5.1 Periaate ja tavoitteet

Mustalaatikkotestausta kutsutaan myös funktionaaliseksi (engl. *functional*) testaamiseksi. Siinä nimittäin tutkitaan sovelluksen käyttäytymistä määrittelyihin ja arkkitehtuuriin verrattuna, eikä ohjelman koodin toimintatapaan kiinnitetä huomiota. Paakki määrittelee luentomonisteessaan [Paakki, luku 5] mustalaatikkotestauksen perusajatuksen varsin yksinkertaisesti. Ensin sovellukselle annetaan tietty syöte x, ja saadaan sovelluksen vaste $f(x)$. Vastetta $f(x)$ verrataan tunnettuun oikeaan vasteeseen y. Jos $f(x)=y$, testi on läpäisty, muuten on havaittu virhe.

Mustalaatikkotestaus kuuluu pääasiallisesti Sommervillen kirjassaan [Sommerville, luku 20.1] esittelemään puutetestaukseen (engl. *defect testing*), joten onnistuessaan testi tuottaa virheellisiä vasteita. Suunnitteluvaiheessa olisi pyrittävä kokoamaan testitapaukset, jotka tuottavat määrittelyjen vastaisia vasteita. Vaikeutena on keksiä testiaineisto, joka suhteellisen suurella todennäköisyydellä tuottaisi virheitä. Usein tämä valinta perustuu testaajien aiempaan kokemukseen, mutta tässä luvussa käsitellään järjestelmällisempiä tapoja ratkaista tämä ongelma.

Patton jakaa kirjassaan [Patton, luku 5] ohjelmiston kahteen osaan eli tietoon ja suoritettavaan ohjelmaan. Tiedoksi voidaan laskea muun muassa kaikki näppäimistöltä annetut syötteet, hiiren liikkeet ja ruudulla näkyvät vasteet. Ohjelmaksi taas katsotaan suoritettavissa oleva koodi, logiikka ja laskenta. Testaus jaetaan vastaavasti näiden suuntaviivojen mukaisesti. Luku 5.5.7 käsittelee tiedon testausta, kun taas ohjelman testaukseen perustuvaa menetelmää, tilakaaviotestausta, käsitellään luvussa 5.5.9.

Muita mustalaatikkotestauksen pyrkimyksiä on löytää käyttöliittymävirheet, virheet tietorakenteissa tai pääsyssä ulkoisiin tietorakenteisiin, toimintavirheet sekä käynnistys- ja lopetusvirheet. Näissä menetelmissä siis yleisesti keskitytään sovellukselle asetettuihin toiminnallisuusvaatimuksiin ja tämän testauksen tulee täydentää lasilaatikkotestausta.

5.5.2 Staattinen mustalaatikkotestaus tuotteen määrittelylle

Kuten lasilaatikkotestauksessa, myös mustalaatikkotestauksessa voidaan tehdä jako staattiseen ja dynaamiseen testaamiseen. Staattinen mustalaatikkotestaus kattaa kuitenkin vain kirjoitetun tuotteen määrittelyn testaamisen ja dynaamisella suoritetaan kaikkien muiden kehitettyjen osa-alueiden testauksen.

Testaamalla määrittelydokumentaatio vältetään monien virheiden välittyminen virheellisestä dokumentaatiosta koodausvaiheessa tuotteeseen. Koska ainoastaan määrittelydokumentin on oltava valmiina, voidaan staattinen mustalaatikkotestaus suorittaa jo projektin varhaisessa vaiheessa. Virheiden korjauskustannukset ovat alhaiset, koska ne eivät ihanteellisessa tapauksessa ole ehtineet vielä koodiin. Luku staattisesta mustalaatikkotestauksesta perustuu pääasiassa lähteeseen [Patton, luku 4].

Aluksi tuotteen määrittely testataan korkealta tasolta pyrkien hahmottamaan kokonaiskuva sekä havaitsemaan yleisiä ongelmia ja puutteita. Määrittelyä tulee tutkia ja testata asiakkaan näkökulmasta miettien asiakkaan ja käyttäjien odotuksia. Tässä vaiheessa kannattaa haastatella projektiin kuuluvia työntekijöitä ja mahdollisuuksien mukaan myös asiakasta ja käyttäjiä. Vaiheessa ei saa olettaa mitään eli jos jokin kohta tuntuu epäselvältä tai puutteelliselta, kyseessä saattaa olla virhe. Määrittelyä käytetään myös testitapauksia tehtäessä, joten vaihe toimii hyvin oppimisprosessina.

Nykyään sovellukset noudattavat yleisesti hyväksyttyjä sekä asiakkaan ja yritykselle määriteltyjä standardeja. Niitä liittyy muun muassa näyttöihin, teksteihin, komentoihin ja termeihin. Standardien tulee olla näkyvissä määrittelyssä, ja ne pitää myös testata. Määrittelystä standardit siirtyvät tuotteeseen, ja niiden käyttö on osa sovelluksen oikeellisuutta. Testaajan on hyvä hankkia käyttöönsä muita samantyylisiä ohjelmia sekä tutkia ja miettiä niiden pohjalta testauksen lähestymistapaa, sisältöä ja ilmeisiä ongelmakohtia.

Kokonaiskuvan hahmottamisen ja tutkimisen jälkeen tuotteen määrittely testataan alemmalta tasolta. Testaus kannattaa suorittaa käyttäen seuraavaa attribuuttilistaa:

1. Onko määrittely täydellinen vai puuttuuko jotain?
2. Onko määrittely yksiselitteinen ja johdonmukainen?
3. Ovatko esitetyt asiat oikein?
4. Määritelläänkö vain tarvittavat asiat, eikä yhtään liikaa?
5. Vaikuttaako sovellus toteuttamiskelpoiselta?
6. Eihän koodaustapaa määritellä?
7. Ovatko määritellyt ominaisuudet testattavissa?

Tuotteen määrittelystä on tutkittava **käytetty terminologia** ja huomioitava erityisesti tiettyjen sanojen käyttö, jotta virheelliset sanat eivät välity koodausvaiheessa käyttöliittymään. On tutkittava, määritelläänkö sanat tarkasti vai jätetäänkö ne moniselitteisiksi, jolloin niiden merkitys ei ole täsmällinen asiakkaalle, kehittäjille eikä testaajillekaan. Sanat *aina* ja *ei koskaan* pitävät joitain ominaisuuksia itsestäänselvyyksinä, joten on testattava niiden olevan aina päteviä. Testausvaiheessa ei ole mahdollista testata ominaisuuksia, joiden kerrotaan ilmenevän *joskus*. Ominaisuuksia ei voi määritellä sanoilla *hyvä* ja *nopea*, vaan määreiden on oltava laskettavia, testattavissa olevia. Listojen on oltava täydellisiä eivätkä ne saa sisältää termin *ja niin edelleen* tapaisia sanoja. Lisäksi tarkastelun arvoinen osa ovat *jos... niin...* -lauseet, joista puuttuu *muussa tapauksessa* -osa.

5.5.3 Testiaineiston valinnan ongelmia ja periaatteita

Mustalaatikkotestauksessa ensimmäinen vaihe on yleensä sovelluksen toiminnan testaus. Siinä tarkoituksena on löytää virheitä tai puutteita ohjelman määrittelyn ja sen todellisen toiminnan välillä.

Määrittelyjä analysoimalla on saatava tietoon sallitut arvot kaikkiin sovelluksen syötekenttiin. Yleensä testaus suoritetaan myös määrittelyalueen ulkopuolisilla syötteillä. Jos sovelluksen toiminnasta ei ole tietoa, olisi testattava kaikilla mahdollisilla syötteillä, jotta voitaisiin olla varmoja toiminnan oikeellisuudesta. Kaikkien mahdollisten syötekombinaatioiden testaaminen on kuitenkin käytännössä mahdotonta.

Kattavan testiaineiston valinta on vaikea ongelma. Valinta toteutetaan usein testaajan omien mielipiteiden ja kokemusten pohjalta, eikä systemaattista menetelmää käyttäen. Tämä korostuu erityisesti pienissä projekteissa, joissa ohjelmoijat itse testaavat ja valitsevat testiaineiston. Tällöin he huomaamattaan voivat valita koodiin sopivia aineistoja sekä lähes välttelevät valitsemasta vaikeita tai harvoin esiintyviä ongelmia tuottavia syötteitä testitapauksiinsa. Tällöin testaus kattaa vain osan sovelluksesta. Patton muistuttaa kirjassaan [Patton, luku 5] tällä päästävän lähinnä *test-to-pass* -tyyppiseen testaamiseen eli voidaan taata toiminta tavanomaisimmissa tapauksissa.

Sovelluksen perinpohjaisessa testaamisessa kaikkia mahdollisia syötemuunnelmia ei voida kokeilla, mikä asettaa testiaineiston valinnalle suuria vaatimuksia. Melzer esittää tutkielmassaan [Melzer] ongelman tarpeeksi kattavan aineiston testaamisen ja resurssien kuluttamisen välillä. Olisi koottava kaikista riskialtteimmat tapaukset ennakkotietojen ja olettamusten perusteella. Testitapausten määrää kannattaa vähentää arvoalueen pienentämisellä käyttäen esimerkiksi luvuissa 5.5.4 ja 5.5.5 esiteltäviä jakoa ekvivalenssiluokkiin tai raja-arvoanalyysiä.

5.5.4 Testiaineiston valinta ekvivalenssiluokkien avulla

Ensimmäinen mahdollinen menetelmä testiaineiston valinnassa on syöteavaruuden jako ekvivalenssiluokkiin (engl. *equivalence partitioning*). Yhteen ekvivalenssiluokkaan kuuluvat aina keskenään samantyyliset syötteet. Testauksessa käytetään ainakin yhtä yksilöä kustakin luokasta. Jos testi toimii yhdellä yksilöllä, oletetaan ohjelman toimivan kaikilla samaan luokkaan kuuluvilla. Tämän jälkeen käydään systemaattisesti läpi kaikki luokat. Ekvivalenssiluokkien avulla saadaan mahdollisten testiarvojen lukumäärä huomattavasti putoamaan, mutta määrittelyjen perusteella pitää osata jakaa aineisto luokkiin oikein.

Sommerville kertoo kirjassaan [Sommerville, luku 20] mahdollisia luokkia yleisesti olevan positiiviset luvut, negatiiviset luvut, kirjaimet ja muut merkit. Testidatan valinnassa on otettava mukaan myös hyväksyttyjen arvojen ulkopuolella olevia luokkia. Tällä tavoin voidaan testata, että ohjelma osaa käsitellä oikein virheellisiä arvoja, sekä saadaan jako oikeaan ja virheelliseen ekvivalenssiluokkaan. Esimerkiksi positiivisen kokonaisluvun syötteenään haluavan kentän testauksessa mahdollisia oikeaan ekvivalenssiluokkaan kuuluvia syötteitä ovat esimerkiksi luvut 0, 1, 2,..., kun taas virheelliseen ekvivalenssiluokkaan kuuluvat kaikki muut syötteet.

Testiaineiston valinnassa on olemassa tiettyjä **sääntöjä**, joita muun muassa Pressman määrittelee kirjassaan [Pressman, luku 16]. Jos sallitut arvot kuuluvat tietylle arvojen välille *[a, b]*, on valittava kolme luokkaa. Ensimmäiseen luokkaan kuuluvat sallitut arvot, toinen luokka muodostuu sallittuja arvoja pienemmistä ja kolmas luokka suuremmista arvoista. Taulukko 1 esittää muokattuna IPL:n artikkelissaan [IPL] kuvaaman esimerkin neliöjuuren testaamisessa käytetyistä luokista. Koska neliöjuuren sallittuja syötteitä ovat nolla ja sitä suuremmat luvut, muodostaa luokan *i* eli sallitut syötteet nolla ja sitä suuremmat arvot. Luokaksi *ii* on valittu nollaa pienemmät arvot, jotka tuottavat vasteena virheen. Lisäksi mukaan voidaan ottaa luokka *iii*, joka koostuu ei-numeerisista arvoista, joilla tulee testauksessa saada vasteeksi virhe. Tämä on yksinkertainen esimerkki, kun taas yleensä ohjelmistoissa ekvivalenssiluokkiin jako on huomattavasti monimutkaisempaa.

Syötteiden luokat		**Vasteiden luokat**	
i	>=0	**a**	>=0
ii	<0	**b**	Virhe
iii	Ei-numeerinen	**c**	Virhe

Taulukko 1. Ekvivalenssiluokat neliöjuurelle.

Huomioitaviin tapauksiin kuuluu testata ohjelmaa tyhjällä syötteellä, jos se on sallittua. Sommerville muistuttaa kirjassaan [Sommerville, luku 20.1] testaamaan myös tapaukset, joissa syötteenä annetaan yksi arvo ja lisäksi monta arvoa. Esimerkiksi jonojen oletetaan automaattisesti sisältävän useampia yksilöitä ja ohjelmoijat saattavat unohtaa yhden yksilön mittaisen jonon tarkastelun. Eri testeissä olisi myös käytettävä eripituisia ja eri määrän tietoa sisältäviä aineistoja, jottei virheellinen ohjelma toimi vahingossa oikein jollain tietyllä syötteellä. On myös tarjottava eri määrää syötteitä kuin sovelluksen pitäisi hyväksyä. Sommerville kehottaa myös vaihtamaan virheellisen aineiston paikkaa, jotta tiedetään ohjelman havaitsevan virheen sekä syötteen alusta, keskeltä että lopusta.

Sommerville kehottaa kirjassaan [Sommerville, luku 20.1] näidenkin testien jälkeen huomioimaan, että virheitä on saattanut jäädä jäljelle. Joskus virheellinen syöte aiheuttaa virheen, eikä lopuilla syötteillä enää jatketa testaamista. Tällöin muut virheellisiä arvoja aiheuttavat syötteet jäävät huomiotta. Joskus myös erikoinen monimutkainen syötekombinaatio voi olla virheen aiheuttajana. Haikala ja Märijärvi kirjoittavat kirjassaan [Haikala et al., luku 15], että olennaista on tarkastella testiaineistoa kriittisesti. Tällöin yhtään ekvivalenssiluokkaa ei toivottavasti jää tutkimatta ja tarvittavat yksiköt kustakin luokasta olisivat mukana testauksessa.

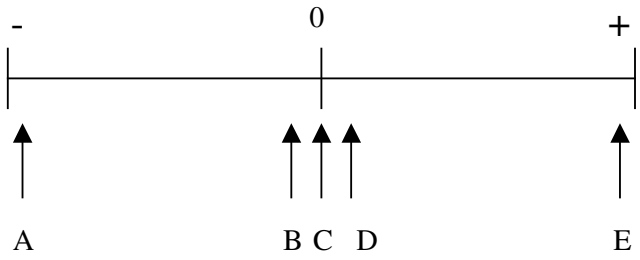
5.5.5 Raja-arvoanalyysi

Yleisesti suurin osa virheistä tapahtuu syötteiden arvoalueen rajoilla verrattuna alueen keskellä oleviin arvoihin. Toinen yleisesti käytetty menetelmä testiaineiston valinnassa onkin **raja-arvoanalyysi** (engl. BVA, *boundary value analysis*). Tässä valitaan tutkittavaksi syötteiden arvoalueiden raja-arvoja siten, että ne ovat mahdollisimman pieniä ja suuria arvoja. Lisäksi valitaan oletettavasti virheitä tuottavia arvoja, kuten nolla. Raja-arvoanalyysillä etsintä saadaan kohdistettua olennaisimpiin paikkoihin ja sillä saadaan myös testiaineiston määrää pienennettyä.

Jos testi epäonnistuu rajalla, se saattaa epäonnistua myös rajojen sisäpuolella olevilla arvoilla. Mutta testin onnistuessa rajalla se hyvin todennäköisesti onnistuu myös rajojen sisäpuolella. Tätä testausmenetelmää sovellettaessa hyvä sääntö on valita kustakin ekvivalenssiluokasta arvo luokan molemmilta rajoilta sekä keskeltä. Sommerville kirjoittaa kirjassaan [Sommerville, luku 20], että yleisesti ohjelmoijat käyttävät yksikkötestaamisessaan tavanomaisimpia arvoja eli aineiston keskivaiheille sijoittuvia arvoja. Tunnettu tosiasia kuitenkin on, että ohjelman virheet yleisesti tapahtuvat lähellä luokan rajaa sijaitsevilla arvoilla.

Pressman kirjoittaa kirjassaan [Pressman, luku 16] raja-arvoanalyysistä menetelmänä tulevan vielä parempi, kun testattaessa käytetään arvoja rajojen molemmilta puolilta. Siis arvoalueen ollessa suljettu väli *[a, b]*, tulee testaukseen ottaa arvot *a* ja *b* sekä arvot näiden molempien ala- ja yläpuolelta. Sen lisäksi, että syötteet valitaan arvoalueiden rajoilta, niin testauksessa tulee pyrkiä tuottamaan myös tuloksena saatavat vasteet niiden arvoalueiden rajoilta. Testausta suunniteltaessa on myös huomattava, että tutkimalla sovellusta tarkkaan sen kaikista kohdista, myös rajoja löytyy lisää ja siten todennäköisyys monien virheiden löytämiselle suurenee.

IPL esittää artikkelissa [IPL] esimerkin raja-arvoanalyysin arvoista. Tämä esimerkki neliöjuuren testaamisesta on esitetty muokattuna kuvassa 11. Ensimmäisenä syötteenä A annetaan suuri negatiivinen arvo, jonka tulee tuottaa vasteenaan ilmoitus laittomasta syötteestä. Toisena on testattava arvo syötteellä nolla, joka kuvassa on esitetty pisteenä C. Koska tämä on raja-arvo, on testattava myös juuri sen viereiset negatiivinen ja positiivinen arvo B ja D. Näiden lisäksi testataan mahdollisimman suurella positiivisella arvolla E, jonka tulee tuottaa vasteenaan sallittu oikea arvo.



Kuva 11. Esimerkki raja-arvoanalyysistä neliöjuuren testauksessa [IPL].

5.5.6 Muita testiaineiston valinnassa käytettäviä menetelmiä

Kautto esittelee opinnäytetyössään [Kautto] kolmantena testiaineiston valinnan menetelmänä **virheenarvausmenetelmän**, jossa käytetään testaajan kokemuksia aiemmista virheistä. Nämä eivät välttämättä sisälly ekvivalenssiluokkiin, eivätkä raja-arvoihin, mutta tuottavat helposti virheitä. Näitä voivat olla esimerkiksi arvo nolla sekä arvot, joita on useampia testiaineistossa.

Patton mainitsee kirjassaan [Patton, luku 5] arvausmenetelmän olevan mahdollinen testaajien kokemuksen karttuessa. Usein testaajien ollessa uusia projektissa, tämä menetelmä ei sovellu käytettäväksi. Muutenkaan tämä ei sovi ainoaksi menetelmäksi, mutta se on käyttökelpoinen muiden menetelmien tukena.

Mahdollinen menetelmä on myös **satunnaisesti valittu data**. Tämä ei ole luotettava menetelmä, eikä tätä tule käyttää kuin tapauksissa, joissa alustava testaus on käynnistettävä nopeasti ennen varsinaisia testejä. Etuna satunnaisuudessa voidaan pitää sitä, että testiaineiston valinnan väärät oletukset eivät heijastu testidatan valintaan.

5.5.7 Tiedon testaus

Patton määrittelee kirjassaan [Patton, luku 5], että tiedon testauksessa on seurattava, että käyttäjän antama tieto, sovelluksen tuottamat tulokset ja väliaikaiset sovelluksen sisäiset tiedot käsitellään oikein. Kaikkiaan tietoa on paljon, eikä täydellistä yksityiskohtaista testausta voida suorittaa. Tiedon määrää onkin rajoitettava jakamalla tietoalueet ekvivalenssiluokiksi ja käyttämällä sen jälkeen raja-arvoanalyysiä apuna.

Tarvittavien luokkiinjakojen jälkeen päästään testaamaan sovelluksen kykyä käsitellä syötteenä annetun tiedon erikoistapauksia. Patton kirjoittaa kirjassaan [Patton, luku 5], että käytännössä testattavia syötteitä ovat nollat ja tyhjät arvot. Esimerkiksi tekstikenttien toiminta on hyvä testata jätettäessä kenttä tyhjäksi. Joissain tapauksissa kenttä saattaa hyväksyä tyhjän arvon, mutta sovellus ei osaakaan käsitellä tietoa, jolloin päädytään virhetilanteeseen. Kun kentät on määrätty pakollisiksi, olisi käyttäjän saatava kunnollinen huomautus arvon puuttumisesta. Tyhjään kenttään verrattavia arvoja ovat nolla, null, oletusarvo ja tyhjä. Näitä kaikkia kannattaa kokeilla. Tosin nämä syötteet kannattaisi automaattisesti liittää erikoistapauksiksi ekvivalenssiluokkien joukkoon.

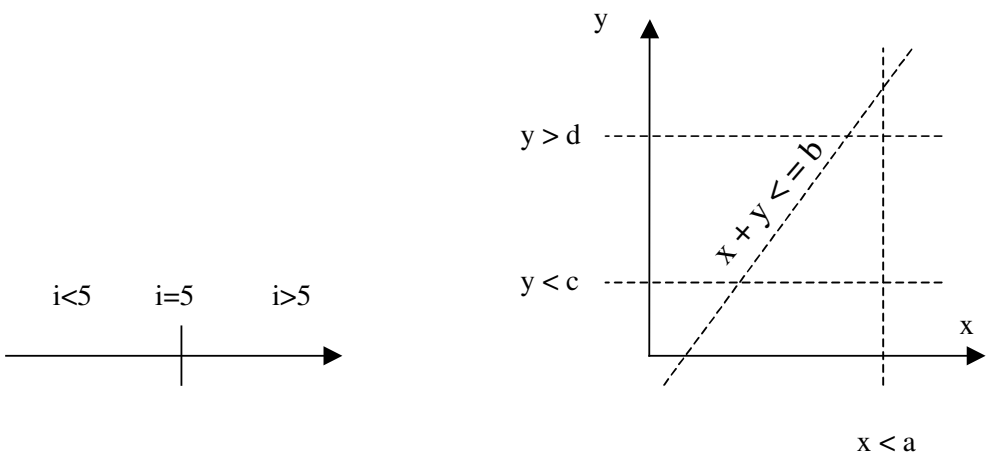
Patton muistuttaa kirjassaan [Patton, luku 5] huomioimaan aina myös testauksen, jolla pyritään testien epäonnistumiseen. Tällöin mahdollinen testaustapa on muun muassa kaikenlaisen epäkelvon tiedon syöttäminen. Tällä testauksella pyritään ainoastaan testin epäonnistumiseen, kun taas aiemmilla testitapauksilla on periaatteessa osoitettu sovelluksen toimivan. Tämä on olennaista, koska sovelluksen käyttäjä syöttää myös epäkelpoja arvoja vahingossa ja tarkoituksellisesti. Ohjelmoija sen sijaan yleensä välttelee tällaista kokeilua alitajuisesti tai tarkoituksellisesti.

5.5.8 Aluetestaus

Melzer esittelee kirjoituksessaan [Melzer] vielä erään sovelluksen tietoon liittyvän testin, aluetestauksen (engl. *domain testing*). Siinä keskitytään testaamaan alueella olevia erikoispisteitä. Tämä on jokseenkin päällekkäinen aiemmin esitetyn raja-arvoanalyysin kanssa, mutta se on esitetty erillisenä menetelmänä lähteissä [Melzer] ja [Koikkainen]. Sovelluksen kaikkien muuttujien arvoja tutkitaan ja pyrkimys on osoittaa niiden olevan sallitun alueen ulkopuolella. Aluetestauksen yhteydessä on tarkasteltava myös, että ohjelma hyväksyy syötteiksi vain sallittuja arvoja, sillä yleensä ohjelma tuottaa järjettömiä tuloksia kiellettyjä arvoja syötettäessä.

Aluetestauksessa olisikin aina testattava muun muassa, onko kaikki muuttujat määritelty oikein, taulut oikeissa rajoissa ja vertaillaanko vain keskenään samantyyppisiä arvoja sekä onko loogisten operaatioiden prioriteetit oikeat. Näiden testien suorittaminen vähentää huomattavasti muun testauksen aikana löydettyjen virheiden määrää.

Koikkalainen esittää luentomonisteessaan [Koikkalainen] aluetestauksen kulkua. Siinä ensin etsitään sovelluksessa olevat arvoalueet ja niiden rajat, sitten valitaan yksi testitapaus arvoalueen sisältä, ja testi alueen ulkopuolelta. Kuva 12 esittää kaksi esimerkkiä mahdollisesta aluetestauksesta.



Kuva 12. Kaksi esimerkkiä aluetestauksesta [Koikkalainen].

Kuvassa vasemmalla puolella on aluetestauksessa tyypillinen tapaus, jossa alue määräytyy ehtolauseen mukaisesti. Tämä liittyy kyseisessä sovelluksessa tapaukseen, jossa ehtolausetta toistetaan, kunnes arvo on viisi. Tällöin testattavaksi jää kolme aluetta eli viittä pienemmät, viittä suuremmat ja tapaus arvon ollessa tasan viisi. Oikealla puolella oleva kuva esittää huomattavasti vaativampaa tapausta, jossa muuttujien määrä on kasvanut niin suureksi, ettei täydellinen kriittisten kohtien läpikäynti enää onnistu.

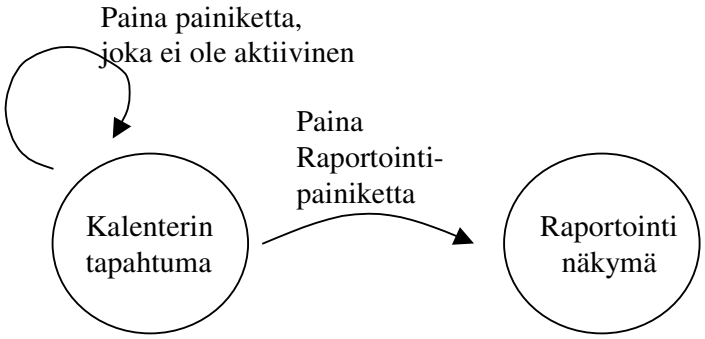
5.5.9 Tilakaaviotestaus

Edelliset luvut käsittelivät sovelluksessa olevan tiedon oikeellisuuden testaamista. Toinen lähestymistapa on tarkastella sovelluksen tiloja, johon käyttökelpoinen funktionaalisen testauksen menetelmä on tilakaaviotestaus (engl. *state test*). Tämä sopii varsinkin sovelluksille, joiden vaatimukset on alunperinkin määritelty tilakaavioina. Testauksessa tutkitaan sovellusta sen ollessa eri tiloissa sekä siirtymistä näiden tilojen välillä. Luku perustuu pääosin lähteeseen [Patton, luku 5].

Ensinnäkin sovellus on testattava sen ollessa tietyssä tilassa, jolloin tarkastellaan, ovatko kaikki osiot oikein. Toisaalta on testattava siirtyminen tilojen välillä ja tarkkailtava, tapahtuuko siirtyminen oikein ja muuttuvatko tilat määrittelyn mukaisesti. Testitapauksilla testataan siirtymät tilojen välillä luomalla tapahtumia, jotka johtavat näihin siirtymiin.

Ensimmäinen tehtävä Pattonin mukaan on **tilasiirtymäkaavion** (engl. *state transition map*) luominen. Ensinnäkin kaaviossa on aina esiteltävä jokainen itsenäinen tila, jossa sovellus voi olla. Tilojen välille on kuvattava siirtymät ja siirtymien yhteyteen on kirjoitettava syöte tai tilanne, jolla tilojen siirtymä tapahtuu. Tällainen siirtymän aiheuttaja voi olla esimerkiksi painikkeen valitseminen. Lisäksi kaaviossa on kuvattava, mitä tapahtuu tilaan tultaessa. Tällainen tapahtuma voi olla esimerkiksi uuden ikkunan aukeaminen.

Kuva 13 esittää yhden esimerkin tilasiirtymäkaaviosta. Tässä sovellus on aluksi tilassa, jossa se esittää kalenterin tapahtuman. Painettaessa painiketta, joka ei ole aktiivinen, tila pysyy samana. Painettaessa *Raportointi*-painiketta sovellus siirtyy raportointitilaan. Tämä esimerkki on yksinkertainen, kun taas yleensä kaaviot ovat varsin isoja.



Kuva 13. Tilasiirtymäkaavio.

Ohjelman tiloja on pienissäkin sovelluksissa lähes ääretön määrä, eikä niitä kaikkia voida käydä läpi. Luvussa 5.5.7 käsitellyssä aineiston testaamisessa testitapausten määrää pyritään vähentämään ekvivalenssiluokkien avulla. Myös ohjelman tiloja on vähennettävä käyttäen vastaavia tunnettuja menetelmiä. Lisäksi on pyrittävä vähentämään riskejä, jotka johtuvat siitä, että jätetään osa tiloista ja poluista testaamatta.

Ensinnäkin pitää testata pääsy jokaiseen tilaan. Tässä testauksen ensimmäisessä vaiheessa reitti tilaan pääsemiseksi ei ole olennainen, vaan ainoastaan **jokaisessa tilassa käynti**. Kaikkiin tiloihin tulee olla pääsy sovelluksessa, koska muutoin niitä ei tarvita. Toiseksi on testattava kaikki **tavanomaisimmat polut**, koska ne ovat käytetyimpiä ja niiden varmasti halutaan toimivan käytännössä. Kolmantena testauksessa on huomioitava **vähiten tavanomaisimmat polut**, jotka ovat todennäköisimmin jääneet kehittäjiltä testaamatta. Usein testaaja on niiden ensimmäinen käyttäjä, joten ne saattavat sisältää monia virheitä.

Neljäs olennainen testaus ovat **virhetilanteet**. Virhetilanteiden luonti on välillä varsin monimutkaista. Kehittäjät koodaavat virheiden käsittelyn, mutta eivät usein testaa sitä. Tämä saattaa johtaa siihen, että virhetilanteissa saadaan vääriä virheilmoituksia tai sovellus ei palaudu haluttuun tilaan virheelliseen tilaan jouduttuaan. Viimeisenä olennaisena asiana on **satunnainen testaus** siirtymällä monesta eri tilasta eri puolelle sovellusta.

Näiden positiivisten testausten lisäksi on hyvä suorittaa **negatiivisia testitapauksia**, joilla pyritään testien epäonnistumiseen. Ensinnäkin käytetään laittomia tiloja ja siirtymiä näiden välillä. Eräs olennainen testaus on monien tehtävien suorittaminen yhtäaikaisesti. Joskus näillä tapahtumilla saa sovelluksen tilaan, jossa ei tiedetä, kumpi tehtävistä tulisi suorittaa ensin ja molemmat epäonnistuvat. Tätä voi kokeilla vaikka tulostamalla suuren haun aikana tai painamalla monia painikkeita lähes samanaikaisesti. Muita negatiivisiksi luokiteltavia testejä ovat rasitustestaukset, joita käsitellään testausstrategioiden osana luvussa 5.6.

5.5.10 Muita mustalaatikkotestausmenetelmiä

Beizer esittää kirjassaan [Beizer, luku 3] muita systemaattisia mustalaatikkomenetelmiä, joita ovat tapahtumatestaus (engl. *transaction-flow testing*), syntaksitestaus (engl. *syntax testing*) ja logiikkatestaus (engl. *logic testing*). Näistä testausmenetelmistä kutakin esitellään tämän luvun seuraavissa kappaleissa.

Ensimmäinen näistä Beizerin esittelemistä menetelmistä on **tapahtumatestaus**, joka on toiminnallinen testaus ja siinä toimitaan käyttäjän näkökulmasta. Rakenteellisen testauksen menetelmä, polkutestaus (katso luku 5.4.5), on varsin samankaltainen tekniikoiltaan ja menetelmiltään. Yksi tapahtuma on käyttäjän näkökulmasta nähtynä yksi toiminta, joka yleensä koostuu syötteen antamisesta, ohjelman sisäisestä syötteen käsittelystä ja sen jälkeisestä vasteen saannista. Järjestelmän puolesta nähtynä tämä saattaa koostua noin kymmenestä askeleesta, mutta käyttäjän näkökulmasta kyseessä on vain yksi transaktio.

Tapahtumatestauksessa pitää ensin tehdä tapahtumakaaviot, sitten testata polkutestauksen tavoin kaikki prosessit. Tosin tässä korkean tason testauksessa polkujen valinta eroaa oleellisesti polkutestauksesta. Ensinnäkin kannattaa valita kattava joukko peruspoluista, mutta niiden lisäksi myös muutama mahdollisimman pitkä ja mutkikas sovelluksen alusta loppuun kulkeva polku. Tällaisilla poluilla voidaan jopa hyvin suunnitelluissa sovelluksissa paljastaa puuttuvia kytkentöjä, kaksinkertaisia kytkentöjä ja rajapintaongelmia. Tällä tavoin luodaan ajoissa paljon vaikeuksia, jotka muuten huomattaisiin vasta hyväksymistestausvaiheessa.

Syntaksitestausta voidaan soveltaa suhteellisen vähän testauksen korkeammilla tasoilla. Beizer kirjoittaa, että tällä tasolla syntaksitestauksen pitäisi olla kohdistunutta syntaksin osiin, joissa käsitellään elementtien vuorovaikutusta. Elementin syntaksi on testattu yksikkötestaustasolla, mutta nyt pitäisi vielä testata toiminnan logiikka. Vaikkakin tämä testaus on toiminnallista, sen tuottamat virheet on tutkittava ohjelman rakennetta tarkastellen.

Kuten yksikkötason testauksessa, myös korkeamman tason **logiikkatestauksessa** päätöstauluja ja Boolen algebraa käytetään päättelyissä. Beizer huomauttaa, että logiikka tässä voi olla epäsuorempaa johtuen systeemin toiminnallisuuksien näkymisestä ja vaikutuksesta koko testaukseen. Toiminnallisuuksien vaikutukset pitäisikin pyrkiä poistamaan testauksen tieltä ja keskittyä ainoastaan lopulta saataviin totuusarvoihin. Korkean tason logiikkatestaus tulee suhteellisen helpoksi, kun keskitytään kaikkialla vain arvoihin ”tosi” ja ”epätosi”.

5.5.11 Luova virheiden etsintä

Systemaattisilla testeillä ei välttämättä löydetä kaikkia havaittavissa olevia virheitä. Tunnettu tosiasiahan on, että sovelluksiin aina jää virheitä, vaikka testaus olisi kuinka kattavaa. Systemaattisesti suoritetun testauksen lopuksi kannattaakin olla luova virheiden etsinnässä. Käytännön kokemuksesta testaajat tietävätkin, että testitapauksilla ei havaita kaikkia virheitä, vaan aina tarvitaan niiden ulkopuolisia testauksia. Testauksen aikana testaaja tutustuu lisää sovellukseen ja tulisi kokeilla toiminnallisuuksia vielä uusilla tavoilla.

Mahdollisina muina käytettävinä menetelminä Patton mainitsee kirjassaan [Patton, luku 5] **käyttäytymisen ns. tyhmän käyttäjän tapaan**. Uudet ja sovellusta tuntemattomat käyttäjät tekevät sovelluksella kaikkea mahdollista yrittäessään käyttää sitä. He eivät tunne sääntöjä, eivätkä pääsääntöisesti lue käyttöohjeita, joten heillä ei ole mitään ennakko-oletuksia. He tulevat painamaan kaikkia mahdollisia painikkeita, syöttämään ennalta arvaamatonta ja pituudeltaan vaihtelevaa tekstiä sekä kulkemaan tilasta toiseen satunnaisesti. Sovellus ei saisi kaatua ja sen olisi muutenkin toimittava näissä tilanteissa järkevästi käyttäjää kohtaan.

Patton kehottaa myös etsimään virheitä sieltä, missä niitä on aiemminkin ollut, sillä **virheillä on usein tapana kasaantua**. Jos tietyssä osassa ohjelmaa monesta ominaisuudesta löytyy virheitä, kannattaa testata myös loput ominaisuudet kunnolla. Tavanomaisesti kehittäjien tapana on korjata vain tietty raportoitu virhe. Jos siis virheraportti ilmoittaa sovelluksen kaatuvan syötettäessä pitkä tieto yhteen kenttään, kehittäjät korjaavat tämän. Automaattisesti kehittäjät eivät ehkä kuitenkaan korjaa toimintaa, jossa kahteen kenttään syötetään pitkä tieto. Tällöin sovellus saattaa kaatua, vaikka asiaa ei ollut esitetty virheraportissa.

Lisäksi **testauskokemus** tuo tietoa siitä, mistä virheitä kannattaa etsiä. Tämä tarkoittaa yleistä oppimista kaikista sovelluksista. Tietysti myös juuri testattavana olevasta sovelluksesta oppii havaitsemaan paikkoja, joista virheitä ehkä vielä useankin testauskierroksen jälkeen saattaa löytyä. Kuitenkaan tätä tekniikkaa ei opi muuten kuin harjoituksen kautta ja ajan kuluessa.

5.5.12 Hyötyjä ja haittoja

Melzer kuvaa tutkielmassaan [Melzer] useita mustalaatikkotestauksen etuja ja haittoja. Eräs etu on se, että testit ovat helposti uusittavissa. Samalla, kun testataan sovellus, tulee ympäristökin testattua. Lisäksi samaa testaustapaa voidaan käyttää monta kertaa uudelleen.

Tämän testauksen haittapuolina Melzer näkee sen, että tulokset voivat olla yliarvioituja, eikä kaikkia sovelluksen ominaisuuksia voida testata mustalaatikkotestauksella. Lisäksi, koska tässä ei testata sovelluksen sisäistä toimintaa, ei syy virheille välttämättä löydy, vaan niiden etsintä jää ohjelmoijien vastuulle. Toisaalta testaamisessa ei tarvita lähdekoodia. Tämä on yleensä hyvä puoli, koska koodia ei kaikissa tapauksissa ole saatavilla.

Hyvänä puolena voidaan pitää sitä, että mustalaatikkotestaamista voidaan soveltaa testauksen kaikissa vaiheissa. Tosin yksikkötestauksessa tämä menetelmä ei ole paras mahdollinen, mutta se on hyvin sovellettavissa integraatio-, systeemi- ja hyväksymistestauksessa. Valitettavan usein mustalaatikkotestaus jää projekteissa loppupuolelle, joten sen aikana löydetyt virheet voivat aiheuttaa huomattavia lisäkustannuksia ja töitä niin korjaajille, testaajille kuin dokumentaatioiden ylläpitäjille.

5.6 Testausstrategioita

Testausmenetelmät jaetaan musta- ja lasilaatikkomenetelmiin. Näiden lisäksi on olemassa testausstrategioita, jotka eivät suoranaisesti kuulu kumpaankaan näistä. Testausstrategioita saatetaan käyttää monessa eri testausvaiheessa ja ne ovat yleisiä joihinkin menetelmiin liittyviä osia. Koska ne eivät kuulu minnekään varsinaisesti, ne esitellään tässä omana lukunaan.

5.6.1 Rasitustestaus

Rasitustestauksessa (engl. *Stress testing*) tarkoituksena on testata järjestelmän suorituskykyä ja toimintavarmuutta. Tarkoituksena on näyttää, että sovellus ei pysty käsittelemään suuria määriä tietoa, vaikka sen määrittelyjen mukaisesti pitäisi pystyä tähän.

Rasitustestaus mielletään joskus systeemitestauksen osaksi ja yhdeksi mustalaatikkotestauksen menetelmäksi. Sommervillen [Sommerville, sivut 457-458] mukaan tämä vaihe voidaan aloittaa, kun järjestelmä on integroitu ja integroinnin tiedetään toimivan tarkoituksellisella tavalla. Sommerville kehottaa käyttämään rasitustestausta myös rajapintojen testauksessa integraatiotestauksen aikana.

Melzer jakaa tutkielmassaan [Melzer] rasitustestit **kahteen erilaiseen luokkaan**. Ensimmäisessä järjestelmää kuormitetaan pitkäaikaisesti suurella määrällä tietoa, esimerkiksi muutaman päivän ajan. Toinen on rasitustestausta alkuperäisellä tavallaan eli järjestelmää kuormitetaan lyhyellä aikavälillä huippumäärällä tietoa. Jos sovelluksen on määrittelyjen mukaisesti toimittava 50 yhtäaikaisella käyttäjällä, niin sen testaus tulee suorittaa tapauksella, jossa vähintään tämä maksimimäärä käyttäjiä suorittaa toimintoja samanaikaisesti. Melzer muistuttaa, että tietysti sovellus tulisi testata myös muun muassa 51 käyttäjällä tarkistaen vasteaika ja toimivuus pienellä ylikuormituksella. Joissain sovelluksissa ylikuormitustilanteita tapahtuu ja niiden käsittelemättä jättäminen yleensä aiheuttaa vaikeuksia.

Rasitustestausta voidaan jatkaa kuormittamalla systeemiä aina vain enemmän, kunnes **järjestelmä kaatuu**. Sommerville esittelee kirjassaan [Sommerville, sivut 457-458] tällaisen testauksen kaksi päämäärää. Ensinnäkin saadaan selville järjestelmän toiminta vikatilanteissa. Rasitustestauksessa yllättävän suuri määrä tietoa voi olla ainoana syynä vikatilanteeseen. Johtuipa vikatilanne mistä syystä tahansa, olisi kaatuminen selvitettävä. Sen aikana tieto ei saa kadota tai tuhoutua, eikä käyttäjä saa joutua liian yllättäviin tilanteisiin. Ylikuormituksen tapahtuessa järjestelmän tulisi käyttäytyä käyttäjän huomioivalla tavalla, eikä kaatua odottamattomasti käyttäjää informoimatta.

Toinen päämäärä järjestelmän kaatamiseen tähtäävässä rasitustestauksessa Sommervillen mukaan on tutkia systeemin **rasituksesta johtuvia yllättäviä vikatilanteita**. Rasitustestauksessa vikatilanne yleensä aiheutuu liian suuren syötemäärän takia. Mutta syötettäessä paljon tietoa saattaa syötekombinaatioista löytyä tilanne, jolla systeemi kaatuu jo sallitulla määrällä syötteitä. Näiden tilanteiden esiintyminen on harvinaista, mutta nämäkin on hyvä tutkia.

Sommervillen mukaan rasitustestaus on todella olennaista hajautetuissa järjestelmissä. Näissä kuormitus aiheuttaa verkon tukkeutumista, jolloin prosessit tulevat hitaammiksi niiden odottaessa vaadittua tietoa muilta prosesseilta.

Patton määrittelee kirjassaan [Patton, luku 5] rasitustestaukseksi lähes päinvastaisen menetelmän eli **sovelluksen ajamisen heikoissa olosuhteissa** ja pitää yllä kuvattua rasitustestiä **kuormitustestinä** (engl. *load testing*). Pattonin mukaan rasitustestauksessa sovellusta testataan mahdollisimman vähäisellä muistilla, hitaalla prosessorilla ja niin edelleen. Kaikki ulkoiset ominaisuudet pyritään vähentämään niiden vähimmäisarvoihin.

Lisäksi eräs rasitustestien joukkoon kuuluva testi on **toistotesti** (engl. *repetition test*). Tässä sama yksittäinen askel toistetaan aina uudelleen. Tämä voi olla kalenterisovelluksessa päivämäärien selaaminen eteenpäin. Patton huomauttaa kirjassaan [Patton, luku 5], että tässä virhe saattaa löytyä jo muutamien tai vasta satojen toistojen jälkeen. Erityisesti tällä voidaan havaita mahdollisia muistivuotoja eli sovellus jättää vapauttamatta kuormittamansa muistin. Tällaista on syytä epäillä ohjelmissa, jotka toimivat aluksi hyvin, mutta alkavat vähitellen hidastua. Tällaisten operaatioiden toistaminen saattaa olla mahdollista vain testaustyökalujen avulla.

5.6.2 Käyttöliittymätestaus

Käyttöliittymätestaus käsittää kaikki osa-alueet, jotka sovelluksessa ovat käyttäjän ulottuvilla. Käyttöliittymän testaus on olennainen osa sekä systeemi- että hyväksymistestauksessa. Näissä ei ole käytössä lähdekoodia, joten testaus on toteutettava mustalaatikkotestauksena. Testaus toteutetaan periaatteessa yksinkertaisesti käyttämällä sovellusta ja tarkkailemalla sitä käyttäjän näkökulmasta.

Uudelleenkäytettävät komponentit ovat nykyään yleisiä, mikä tekee käyttöliittymien kehittämisestä suhteellisen nopeaa ja helppoa. Toisaalta samanaikaisesti kasvava käyttöliittymien mutkikkuus aiheuttaa lisää haasteita testauksen suunnittelulle ja suorittamiselle. Nykyään käyttöliittymillä on usein samantapainen ulkoasu, joka mahdollistaa tiettyjen yleisien testien kehittämisen. Eräs yksityiskohtainen listaus käyttöliittymätestauksen aikana tarkastettavista asioista on esitetty Pressmanin kirjassa [Pressman, sivut 493-494].

Käyttöliittymävirheet ovat usein varsin subjektiivisia, kustakin käyttäjästä riippuvia. Patton listaa kirjassaan [Patton, luku 11] **seitsemän olennaista käyttöliittymän piirrettä**, jotka liittyvät jokaiseen käyttöliittymään riippumatta sovelluksen käyttötarkoituksesta. Nämä ovat standardien noudattaminen, joustavuus, oikeellisuus, intuitiivisuus, käyttömukavuus, hyödyllisyys ja johdonmukaisuus. Joissain lähteissä tämä lista esitetään hiukan erinäköisenä. Kuitenkin näihin piirteisiin perustuen testaus voidaan suorittaa. Jos käyttöliittymä on näiden vastainen, on siinä virhe.

Ensimmäinen ja varsinaisesti tärkein tarkkailtava asia käyttöliittymästä on **standardien noudattaminen**. Kun sovellus on toteutettu tietylle ympäristölle, on noudatettava sille asetettuja ohjeita. Tällöin esimerkiksi kaikki Windows-ympäristöön toteutetut sovellukset olisivat suhteellisen yhtenäisiä ja helppoja käyttää, koska jo aiempien sovellusten osaaminen takaisi osaamisen myös uudessa sovelluksessa.

Sovelluksen **intuitiivisuutta** voi testata seuraavia kysymyksiä käyttäen. Onko toiminnallisuus löydettävistä käyttäjän olettamasta paikasta? Onko käyttöliittymä organisoitu järkevästi eli pääseekö paikasta toiseen liikkumaan helposti? Onko tietoa ja toiminnallisuuksia liikaa? Kun käyttäjä ei osaa toimia tietyssä tilanteessa, saako hän tarvitun tiedon käyttöohjeista?

Johdonmukaisuuden tulisi näkyä koko sovelluksen läpi. Tämän piirteen testauksessa olisi erityisesti huomioitava pikapainikkeet, valikkovalinnat, terminologia ja painikkeiden sijoittelu. Patton muistuttaa huomioimaan, pidetäänkö käyttäjää kaikkialla samantasoisena eli esimerkiksi lasten peleissä myös huomioviestit ovat lasten ymmärrettävissä.

Sovellusten tulisi olla myös **joustavia**, jolloin käyttäjä voisi muokata niitä käyttääkseen vain tiettyä osaa funktioista tai toisaalta saadakseen kaikki mahdolliset ominaisuudet käyttöönsä. Tämä liittyy myös siihen, kuinka paljon ohjausta käyttäjä haluaa sovellukselta. Vasta-alkaja tarvitsee paljon neuvoja, kun taas kokenut käyttäjä haluaa suorittaa saman toiminnon käymättä läpi jokaista erillistä vaihetta ja ohjeikkunaa.

Piirteiden joukossa on myös **käyttömukavuus**. Sovelluksen on oltava tarkoituksenmukainen, käsiteltävä virhetilanteet järkevästi ja varoitettava käyttäjää ennen kriittisten operaatioiden suorittamista. Mukavuuteen vaikuttaa myös suorituskyky. Kun toiminnon suoritus kestää kauan, on käyttäjälle ilmoitettava asiasta vaikkapa tilarivin avulla.

Kuudes Pattonin luokittelema ominaisuus on **oikeellisuus**. Sovellusta ei enää verrata sen määrittelyyn, vaan markkinointimateriaaliin ja käyttöohjeeseen. Tekstin on oltava oikein kirjoitettua ja sujuvaa. Ulkoasun on oltava yhtenevä, joka näkyy esimerkiksi painikkeiden kokona ja sijoitteluna. Lisäksi on huomioitava, että sovellus todella tekee sen, mitä se ilmoittaa tekevänsä.

Viimeisenä ominaisuutena on **hyödyllisyys**. Tällä ei tässä tarkoiteta sovelluksen hyödyllisyyttä kokonaisuutena, vaan yksittäisen sovelluksen osan tai piirteen hyödyllisyyttä käyttäjälle.

5.6.3 Vertailutestaus

Vertailutestauksessa (engl. *back-to-back testing, comparison testing*) verrataan ohjelman eri versioiden toimintaa keskenään. Koikkalainen mainitsee [Koikkalainen, luku 12] tämän eroavan muista testauksen menetelmistä siinä, että määrittelyjen oikeaa vastaavuutta ei välttämättä tarvita. Tietysti oikeasta vastaavuudesta voi tässäkin olla hyötyä.

Vertailutestauksessa käytetään useaa erilaista toteutusta ja tuloksia verrataan toisiinsa. Koikkalainen toteaa, että nämä eri toteutukset voivat olla sovelluksen järjestelmän osia tai prototyyppejä. Erityisesti kriittisissä sovelluksissa saatetaan toteuttaa kaksi eri toteutusta toisistaan erillään, vaikka ei alunperinkään ole tarkoitus käyttää kuin toista sovelluksista. Testaus suoritetaan kummallekin versiolle erikseen ja jälkikäteen verrataan niiden toimintaa ja tuloksia. Erityistä luotettavuutta vaativissa sovelluksissa tämä voi olla välttämätön testaus.

Vertailutestauksessa erilliset kehitysryhmät toteuttavat itsenäiset versiot sovelluksesta käyttäen samaa tuotteen määrittelyä. Pressman mainitsee kirjassaan [Pressman, sivut 492-493], että tällaisessa tapauksessa jokainen versioista voidaan testata samaa testiaineistoa käyttäen ja verrata vasteiden samanlaisuutta. Kaikille versioille annetaan sama syöte ja oletetaan kaikkien toteutuksien olevan oikeita, jos vaste on sama kaikissa. Jos taas vasteissa on eroja, sovelluksien vertailun tarkoituksena on havaita, johtuvatko vasteiden erot sovelluksen virheistä.

Vertailutestauksella on huonot puolensa. Pressman kirjoittaa, että jos määritellyissä vaatimuksissa on virhe, se väistämättömästi heijastuu molempiin vertailun sovelluksiin. Jos tämän jälkeen molemmat toimivat samalla tavoin väärin, ei vertailutestauksella voida löytää virhettä.

5.6.4 Mutaatiotestaus ja virheiden kylväminen

Mutaatiotestauksessa (engl. *mutation testing*, *error validation*) sovellukseen tahallisesti lisätään virheitä. Koikkalainen kirjoittaa luentomonisteessaan [Koikkalainen], että tämän jälkeen tulee suorittaa testaus ja verrata, kuinka monta prosenttia tahallisesti aiheutetuista virheistä löytyi. Tätä prosenttiosuutta voidaan pitää suuntaa-antavana mittana kaikkien sovelluksessa olevien virheiden löytymisen onnistumiseen. Tätä testausta voidaan periaatteessa soveltaa kaikissa testauksen vaiheissa.

Haikala ja Märijärvi määrittelevät kirjassaan [Haikala et al., sivu 278] vastaavan testausmenetelmän, vaikkakin eri nimellä eli virheiden kylvämisenä (engl. *error seeding*). Tässä menetelmässä sovellukseen lisätään tahallisesti virheitä, kun taas mutaatiotestauksessa Haikalan ja Märijärven mukaan virheitä kylvetään sovelluksen eri versioihin.

Haikala ja Märijärvi määrittelevät virheiden kylvämisen tekniikkana seuraavasti. Alussa tehdään oletus, että virheiden määrä sovelluksessa on *X*. Tämän jälkeen sovellukseen kylvetään *Y* uutta virhettä. Testauksessa havaitaan *X'* sovelluksessa oikeasti ollutta virhettä ja *Y'* kylvetyistä virheistä. Tässä oletetaan, että molempien ryhmien virheitä on löydetty yhtä suuri osuus kaikista virheistä. Näin voidaan arvioida todellisen virheiden lukumäärän olevan: $X = X' * Y / Y'$.

Mutaatiotestausta tai muita menetelmiä, joissa tahallisesti lisätään virheitä, Haikala ja Märijärvi eivät yleisesti pidä kovin hyvinä menetelminä. Ne kuitenkin tuottavat lisätyötä, mikä ei yleisesti kiireisissä projektiaikatauluissa ole suotavaa. Toisaalta on olemassa riski, että jokin tahallaan tuotettu virhe unohtuu systeemiin sen korjausvaiheessa. Lisäksi tahallaan tuotetut virheet saattavat tuottaa uusia virheitä muihin osiin.

6 Käytännön testauksen toteuttaminen

Luvussa kuvataan ensinnäkin testattava sovellus ja sen osat. Sen jälkeen kuvataan toteutettu testaus. Testaus eteni pääosin V-mallin mukaisesti soveltaen käytännössä luvuissa 3.3-3.5 esitettyä teoriaa testausprosessin kulusta ja testausdokumentaatioiden laatimisesta. Testauksessa on käytetty luvuissa 4 ja 5 esitettyjä menetelmiä apuna. Toteutettu testaus on systeemitestausta, joten käytetyt menetelmät painottuvat luvussa 5.5 esitettyyn mustalaatikkotestaukseen.

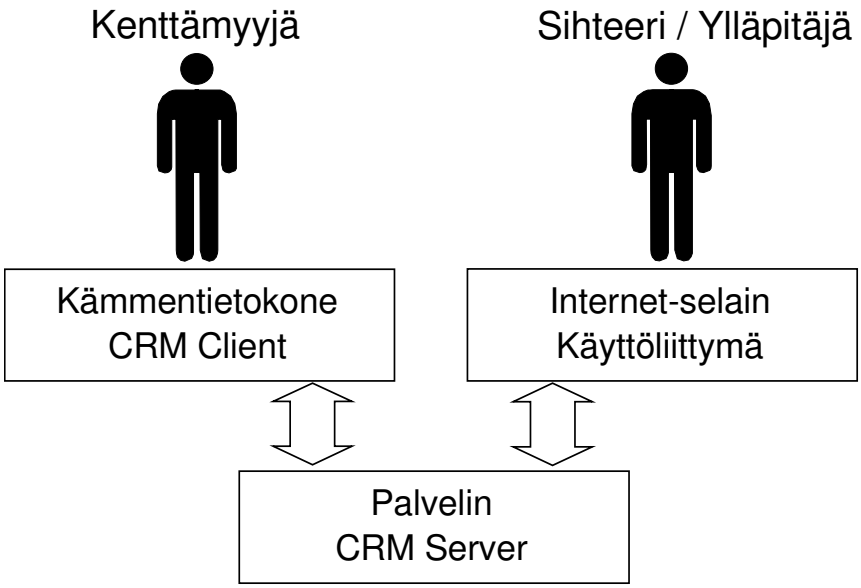
6.1 Testattavan järjestelmän kuvaus

Luku esittelee kehitetyn kenttämyyjien työkalun tarkoitusta ja toimintaa. Kenttämyyjien työn avuksi kehitettiin taskutietokoneella käytettävä sovellus. Samaa järjestelmää käyttävien sihteereiden ja ylläpidon tarpeisiin kehitettiin Internet-selaimen kautta käytettävä sovellus, jonka avulla käytetään palvelinta. Nämä kaksi sovellusta integroitiin kehityksen lopussa yhdessä toimivaksi järjestelmäksi, joten data näiden sovellusten välillä voidaan replikoida aina tarvittaessa.

6.1.1 Järjestelmän tarkoitus

Kenttämyyjä työskentelee päivittäin asiakkaiden luona. Hän tarvitsee tietoa tulevista tapaamisista ja hänen on voitava raportoida käynneistään. Yritys halusi työntekijöilleen helposti käytettävissä olevan sovelluksen. Tavoitteena oli kehittää sovellus, jolla kenttämyyjät voivat nopeasti raportoida asiakaskäyntinsä ilman PC:tä. Ratkaisuna tarpeeseen päädyttiinkin toteuttamaan sovellus taskutietokoneelle, koska kenttämyyjän on helppoa kuljettaa sitä aina työssä mukanaan.

Sovellusprojektina toteutettiin CRM Client -niminen kenttämyyjän työkalu taskutietokoneelle. Palvelinta käytetään WWW-selaimelle kehitetyn käyttöliittymän kautta. Tämä palvelinsovellus kulkee nimellä CRM Server. Sovelluksessa on kolme erilaista käyttäjäroolia: kenttämyyjä, sihteeri ja ylläpitäjä. Kukin heistä hyödyntää järjestelmää omin tavoin asiakkuuden hallintaan. Kuva 14 esittää systeemin osat ja pääasialliset käyttäjien roolit yksinkertaistetusti.



Kuva 14. Järjestelmän rakenne.

6.1.2 Järjestelmän osat ja toiminta

Uuden CRM Client -sovelluksen avulla **kenttämyyjä** saa asiakkaan tiedot ja tapaamisen tavoitteet tietoonsa nopeasti ennen asiakkaan tapaamista, joten hän pystyy valmistautumaan paremmin. Ylläpito pystyy määrittelemään kaavakkeet raporteille, jotka kenttämyyjät täyttävät asiakaskäyntien jälkeen. Kenttämyyjä voi täyttää tämän raportin heti käynnin jälkeen taskutietokoneella ja replikoida datan yrityksen tietokantaan ennen siirtymistä seuraavaan tapaamiseen. Näin hän tarvitsee tietokonettaan ja pääsyä Internetiin käyttääkseen CRM Server -sovellusta vain tietyissä harvoin suoritettavissa tehtävissä.

Sihteerit taas eivät tarvitse CRM Client -sovellusta, sillä he työskentelevät toimistolla ja käyttävät tietokonetta töissään. Sovellusprojektissa toteutettiinkin heidän tarpeisiinsa CRM Server -niminen palvelinsovellus, jota käytetään Internet-selaimelle kehitetyn käyttöliittymän kautta. Tällä sihteerit voivat hallita kenttämyyjien tapaamisia ja tutkia raportointia. **Ylläpitäjät** käyttävät tätä sovellusta käyttäjien hallintaan. Sihteerit näkevät sovelluksessa raportit, jotka perustuvat kenttämyyjiltä kerättyyn dataan. Tämän sovelluksen avulla sihteerillä on aina käytössään ajantasaista tietoa toisin kuin järjestelmissä, joissa käytetään kuukausittaista raportointia.

CRM Server toimii palvelinsovelluksena ja sille kehitetyn WWW-selaimella toimivan käyttöliittymän kautta on aina nähtävissä ajan tasalla olevat tiedot. CRM Client on asiakas, jonka tiedot kenttämyyjä voi aina halutessaan replikoida palvelimen kanssa yhteneväksi. Replikoinnin yhteydessä tiedot replikoituvat myös taskutietokoneen kalenteriin. Sen sijaan kalenteriin tehdyt muutokset eivät replikoidu kehitettyyn järjestelmään.

6.2 Järjestelmän osat systeemitestauksen kannalta

Systeemitestausvaiheessa toteutetussa testauksessa käytettiin menetelmänä luvussa 5.5 käsiteltyä mustalaatikkotestausta. Täten sovelluksen sisäistä toimintaa ei testattu, vaan ainoastaan syötteitä ja vasteita tarkkailemalla testattiin toiminnan oikeellisuutta. Systeemitestaus painottui käyttöliittymien ulkoasuun, toiminnallisuuden testaamiseen käyttöliittymien kautta sekä tietojen replikoitumiseen.

Systeemitestaus toteutettiin **kolmessa vaiheessa** sovellusprojektin kulun mukaisesti. Ensimmäisenä testattiin asiakkaana oleva taskutietokoneella toimiva CRM Client -sovellus. Toisena testattiin palvelinpuolen CRM Server -sovellus, jota käytetään Internet-selaimella käyttöliittymän kautta. Tärkeimpinä tavoitteina näissä kummassakin testauksessa oli käyttöliittymien ulkoasun ja toiminnan oikeellisuus.

Kolmas systeemitestauksen osa-alue oli järjestelmän integraatiotestaus, jossa testattiin tietojen replikoitumisen oikeellisuus. Tämä testaus toteutettiin ensinnäkin CRM Client ja CRM Server -sovelluksien välillä sekä toiseksi tietojen replikoitumisen onnistumisena taskutietokoneen kalenteriin.

Systeemitestauksessa kaikkien kolmen testattavan osa-alueen testit ovat keskenään melko samanlaisia. Tässä luvussa kuitenkin tarkastellaan näiden testauksen ominaisuuksia ja erityisesti näiden kolmen testauksen alueen eroavuuksia.

Testauksessa tarkasteltiin ainoastaan kehitetyn sovelluksen toimintaa. Täten mitään muita järjestelmän hyödyntämiä sovelluksia ei testattu. Testausvaiheessa tehtiin vähäisessä määrin rasitustestausta (katso luku 5.6.1). Käytettävyyttä (katso luku 4.3.2) testattiin vain muun testauksen ohella implisiittisesti.

Huomioitava seikka on, että tämä kehitetty versio on tuotteen ensimmäinen versio. Kaikissa kohdin ei siten kehityksen ja testauksen osalta pyritty täydellisyyteen, vaan jätettiin sovelluksen käyttäjille vastuuta käyttää sitä suhteellisen hyvin ohjeita noudattaen. Myöhemmissä versioissa saatetaan kehittää enemmän käyttäjän tekemien virheiden käsittelyä. Kaikki virhetilanteet eivät monestikaan johdu tahallisesta väärinkäytöstä, vaan osa niistä johtuu vahingoista.

6.2.1 CRM Client -sovelluksen testaus

Systeemitestauksen ensimmäinen vaihe käsitti taskutietokoneella toimivan CRM Client -sovelluksen testauksen. Testauksessa käytiin läpi kaikki sovellukseen kehitetyt näkymät ja dialogit. CRM Client testattiin varmistaen sovelluksen toiminnallisuus.

Ensinnäkin testauksen piirissä oli **käyttöliittymän** ulkoasu sekä kaikkien kenttien ja näkymien tietojen oikeellisuus. Toiseksi testattiin sovelluksen **toiminnallisuudet**. CRM Client -sovelluksen systeemitestausvaiheessa toiminnallisuuden testaus oli rajoitettu ainoastaan tähän sovellukseen, eikä replikoitumista palvelimen tai puhelimen kalenterin välillä vielä tapahtunut.

Myös **tietokantatestausta** suoritettiin jossain määrin. Taskutietokoneen tietokantaan oli ennen testauksen aloittamista tallennettu testausta varten luotua dataa, joten sen oikeellinen tallentuminen oli testattava. Sovelluksessa käyttäjä voi myös kirjoittaa pieniä raportteja, joiden tallentuminen puhelimen tietokantaan oli testattava.

Sovellusta ei varsinaisesti testattu rasitustilanteessa (katso luku 5.6.1). Suorituskykyä ja toimintavarmuutta testattiin implisiittisesti muiden testien ohella seuraten, että sovelluksessa ei ole mitään erityisiä hitaita toimintoja haittaamassa sovelluksen käyttöä. Testauksen aikana tarkkailtiin jossain määrin myös käytettävyyttä (katso luku 4.3.2) miettien, tuntuvatko toiminnot intuitiivisilta loppukäyttäjille.

6.2.2 CRM Server -sovelluksen testaus

CRM Server -sovellus testattiin käyttäen Internet-selainta. Tämän palvelinsovelluksen systeemitestausvaiheessa ei testattu kommunikointia muiden projektissa kehitettyjen sovellusten kanssa. Vastaavasti kuin CRM Client -sovelluksen testauksessa, niin tässäkin ensimmäinen tärkeä asia oli **käyttöliittymän** ulkoasun testaaminen. Toisaalta olennaista oli kaikkien käyttöliittymässä olevien **toiminnallisuuksien** testaaminen.

Kolmas erityisen tärkeä testauksen osa-alue olivat **tietokantahaut** ja **tietokantaan tallentaminen**. Joka kerta tietoa luettaessa ja kirjoitettaessa tarkistettiin oikeellisuus sekä sovelluksessa että tietokannassa.

Käyttöliittymän **käytettävyyttä** (katso luku 4.3.2) ei suoranaisesti testattu, mutta muiden testien ohessa tarkastettiin käytettävyyden olevan riittävä eli on mahdollista tehdä kaikki määritellyt tehtävät ilman erityisiä vaikeuksia. Suoritettua rasitustestausta (katso luku 5.6.1) ja muita CRM Server -sovelluksen systeemitestaukseen liittyviä yksityiskohtia käsitellään luvussa 6.4.

6.2.3 Järjestelmän integraatiotestaus

Järjestelmän integraatiotestaus kohdistuu määritelmänsä mukaisesti kaikkien sovellukseen kuuluvien osa-alueiden integraatioon. Käytetyllä dynaamisella mustalaatikkotestauksella varmistetaan, että järjestelmiin toteutetut integraatiotoimenpiteet muodostavat oikealla tavalla toimivan järjestelmän. Lisäksi on testattava integraatio järjestelmän ja ympäristön välillä, joka käsittää rajapinnat sekä laitteiston, ohjelmiston ja ympäristön osien integraation. Määritelmän mukaan on huomioitava, aiheuttaako kehitys muutoksia nykyiseen, käytössäolevaan ympäristöön.

Tässä järjestelmän integraatiotestauksessa tärkeintä oli **replikoitumisen** testaaminen. Replikoinnin tuli toimia kolmen eri sovelluksen välillä. Ensimmäisenä testattiin datan replikoituminen CRM Client -sovelluksen ja CRM Serverin välillä. Toinen testauksen kohde oli datan replikoituminen CRM Clientin ja taskutietokoneen kalenterin välillä. Tämä jälkimmäinen replikointi oli vain yksisuuntaista, koska taskutietokoneen kalenteriin tehtyjä muutoksia ei replikoitu kehitettyyn sovellukseen.

Tässä testausvaiheessa ei suoritettu varsinaista rasitustestausta (katso luku 5.6.1), koska se on suuresti riippuvainen replikoinnin nopeudesta. Testauksessa käytetty ympäristö oli teholtaan huonompi kuin asiakkailla loppukäytössä oleva.

6.3 Yleistä testauksen suunnittelusta ja toteutuksesta

Projektissa käytetty testaussuunnitelman malli ohjasi suunnitteluvaihetta, sillä se muistutti huomioimaan olennaisia testauksen suorittamiseen liittyviä tekijöitä. Testaussuunnitelmassa esiteltiin koko suunniteltu systeemitestauksen kulku ja siihen vaikuttaneet tekijät.

Menetelmänä systeemitestauksessa käytettiin kaikissa vaiheissa pääosin dynaamista mustalaatikkotestausta, joka pitkälti ohjaa testauksen suorittamista. Erityisesti mustalaatikkomenetelmän vaikutus näkyi testitapauksissa. Menetelmä antoi neuvoja muun muassa testiaineiston valintaan (katso luvut 5.5.3-5.5.6) ja testauksen kulun suorituspolkuihin (katso luku 5.5.9). Testausstrategioista sovellettiin rasitus- ja käyttöliittymätestausta (katso luvut 5.6.1 ja 5.6.2). Näiden lisäksi testaukseen vaikuttaneita tekijöitä olivat muun muassa aikataulu, roolit ja työkalujen käyttö (katso luvut 3.4.1, 3.4.4 ja 3.6). Kaikki nämä testaukseen vaikuttaneet tekijät kirjoitettiin osaksi testaussuunnitelmaa.

6.3.1 Testauksen dokumentaatiot

Testaus toteutettiin V-mallin mukaisesti. Tällöin testauksen suunnitelmat pyrittiin suurimmalta osin tekemään projektin määrittelyjen valmistuttua. **Testaussuunnitelmassa** (katso luku 3.5.1) esiteltiin testauksen toteuttamiseksi tarvittavat asiat. Kullekin testauksen kolmesta vaiheesta tehtiin omat testaussuunnitelmansa.

Ensinnäkin testaussuunnitelmissa määriteltiin testauksen osa-alueet ja painopiste, jotka kunkin testauksen kannalta on esitetty luvussa 6.2. Testaus myös rajattiin koskemaan tiettyjä toiminnallisuuksia. Suunnitelma kuvasi suunnitellun aikataulun ja testausresurssit yksityiskohtaisesti ihmisistä työkaluihin.

Testaussuunnitelmassa esiteltiin myös testauksen **tavoite** kuvaten, miksi testausta suoritetaan. Tärkeimpänä tavoitteena on varmistaa järjestelmän oikeellisuus suunnitteludokumentaatioon verrattuna. Tällä myös pyritään useissa projekteissa esittämään muillekin kuin testaajille testausvaiheen tärkeys ja välttämättömyys.

Seuraavaksi kuvattiin testauksen **lähestymistapa** eli miten testataan. Tässä tarkoitetaan mustalaatikkotestauksen (katso luku 5.5) mukaisesti kirjoitettujen testitapausten suorittamista sekä joitakin rasitus- ja käyttöliittymätestejä (katso luvut 5.6.1-5.6.2).

Testaussuunnitelmassa määriteltiin myös **aloitus- ja lopetuskriteerit** kullekin testikierrokselle. Testaus voitiin aloittaa, kun kehittäjät olivat saaneet tarvittavat yksikkö- ja integraatiotestit suoritetuiksi. Kukin kirjoitettu testitapaus määriteltiin hyväksytysti suoritetuksi, kun kaikki siihen kirjatut tulokset oli saatu johdonmukaisesti tuotettua.

Testaussuunnitelma myös määritteli, milloin testaus tulisi **keskeyttää** kesken testikierroksen. Esimerkiksi tässä testauksessa kierros määriteltiin keskeytettäväksi aina tuhoisan virheen ilmaantuessa. Tuhoisat virheet tuli aina korjata ennen kuin testausta voitiin jatkaa. Vakaviksi luokitellut virheet aiheuttivat testauksen keskeytyksen ainoastaan, jos ne olivat todella olennaisia. Alempien tasojen virheiden ilmaantuessa ei testauskierrosta tarvinnut keskeyttää.

Testitapaukset (katso luku 3.5.2) muodostivat yhden testaussuunnitelman tärkeimmistä osista. Ne ohjeistivat itse testauksen suorittamisen. Yleisesti ne auttoivat myös toistettavuudessa, koska kirjoitettujen tarkkojen ohjeiden mukaan samat testit voitiin suorittaa uudelleen. Isommissa projekteissa eri testaajat voivat pelkät testitapaukset ohjeenaan suorittaa toisiltaan apua kysymättä kaiken lähes samalla tavalla kuin testitapauksen suunnittelija. Testitapauksien suorittamista käytettiin myös jäljitettävyydessä eli arvioitaessa, kuinka suuri osa testauksesta on suoritettu ja kuinka paljon aikaa testaus vielä tulisi vaatimaan. Näiden valintaa käsitellään tarkemmin testausmenetelmien valinnan osassa luvussa 6.4. Testitapauksia käytettiin systeemitestauksessa ja myös asiakas halusi hyödyntää hyväksymistestauksessaan osaa niistä.

Testitapauksia suoritettaessa löydettiin virheitä. Ne raportoitiin käyttäen ennalta määriteltyä **virheraporttia** (katso luku 3.5.3), joka sitten toimitettiin kehittäjille. Tätä systeemitestausta varten muokattu virheraportti on esitetty liitteessä 1.

6.3.2 Testauksen ajoitus, roolit ja työkalut

Kehitetyn järjestelmän ohjelmistoprojekti jakaantui kahdeksi vaiheeksi. Ensimmäisessä vaiheessa toteutettiin CRM Client -sovellus taskutietokoneelle. Toisessa vaiheessa toteutettiin CRM Server -sovellus palvelimelle ja toteutettiin koko systeemin integraatio. Kehitysvaiheen aikana koodaajat toteuttivat yksikkö- ja integraatiotestaukset. Niiden jälkeen oli tässä käsitellyn systeemitestauksen vuoro, jonka testiryhmä suoritti. Systeemitestausvaiheen päätyttyä hyväksytysti asiakas toteutti hyväksymistestauksen omissa tiloissaan.

Sovelluksen systeemitestaus **ajoittui** (katso luku 3.4.1) järjestelmän kehityksen mukaisesti. Projektin ensimmäisen vaiheen kehityksen valmistuttua toteutettiin CRM Client -sovelluksen systeemitestaus ja vastaavasti toisen vaiheen jälkeen toteutettiin CRM Server -sovelluksen systeemitestaus. Kun nämä olivat päättyneet, suoritettiin koko systeemin integraatiotestaus.

Testauksen **lopetusajankohta** (katso luku 3.4.2) määräytyi periaatteessa ennalta määrätyn ajan mukaisesti. Kullekin testausvaiheelle oli ennalta laskettu tietty määrä päiviä käytettäväksi. Testauskierroksen eli vaiheeseen kuuluvien testitapausten suorittamisen kertaalleen arvioitiin kestävän päivän. Kunkin testauskierroksen perään oli varattu kehittäjille korjausaikaa. Testaus sai joissain vaiheissa lisäaikaa, koska virheitä edelleen löytyi. Täten varsinaiseksi lopetusajankohdaksi muotoutui ajankohta, jolloin testauskierroksella ei löytynyt enää tuhoisia tai vakavia virheitä ja häiritseviä virheitä oli korkeintaan muutamia.

Sovellusprojekti oli pieni, joten testiryhmään kuului vain yksi ihminen. Täten erityistä **roolijakoa** (katso luku 3.4.4) ei testauksessa tarvittu. Testiryhmä toteutti pääasiassa itsenäisesti testauksen suunnittelun ja toteutuksen.

Testaus toteutettiin suorittamalla kaikki etukäteen valmistetut testitapaukset pääosin manuaalisesti. Käytössä oli **työkalu** muutamien suorituskykytestien tekoon, mutta sitä käytettiin vain vähän (katso luku 3.6). Rasituksen siedon mittaaminen suuressa mittakaavassa ei olisi ollut järkevää, koska testausympäristö oli teholtaan huomattavasti alhaisempi kuin asiakkaalla oleva ympäristö.

6.4 Esimerkkinä palvelimen mustalaatikkotestaus

Kehitetyssä järjestelmässä kaikkien kolmen osa-alueen testaus oli suhteellisen samanlaista, vaikka kyseessä olivatkin sovellukset erityyppisissä laitteissa. Täten tutkielmassa tarkemmin esitellään vain yhden osa-alueen eli CRM Server -palvelimen testaus.

Testaus suunniteltiin ja osittain myös suoritettiin ennen kuin testauksen suorittaja tunsi teoriaa tässä tutkielmassa esitetyssä laajuudessa. Tämä vaikutti erityisesti siihen, että testausta toteutettaessa ei ollut tietoa kaikista käsitteistä. Niiltä osin kuin testausta tästä huolimatta on osattu toteuttaa tiettyjen käsitteiden mukaisesti, niitä esitellään.

6.4.1 Testauksen periaatteita

Testaus on pääasiassa **dynaamista mustalaatikkotestausta** (katso luku 5.5) keskittyen loppukäyttäjälle näkyvien ominaisuuksien ja toiminnallisuuksien testaamiseen. Päätavoitteena oli varmentaa CRM Server -sovelluksen ulkoasun ja toiminnallisuuden olevan määriteltyjen vaatimusten mukainen.

Testauksen yhteydessä testattiin myös datan tietokantaan tallentamisen ja sieltä lukemisen oikeellisuus. Nämä testit oli määritelty jo testitapauksiin. Testauksessa ei sen sijaan kiinnitetty huomiota yhteistoimintaan muiden toteutettujen ja palvelimella toimivien sovellusten kanssa. Replikoinnin suorittamista testattiin vasta myöhemmin toteutetun järjestelmän integraatiotestauksen aikana.

Testitapaukset luotiin käyttöliittymän määrittelydokumentin ja käyttötapausten määrittelyjen pohjalta. Testitapaukset kirjoitettiin näihin perustuen aiemmista projekteista saadun kokemuksen mukaisesti. Valitsemalla testitapaukset näin ne pyrittiin saamaan mahdollisimman kattaviksi.

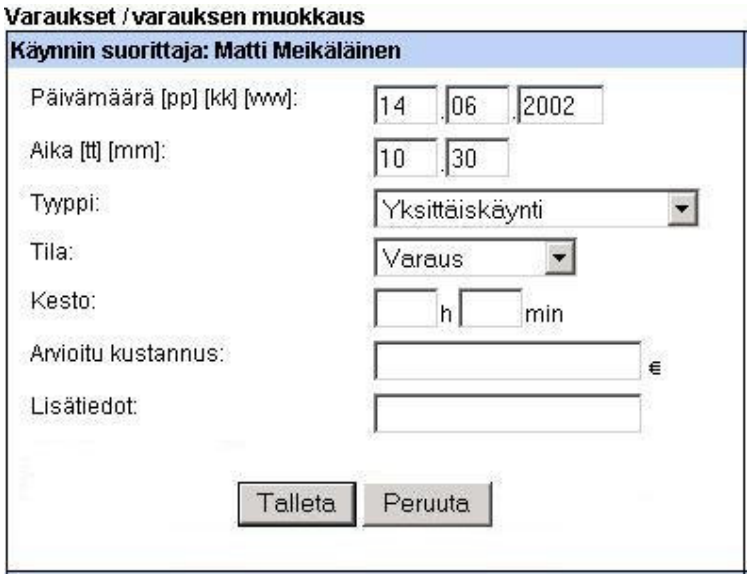
Oikeastaan testauksessa sovellettiin myös **staattista mustalaatikkotestausta** (katso luku 5.5.2), koska kirjoitettaessa testitapauksia käytiin tuotteen määrittelyä tarkasti läpi. Sieltä havaittiin tämän aikana joitakin virheitä tai puutteita, jotka määrittelyn kirjoittaja korjasi ja tarkensi dokumenttiin ajan salliessa.

Pääasiassa testaus tehtiin siis dynaamista mustalaatikkotestausta käyttäen. Testausta varten kirjoitettiin kymmenen testitapausta. Näistä kukin liittyi aina päänäkymiin. Kussakin testitapauksessa testattiin pääsy kyseiseen näkymään, tarkkailtiin näkymän oikeellisuutta ja testattiin toiminnallisuus.

6.4.2 Testiaineiston valinta kalenteritapahtumien luontia varten

Kalenteritapahtumien luonti on yksi sovelluksen perustehtävistä. Se esitelläänkin tässä tarkemmin esimerkkinä siitä, kuinka **dynaamisen mustalaatikkotestauksen menetelmä** ohjaa testiaineiston valintaa. Mustalaatikkotestauksen menetelmiä käytettiin testiaineiston valinnassa, vaikka sen suorittamisen aikana ei vielä ollut riittävästi tietoa testauksen teoriasta ja käsitteistä, kuten ekvivalenssiluokista.

Kalenteritapahtumat luotiin CRM Server -sovelluksessa kuvassa 15 esitetyn kalenteritapahtuman varausikkunan kautta. Erilaisia kenttien tyyppejä on neljä eli aika, alasvetovalikko, tekstikenttä ja kenttä maksulle. Näille kaikille on testitapauksia varten määriteltävä **ekvivalenssiluokat**.



Kuva 15. Kalenteritapahtuman luontinäkymä.

Kalenteritapahtuman luonnin päivämäärä, aika ja kesto ottavat syötteenään vain numeroita. Ne toteutettiin ilman erityisiä oikeellisuustarkastuksia. Siis jos päivämääräksi antaa `32.06.2002`, muuttaa systeemi sen automaattisesti päivämääräksi `2.7.2002`. Täten ekvivalenssiluokkia on kaksi. Oikeaan luokkaan päivämäärän päivän ja kuukauden tapauksessa kuuluvat numerot `1, 2,...,99` ja vuoden tapauksessa `1, 2,...,9999`. Kaikki muut syötteet kuuluvat virheelliseen luokkaan. Testauksen aikana alasvetolistat on kokeiltava jokaisella vaihtoehdollaan. Tekstikentät hyväksyvät kaikenlaisia syötteitä, jotka eivät ylitä maksimipituutta.

Rahamäärän kenttä on varsin mielenkiintoinen testauksen kannalta. Sen oikeaan luokkaan kuuluvat lukuarvot. Väärään luokkaan kuuluvat kaikki muut. Tässä testitapauksiin valitaan syötteet sallituilla arvoilla. Näitä sallittuja arvoja ovat `123` ja `123,5`. Lisäksi valitaan virheellisen luokan syöte `kissa`. Sommerville kehotti muuttamaan virheellisen syötteen paikkaa (katso luku 5.5.4), joten on hyvä antaa syötteenä myös `kissa123`, `12kissa3` ja `123kissa`. Tämä kenttä ei ole pakollinen ja myös tyhjän arvon pitäisi toimia, joten testauksessa kokeiltiin sitäkin.

Sitten päästään valitsemaan dataa **raja-arvojen** avulla (katso luku 5.5.5). Tässä päivämääräkenttään kiinnostavia tapauksia ovat vuosissa varsin pienet ja suuret. Siis annetaan pienin arvo `1.1.0001`. Tässä projektissa oletettiin tätä sovellusta käytettävän lähivuosina, eikä enää satojen vuosien päästä. Projektissa taskutietokoneen sovellus asetettiinkin tunnistamaan suurimmaksi päivämääräksi päivän `1.1.2500`, joka siis valittiin syötteeksi. Syötteenä kokeiltiin myös liian isona arvona `2.1.2500`. Liian pienistä arvoista kannattaa valita muun muassa `0.0.0000`. Miinusmerkkisiä arvoja näihin kenttiin ei voi syöttää, joten niitä ei voi testatakaan.

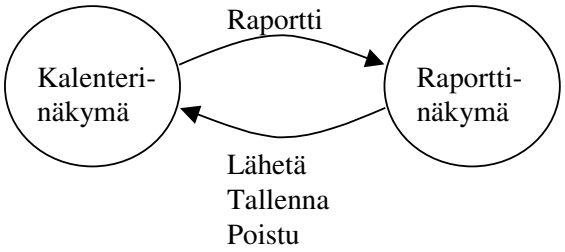
Näiden lisäksi tähän kenttään **arvattiin** arvoja (katso luku 5.5.6). Kalenteritapahtuman luonnissa yllä olevat valintatavat ovat suhteellisen perusteellisia, joten **satunnaisten** arvojen käyttö tuskin lisää onnistumista, mutta niitäkin pyrittiin kokeilemaan.

6.4.3 Toteutus mustalaatikkomenetelmän keinoin

Dynaamisen mustalaatikkotestauksen menetelmä käsittää sekä tiedon että toiminnan testauksen (katso luvut 5.5.7, 5.5.9). **Tiedon testauksessa** tutkitaan periaatteessa kaiken tiedon käsittelyä. Tässä tutkittiin erityisesti sovelluksen kykyä käsitellä syötteiden erikoistapauksia. Näitä olivat muun muassa nollien ja oletusarvojen käyttäytyminen.

Tiedon testaukseen liittyi myös pakollisiksi määrättyjen kenttien jättäminen tyhjäksi (katso luku 5.5.7). Näiden jättäminen tyhjiksi ei saisi onnistua tai sitä yritettäessä olisi vähintään saatava huomautus arvon puuttumisesta. Kalenteritapahtuman luonnissa pakollisia kenttiä olivat päivämäärä ja aika. Niinpä niitä kokeiltiin tyhjillä arvoilla. Tyhjä arvo syötteenä annettaessa tuli saada ainakin kunnollinen huomautus arvon puuttumisesta. Myös nollat kuuluvat erikoistapauksiin, joita sovelluksen ei kuulunut hyväksyä.

Toiminnan testaukseen käytettävää tilakaaviotestausta (katso luku 5.5.9) toteutettiin arkkitehdin suunnitteluvaiheessa toteuttaman tilakaavion avulla. Tässä kaaviossa oli esitetty kaikki näkymät ja dialogit sekä komennot niiden välillä liikkumiseen. Kuva 16 esittää pienen osan tästä tilakaaviosta. Testaus toteutettiin kokeilemalla polkuja näiden välillä.



Kuva 16. Tilakaavion raporttinäkymän toiminta.

Tilakaaviotestausta varten kirjoitettiin testitapaukset, joissa suorituspolut kattoivat kaikki näkymät. Tapauksilla testattiin pääsy kaikkiin näkymiin sekä niiden yleinen toiminnallisuus. Ensin kuljettiin valittuun testattavaan näkymään ja annettiin syötteet kaikkiin kenttiin, minkä jälkeen poistuttiin tallentamatta muutoksia. Seuraavaksi valittiin sama näkymä uudelleen ja tarkastettiin kenttien olevan alkuperäisissä arvoissaan. Kentät täytettiin ja näkymästä poistuttiin tallentaen muutokset. Tämä näkymä valittiin vielä uudelleen ja tarkastettiin, että tieto todella oli tallentunut.

Luvussa 5.5.11 esitettiin vielä muitakin menetelmiä. Näitä olivat käyttäytyminen tyhmänä käyttäjänä, virheiden etsinnän suuntaus aiemman tiedon perusteella ja kokemuksen tuoma lähestymistapa. Näitä ei kirjoitettu testitapauksiin, mutta yhden testaajan projektissa niitä kuitenkin voitiin käyttää. Virheiden etsinnän kohdistus tiettyihin toiminnallisuuksiin voitiin toteuttaa vasta, kun oli testattu jonkin aikaa. Tämän jälkeen testausta suunnattiin erityisesti virheitä sisältäneisiin kohtiin. Kokemus kasvoi jo yhdenkin projektin aikana ja testaaja oppi etsimään virheitä niille ominaisista tilanteista.

6.4.4 Muut systeemitestauksen menetelmät

Osittain testitapaukset kirjoitettiin hyväksyntä- ja osittain hylkäämistesteiksi (katso luku 5.2). Näin ensin pyrittiin vain kokeilemaan koodin perustoiminta **hyväksyntätesteillä**, joita esitettiin luvussa 6.4.3. Kun tämä testaus saatiin hyväksyttynä läpi, siirryttiin **hylkäämistesteihin**. Näiden aikana pyrkimys oli kaataa sovellus ja kokeilla mahdollisimman erikoisia polkuja, jotta vielä löydettäisiin piileviä virheitä.

Hylkäämistesteihin valittiin tekstikenttien arvoiksi sekä tyhjiä että erittäin pitkiä kaikkia mahdollisia merkkejä sisältäviä syötteitä. Sovelluksessa erityisesti skandinaavisten merkkien käsittelyyn kiinnitettiin huomiota. Hylkäämistesteissä kokeiltiin, mitä numeerisissa kentissä tapahtuisi isoilla arvoilla. Ensin päivämääräkenttä testattiin kaikista suurimmalla numerokenttiin mahtuvalla arvolla `31.12.9999`. Toiseksi annettiin syöte `32.12.9999`. Tämän sovellus pyrki muuttamaan päiväksi `1.1.10000`, joka taas ei mahtuisi kenttään.

Testaus sisälsi myös **rasitustestausta** (katso luku 5.6.1) työkalun avulla. Tällä testattiin järjestelmän toimintaa, kun yhtäaikaisia käyttäjiä olisi 5, 10 tai 20. Tämän enempää yhtäaikaisia käyttäjiä järjestelmällä ei tule olemaan, joten suurempia määriä ei kehitysversion aikana ollut olennaista kokeillakaan. Näillä tapauksilla testattiin myös, mitä tapahtuu, kun järjestelmä saa monta yhtäaikaista pyyntöä. Näitä testejä tosin vaikeutti testausympäristön verkon hitaus.

Regressiotestaus (katso luku 4.3.5) suoritettiin aina, kun kehittäjät saivat edellisen testikierroksen virheet korjatuiksi. Tätä testausta varten ei kirjoitettu uusia testitapauksia, vaan käytettiin soveltuvin osin olemassaolevia testitapauksia. Kullakin regressiotestauskierroksella keskityttiin lähinnä vain kehittäjiltä takaisin saatuihin virheraportteihin. Niissä esitellyistä ennen korjausta ilmenneistä virheistä testattiin, olivatko ne kunnossa ja oikealla tavalla toimivia. Lisäksi regressiotestauksessa pyrittiin havaitsemaan, oliko korjauksella vaikutuksia muualle sovellukseen.

7 Testauksen tulosten ja valittujen menetelmien analysointia

Luvussa esitellään toteutetun järjestelmän kehitysprojektin (katso luku 6) testauksessa saatuja tuloksia ja analysoidaan niiden vaikutusta. Testikierrokset käytiin läpi useamman kerran, mutta kuitenkin joitakin virheitä löytyi vielä testauksen jälkeenkin. Luvussa käsitellään myös, olisiko testausta jotenkin muuten suorittamalla löydetty nämä virheet aiemmin ja helpommin sekä taattu perusteellisempi testaus. Tämän tutkimuksen ja testauksen tulosten analysoinnin on tarkoitus parantaa testauksen toteutusta seuraavissa projekteissa.

7.1 Testauksen tuloksia

Testaus suoritettiin V-mallin (katso luku 3.3) ohjeita mukaillen. Testaussuunnitelmista (katso luku 3.5) tehtiin ensimmäiset versiot heti tuotteen määrittelyjen valmistuttua. Tosin määrittelyt muuttuivat ja tarkentuivat paljon koodauksen aikana sekä joitakin muutoksia tehtiin jopa testauksen aikana. Täten myös testaussuunnitelmia täytyi päivittää myöhäisessä vaiheessa. Tästä aiheutui joitakin virheitä testaussuunnitelmiin, koska kaikki määrittelyjen muutokset eivät tulleet testiryhmän tietoon tai jäivät testauksen kiireiden takia kirjaamatta.

Suoritettu testaus kuului systeemitestausvaiheeseen. Tutkielman luku 4 käsitteli kuitenkin kaikki testausvaiheet kunnollisen kokonaiskuvan saamiseksi. Testausta suoritettaessa tästä oli havaittavissa hyöty, että ymmärsi ja arvosti enemmän muidenkin testausvaiheiden merkitystä sekä erityisesti kehittäjien ja loppukäyttäjien suorittamaa testausta.

Luvun 3.3 ohjeiden mukaisesti ensisijainen tehtävä testauksella on varmentaa, että järjestelmä toteuttaa asiakkaan vaatimukset ja käyttäjän tarpeet eli on ”rakennettu oikea järjestelmä”. Toisaalta käyttäjien vaatimukset tulee olla toteutettu määrittelyjen mukaisesti eli ”järjestelmä on rakennettu oikein”. Näistä ensimmäinen tehtävä kuuluu suurimmalta osin vasta hyväksymistestauksen piiriin. Jälkimmäinen taas olettaa määrittelyn olevan luotettava ja täydellinen.

Suoritettu testaus pohjautui kahdelle edellä mainitulle testauksen ohjeelle. Tosin määrittelyn paikoittaisten puutteiden vuoksi testauksessa ilmeni muutamia epäselvyyksiä. On kuitenkin vaikea arvioida jälkikäteen näiden hidastavaa vaikutusta.

Lisäksi käsitelty systeemitestaus pohjautui erityisesti mustalaatikkotestaukselle ja sovelsi paikoitellen testausstrategioita (katso luvut 5.5 ja 5.6). Tutkielmassa käsiteltyä lasilaatikkotestausta (5.3-5.4) käyttivät periaatteessa vain kehittäjät jossain määrin. Se kuitenkin antoi muutamia hyviä ja tuloksellisia vihjeitä testauksen suorittamiseen. Toivon mukaan tässä projektissa kehitetyn sovelluksen seuraavia versioita kehitettäessä päästään kokeilemaan myös lasilaatikkomenetelmiä systemaattisemmin ja voidaan havaita, miten niiden käyttö vaikuttaa saataviin tuloksiin.

Osa testauksesta toistettiin monta kertaa, joten työkalu regressiotestausta varten olisi ollut kätevä vähentäen rutiininomaista työtä. Testaustyökalua käytettiin jonkin verran rasitustestien tekoon, mutta sen käyttöä häiritsi testauksen toteutus kehitysympäristössä. Ymmärrettävästi kehitystyö hidasti ympäristöä ja välillä ympäristö ei ollut lainkaan testauksen käytettävissä kehityksen takia. Testaus pitäisi aina voida suorittaa omassa testiympäristössään. Tämä projekti oli kuitenkin suhteellisen pieni, joten siinä ei oltu nähty olennaiseksi rakentaa toista ympäristöä.

7.1.1 CRM Clientin testauksen tulokset

CRM Client -sovelluksen systeemitestauksen täydelliset testikierrokset suoritettiin viiden eri päivän aikana. Jokaista korjauskierrosta seurasi regressiotestaus. Testauksessa käytettiin seitsemää testitapausta, jotka kukin liittyivät aina yhteen päänäytöistä. Ainoastaan yksi näistä testitapauksista ei johtanut virheiden löytämiseen. Yhteensä tämän systeemitestauksen aikana löytyi 39 uutta virhettä, jotka jakaantuivat vakavuuksiltaan (katso luku 2.3) taulukon 2 mukaisesti.

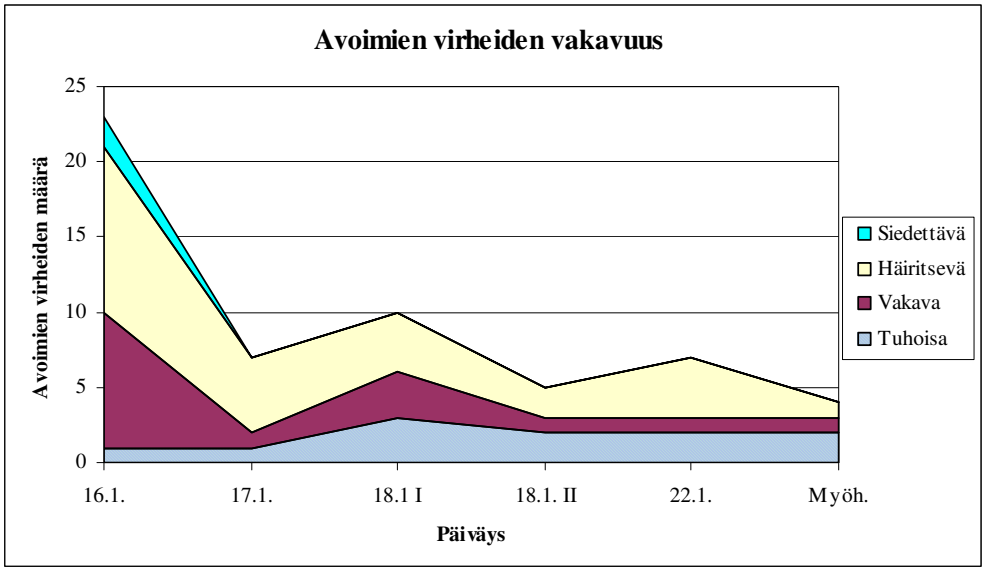
Virheen vakavuus	16.1.	17.1.	18.1. I	18.1. II	22.1.	Yhteensä lkm	Yhteensä %
Tuhoisa	1	0	2	0	0	3	8
Vakava	9	1	3	1	1	15	38
Häiritsevä	11	2	2	1	3	19	49
Siedettävä	2	0	0	0	0	2	5
Yhteensä	23	3	7	2	4	39	100

Taulukko 2. Uusien virheiden jakautuminen virheen vakavuuden suhteen.

Suurin osa virheistä löytyi heti ensimmäisellä testauskierroksella. Viimeinen testauskierros sisälsi lähinnä kokeilua testitapausten ulkopuolelta. Silloin löytyneet häiritsevät virheet liittyivät pääosin oletusarvoihin ja vakava virhe tekstikenttään, josta virheellisesti ei voitu poistaa tekstiä. Taulukko 2 havainnollistaa, että 87% virheistä löytyi keskimmäisimmistä luokista eli oli vakavuudeltaan vakavia tai häiritseviä. Kaikkien eri vakavuusluokkien virheitä kuitenkin löytyi.

Kuva 17 esittää, kuinka kauan eri vakavuusasteen virheet olivat avoimina (katso luku 3.4.3). Koko testauksen aikana löytyi kolme tuhoisaa virhettä. Niiden korjausaika oli pidempi kuin muiden. Testauksen loppua lähestyttäessä avoimina oli enemmän häiritseviä kuin varsinaisia vakavia virheitä, mikä on hyvä asia.

Testauksen lopussa avoimeksi jäi neljä virhettä. Itse asiassa niistä kaksi oli tuhoisia virheitä, jotka jäivät odottamaan myöhempää korjausta. Ne ilmaantuivat rasitustestin yhteydessä, vaikka rasitustestausta suoritettiin vain vähäisessä määrin. Testissä selattiin päivämääriä tai vaihdettiin aktiivista näkymää nopeasti, minkä jälkeen järjestelmä kaatui. Nämä korjattiin vielä ennen sovelluksen siirtoa tuotantoon. Avoimiksi jääneet vakava ja häiritsevä virhe johtuivat osittain ympäristöstä ja ne päätettiin korjata vain ajan salliessa.



Kuva 17. Avoimien virheiden vakavuus.

19 löydetystä 39 virheestä liittyi raportointinäkymään. Tässä näkymässä toiminnallisuus olikin sovelluksen monipuolisin. Näkymässä oli mahdollisuus itse raportoida, muokata ja tallentaa tietoa. Testauksen alussa perustehtävät tiedon tallentamisesta ja muokatun tiedon näyttämisestä aiheuttivat virheitä. Näkymälle suoritettiin myös hylkäämistestejä (katso luvut 5.2 ja 5.5.7) syöttämällä epäkelpoa tietoa, joka aiheutti sovelluksen toiminnassa monia virheitä.

Muuten testauksessa löytyi keskimäärin kolme virhettä kutakin näkymää kohden. Näistä noin puolet johtui siitä, että sovelluksen ominaisuuksia, toiminnallisuutta tai näkymien kenttiä puuttui. Muut virheet käsittelivät sovelluksen hitautta, ulkoasun väärää asettelua tai väärän tiedon esittämistä tietokannasta.

7.1.2 CRM Clientin testauksen tulosten analysointi

Testausta ei päästy aloittamaan aivan laaditun aikataulun mukaisesti. Täten testaus oli suoritettava nopeasti, jotta kehityksen seuraavaa vaihetta päästiin jatkamaan. Tämä todennäköisesti vaikutti siihen, että neljä virhettä jäi odottamaan myöhempää korjausta. Yleensä projektin kannalta olisi parempi, että testaukselta ei vähennettäisi aikaa, sillä testausvaiheessa havaitsematta jääneet virheet siirtyvät loppukäyttäjille. Jo tuotannossa olevan sovelluksen virheiden korjaaminen on huomattavan kallista ja on huonoa mainosta sovelluksen kehittäjälle.

Testauksen kannalta olisi ollut hyvä, jos sovellusta olisi päästy kokeilemaan ennen testauksen aloittamista. Silloin olisi voinut aavistella virheitä. Tämän takia testaajien osallistuminen muun muassa staattisiin lasilaatikkotesteihin (katso luku 5.3) olisi hyvä ratkaisu, mutta näitä testejä ei tässä projektissa toteutettu. Tietysti etukäteen pyrittiin miettimään mahdollisia virhekohtia, jotka tulisi testata huolella. Nämä suunnittelut liittyivät tuotteen määritettyjen vaatimusten tarkasteluun, eikä esimerkiksi muita samantyylisiä sovelluksia ollut kokeiltavana.

CRM Client -sovellukseen tuli muutoksia myöhemmin ja sen testauskierros uusittiin vielä palvelinsovelluksen testauksen aikana. Tällöin löydettiin joitakin uusia virheitä, mutta niistä useimmat olivat vain häiritseviä.

Hankaluutena oli, että testidataa oli suhteellisen vähän. Täten testauksen yhteydessä jäi epäilys, että joitakin virhetilanteita syntyy vasta, kun sovelluksessa on saatavilla riittävä määrä dataa. Tämä paljastui todeksi, kun liitettyjä asiakkaita oli huomattavan paljon, jolloin sovellus muuttui varsin hitaaksi. Vaatimusmäärittelyssä tulisi aina määrittää tarkat arvot sallituille maksimimäärille, jotta testauksessa huomioitaisiin nämäkin. Tämän määritellyn raja-arvon huomattavan ylityksen jälkeen sovelluksen hitaus jää asiakkaan vastuulle. Tässä tapauksessa näin suuren datamäärän luominen testaamisen tässä vaiheessa olisi ollut liian aikaa vievää saatuihin tuloksiin nähden.

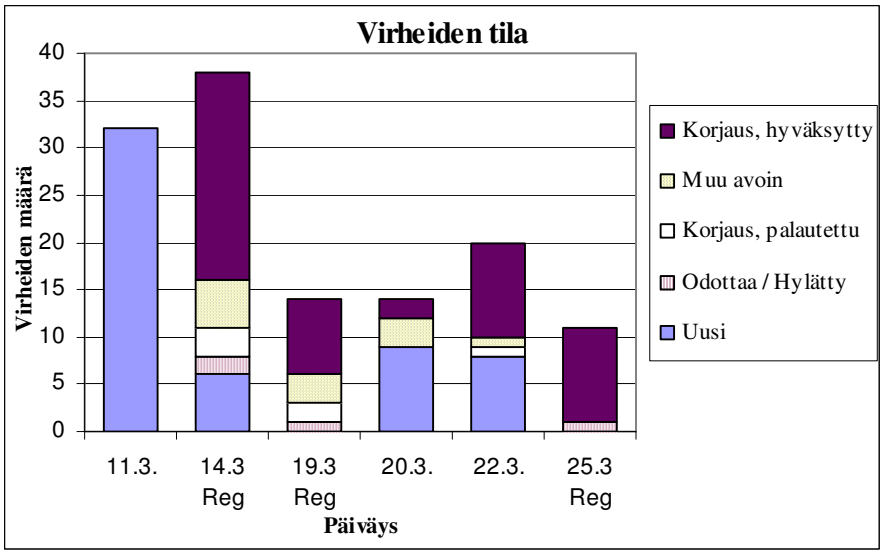
7.1.3 CRM Serverin testauksen tuloksia

Palvelinsovelluksen testauksessa kaikkien kymmenen testitapauksen osa-alueilta löytyi virheitä ainakin kerran. Virheitä löydettiin yhteensä 55 kappaletta. Taulukko 3 esittää löydettyjen uusien virheiden jakautumista eri vakavuusluokkiin. Testauksen aikana ei raportoitu yhtään tuhoisaa virhettä. Tosin myöhemmin tarkastellen voi vetää johtopäätöksen, että kolme vakavaksi luokiteltua virhettä olisi tullut luokitella tuhoisiksi. Sekä vakavia että häiritseviä virheitä löydettiin 24 kappaletta. Loput seitsemän virheistä oli siedettäviä.

Virheen vakavuus	**11.3.**	**14.3.**	**20.3.**	**22.3.**	**Yhteensä lkm**	**Yhteensä %**
Tuhoisa	0	0	0	0	0	0
Vakava	12	3	5	4	24	43,5
Häiritsevä	16	2	3	3	24	43,5
Siedettävä	4	1	1	1	7	13
Yhteensä	32	6	9	8	55	100

Taulukko 3. Uusien virheiden jakautuminen virheen vakavuuden suhteen.

Testauspäivistä 11, 14, 20 ja 22 olivat päiviä, jolloin koko testikierros suoritettiin. Näiden lisäksi päivinä 14, 19 ja 25 suoritettiin regressiotestaus. Kuva 18 esittää virheiden elinkaaren testauksen aikana. Ensinnäkin kuvassa esitetään uudet sekä odottamaan asetetut ja hylätyt virheet. Korjatut virheet on jaoteltu kahdeksi luokaksi. Ensimmäiseen luokkaan kuuluvat vielä virheellisiksi havaitut, jotka siis palautettiin kehittäjille uudelleen korjattaviksi. Toiseen luokkaan kuuluvat hyväksytysti korjatut. Lisäksi taulukossa on kuvattu virheiden ryhmä ”muut avoimet”. Tähän kuuluvat esimerkiksi edellisinä päivinä havaitut virheet, jotka seuraavan testauksen aikana ovat yhä kehittäjillä korjattavina.



Kuva 18. Virheiden tilan muuttuminen testauksen aikana.

Virheitä löydettiin testauksen viimeisinäkin päivinä paljon, koska ensimmäisinä päivinä kolme päätoiminnoista ei ollut lainkaan käytössä. Nämä toiminnot liittyivät asiakkaiden ja toimipaikkojen hallintaan sekä raportointiin. Kahden viimeisen päivän 17 uudesta virheestä 16 liittyi näihin kolmeen päätoiminnallisuuteen.

Yhteensä neljä virhettä kaikista löydetyistä 55 hylättiin (katso luku 3.4.3). Näistä kaksi hylättiin, koska ne olivat statukseltaan siedettäviä ja niiden korjaaminen olisi vaatinut huomattavasti resursseja. Kaksi virheistä hylättiin tämän testauksen piiristä, mutta ne korjattiin myöhemmin ennen tuotantoon menoa.

7.1.4 CRM Serverin testauksen tulosten analysointi

Tutkielman luvussa 5.5.11 esitettiin menetelmä löytää virheitä keskittymällä toimintoihin, joista virheitä on löytynyt aiemminkin. Tällaista testiä sovellettiin muun muassa asiakastiedoissa, joissa käytiin kaikki kentät vuorotellen läpi ja useimpien kohdalla löytyi sama virhe. Asiakkaiden hallinnan osa-alue sisälsi muutenkin runsaasti virheitä. Keskimäärin kullakin kymmenestä testitapauksesta löydettiin neljä virhettä. Tästä huomattavan poikkeuksen teki asiakkaiden hallinta, josta löydettiin 37% kaikista virheistä.

Testaus suoritettiin kehitysympäristössä, mikä osaltaan vaikeutti testaamista. Samaan aikaan tapahtuva kehitys nimittäin hidasti järjestelmää ja kehittäjien tehtyä koodiin olennaisia muutoksia sovellukseen oli aina kirjauduttava uudelleen sisään. Varsinkin tämän takia varsinaiset rasitustestaukset eivät olleet onnistuneita. Havaittu asiakashallinnan listauksessa esiintyvä huomattava hitaus oli rasitustesteistä varsinaisesti ainoa, joka kirjattiin virheraportteihin vakavana virheenä. Tämä vakava virhe jäi odottamaan, jotta se korjattaisiin ennen tuotantoon menoa.

Hylkäämistesteihin valittiin tekstikenttien arvoiksi mahdollisimman pitkiä syötteitä ja kokeiltiin kaikkia mahdollisia merkkejä. Tällä tavalla löydettiin tuhoisia virheitä, koska jotkut kentät hyväksyivät aluksi pidempiä tekstejä kuin tietokantaan mahtui. Sovellus myös hyväksyi pakolliseksi määrättyyn kenttään arvon nolla, mutta tallennettaessa se jätti kentän tyhjäksi ja näin tietokantaan jäi arvoksi `null`.

Hylkäämistesteissä valittiin päiväyskenttiin virheellisiä arvoja, kuten `0.1.1001` ja `32.12.9999`, jotka saivat palvelinsovelluksen joissakin tapauksissa kaatumaan. Mikäli palvelinsovellus pystyikin käsittelemään nämä, niin replikoinnin jälkeen asiakassovellus useissa tapauksissa toimi virheellisesti. Kyseessä oli kuitenkin vasta sovelluksen ensimmäinen versio, joten sovittiin, että käyttäjiltä oletetaan täsmällisyyttä, eikä kaikkia vääriä arvoja tarkisteta.

7.1.5 Systeemin integraatiotestauksen tulokset

Vielä systeemin integraatiotestauksen aikana järjestelmän toiminnallisuutta muutettiin joiltakin osin. Tämä jatkuva muuttaminen aiheutti suuresti lisätyötä dokumentaatioiden ylläpidossa sekä joidenkin muuttuneiden toiminnallisuuksien huomioinnissa ja havaitsemisessa testauksen aikana. Systeemin integraatiotestauksen aikana oli myös suoritettava systeemitestaus kerran uudelleen, koska sovellukseen tuli olennaisia muutoksia. Varsinaisesta muutetusta osasta ei enää löytynyt virheitä, mutta muuten joitakin uusia virheitä havaittiin.

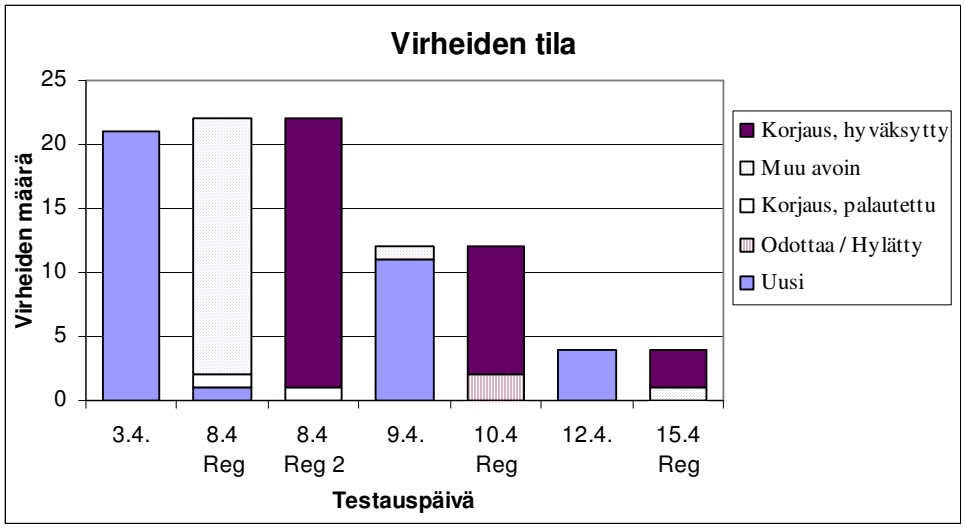
Integraatiotestauksen testikierros suoritettiin kolme kertaa täydellisenä. Yhteensä löydettiin 37 virhettä. Löydetyt uudet virheet jakautuivat vakaavuuksiltaan taulukon 4 mukaisesti.

Virheen vakavuus	**3.4.**	**8.4.**	**9.4.**	**12.4.**	**Yhteensä lkm**	**Yhteensä %**
Tuhoisa	3	1	0	0	4	11
Vakava	10	0	3	3	16	43
Häiritsevä	7	0	6	1	14	38
Siedettävä	1	0	2	0	3	8
Yhteensä	21	1	11	4	37	100

Taulukko 4. Uusien virheiden jakautuminen vakavuuden suhteen.

Löydetyistä virheistä hieman yli puolet kuului tuhoisiin tai vakaviin. Muutamat löydetyistä virheistä liittyivät uudelleen suoritettuun systeemitestaukseen. Osa niistä oli valitettavasti päässyt tähän vaiheeseen saakka. Osa virheistä oli mahdollista huomata vasta tämän testauksen aikana, jolloin järjestelmä oli kokonaisuudessaan käytössä.

Kuva 19 esittää virheiden tilojen muutoksen testauksen aikana. Testauskierros suoritettiin kokonaisuudessaan päivinä 3, 9 ja 12. Päivinä 8, 10 ja 15 suoritettiin regressiotestaus. 8.4. aloitettiin regressiotestauskierros, mutta heti alussa löytyi yksi tuhoisa virhe. Lisäksi kehittäjät olivat korjanneet tähän versioon tuhoisan virheen, joka silti sai järjestelmän kaatumaan edelleen. Tämän takia testaus keskeytettiin ja virheet palautettiin kehittäjille. Myöhemmin samana päivänä 8.4. päästiin regressiotestaus suorittamaan kokonaan.



Kuva 19. Virheiden tilan muuttuminen testauksen aikana.

7.1.6 Systeemin integraatiotestauksen tulosten analysointi

Systeemin integraatiotestauksessa virheiden määrä väheni tasaisesti testauksen loppua kohti. Kullakin testikierroksella löydettyjen virheiden määrä suunnilleen puolittui edelliseen kierrokseen nähden. Kuvasta 19 voidaan havaita, että yleensä regressiotestaukseen saatiin tarkastettavaksi lähes kaikki avoinna olleet virheet. Korjattuina palautetut virheet useimmiten toimivat ja ne voitiin hyväksyä. Virheiden elinkaaret (katso luku 3.4.3) olivat siis yksinkertaisia ja näin testaus eteni suhteellisen nopeasti.

Transaktioiden tiheys CRM-palvelimen ja sen asiakassovellusten välillä on suhteellisen alhainen, joten testauksen painopiste ei ollut suorituskyvyssä, vaan mieluummin asiaankuuluvien toiminnallisuuksien testaamisessa. Siirryttäessä testiaineiston käytöstä asiakkaan todelliseen aineistoon replikoitavan datan määrä lisääntyi huomattavasti, jolloin replikointi alkoi kestää liian kauan. Toiminnallisuutta jouduttiinkin muuttamaan niin, että replikoinnissa ei siirretä kuin toimimisen kannalta välttämätön tieto.

Testitapausten ulkopuolelta sovellettiin menetelmää, jossa testataan virheitä tekevän käyttäjän roolissa välittämättä ohjeista (katso luku 5.5.11). Tällä menetelmällä sovellus saatiin kaatumaan. Tokikaan kehittäjät eivät pitäneet näistä menetelmistä, sillä ne eivät kuulu tavanomaisen käyttäjän toimintaan. Toisaalta hyväkin käyttäjä tekee virheitä joskus vahingossa. Tällainen oli muun muassa testi, jossa CRM Client -sovelluksessa tuotetietojen tekstikenttä kirjoitettiin täyteen ja käytettiin normaalia enemmän skandinaavisia merkkejä. Replikoitaessa sovellus kaatui, koska skandinaaviset merkit veivät tallennusvaiheessa kahden merkin kokoisen tilan. Virhe korjattiin tässä alustavasti hyväksymällä tekstit, joissa skandinaavisia merkkejä on korkeintaan niiden normaalimäärä eli noin 20% koko tekstin pituudesta.

Löydetyistä 37 uudesta virheestä 7 liittyi aiempien vaiheiden testaukseen, mutta ne löydettiin vasta nyt. Loput 30 liittyivät replikoinnin ongelmiin. Moni näistä 30 virheestä johtui siitä, että tietokannat eri sovelluksissa eivät olleet yhtenevät. Tällöin replikoitaessa dataa ensimmäisen sovelluksen pitkästä kentästä toisen sovelluksen lyhyempään sovellus kaatui.

Muut replikoinnin virheet liittyivät siihen, että tietoa replikoitiin liian aikaisin tai joitakin tarvittavia kenttiä ei replikoitu lainkaan. Myös muutamien kenttien muokkaaminen ei virheellisesti ollut mahdollista enää ensimmäisen replikoinnin jälkeen.

7.2 Miten hyvin valitut menetelmät soveltuvat testaukseen?

Testaus toteutettiin dynaamisena mustalaatikkotestauksena. Tämä sopi tähän systeemitestausvaiheeseen ja sillä löydettiin virheet suhteellisen kattavasti. Staattista mustalaatikkotestausta olisi voitu soveltaa enemmän tuotteen määrittelydokumenttiin, jotta siinä olevat virheet ja puutteellisuudet olisi saatu korjattua. Lasilaatikkomenetelmää ei tässä vaiheessa olisi oikeastaan voinutkaan soveltaa.

Toteutettu testaus pohjautui pääosin testitapausten varaan. Testausta varten kirjoitetut testitapaukset kattoivat hyvin hyväksyntätestit, joilla osoitettiin sovelluksen toimivan tavanomaisimmissa tapauksissa. Testitapaukset kattoivat osittain myös hylkäämistestit, vaikkakin useimmat hylkäämistestit suoritettiin testitapausten ulkopuolelta. Testitapaukset toimivat hyvänä apuna tarkistettaessa näkymien ulkoasu ja muistuttivat kaikkien toiminnallisuuksien läpikäynnistä.

Hylkäämistestit sopivat testaukseen hyvin, koska niillä yleensä löydetään paljon virheitä. Tosin näissä testeissä on joitakin hankaluuksia. Usein kehittäjät eivät pidä olennaisina virheitä, jotka on löydetty hyvin harvoin ilmenevällä syötekombinaatiolla. Toteutetun testauksen aikana havaittiin monta toimintoa, jossa sovellus kaatui epätavallisia syötteitä annettaessa. Normaalikäytössä ihmiset antavat kyseisiä syötteitä yleensä vain vahingossa, mutta sovelluksen kaatavat virheet ovat kuitenkin aina tuhoisia. Tätä varten olisikin kullekin testaukselle hyvä sopia ja kirjata säännöt hylkäämistestien suorittamisesta ja siinä löydettyjen virheiden korjaamisesta.

Kaikkiaan virheiden löytämistä voidaan pitää onnistuneena, koska **löydettyjen virheiden vakavuus laski testauskierrosten edetessä**. Kaikkien kolmen testausvaiheen aikana tuhoisat virheet löydettiin testauksen alkupäivinä. Toki jos viimeisenä päivänä löydettäisiin monta tuhoisaa virhettä, testausaikaa jouduttaisiin varmasti pidentämään. Vakavienkin virheiden määrä laski huomattavasti testauksen edetessä. Viimeisimpien päivien virheet olivat lähinnä häiritseviä. Muun muassa CRM Client -sovelluksen systeemitestaus uusittiin vielä myöhemmin ja sen aikana löydetyt virheet olivat pääasiassa vain häiritseviä.

Testiaineiston tulisi olla riittävän kattava, koska kyseessä on systeemitestaus. Testauksessa CRM Client -sovelluksen testiaineiston luonti oli aikaavievää, eikä sitä sen takia tehty kovin perusteellisesti. Täten muutama tähän testaukseen liittyvä virhe löydettiin vasta systeemin integraatiotestauksen aikana, jolloin aineistoa oli riittävästi. Systeemin integraatiotestauksen aikana paljastui yhteensä seitsemän aiempien vaiheiden virhettä, mutta niistä vain kaksi oli vakavia.

Tämän tutkielman kirjoittamisen aikana tuli tutustuttua lähemmin muun muassa hylkäämistesteihin. Tätä uutta opittua oli kiinnostavaa kokeilla kehitettyyn sovellukseen, vaikka varsinainen testaus oli jo suoritettu. Erityisesti hylkäämistesteissä paljastui vielä muutama sovelluksen kaatava toiminta. Sovellus saatiin kaatumaan muun muassa antamalla sovelluksen aikakenttään kirjaimia. Nämä virheet jäivät aiemmin huomaamatta, mutta ne korjataan sovellukseen myöhemmin. Hyvänä asiana voidaan pitää, että tutkielma on opettanut uutta ja nyt tunnetuilla menetelmillä löydetään enemmän virheitä.

7.3 Mitä kannattaisi tehdä toisin teorian pohjalta?

Ohjelmistotestausta käsitteleviin lähteisiin tutustuminen on antanut uusia näkökulmia testauksen toteuttamiseen. Erityisesti verrattaessa teorian ohjeita ja käytännössä toteutetun testauksen tuloksia voidaan huomata, että tiettyjä käytettyjä testaustapoja tulisi muuttaa. Tässä luvussa tarkastellaan testausperiaatteiden ja toteutustapojen muutoksia ja lisäyksiä, joita seuraavissa projekteissa kannattaisi hyödyntää.

Ensinnäkin testausta varten tulisi piirtää koko sovelluksesta **tilakaaviot** (katso luku 5.5.9), joiden avulla voidaan huomioida sovelluksen toimivan oikealla tavalla. Tilakaavioita kannattaa käyttää luvussa 5.5.9 esiteltyjen polkutestien mukaisesti. Tutkielmassa esitetyssä käytännön testauksessa ei näitä kaikkia oltu kirjoitettu testitapauksiin. Testitapauksiin oli kirjoitettu lähinnä peruspolut läpikäyvät testit, joilla jo havaittiin puutteita. Lisäksi apuna oli tuotteen määrittelyssä kuvattu tilakaavio. Erityisesti tuotteen määrittelyn tilakaaviossa esitettyjä näkymien toiminnallisuuksia ei oltu aluksi ohjelmoitu vaatimusten mukaan. Testitapauksiin olisikin hyvä kirjoittaa ylös myös vähiten tavanomaisimpien ja virhetilanteita luovien polkujen läpikäynti.

Testauksessa tulisi tehdä **tarkempi jako ekvivalenssiluokkiin**. Varsinkin useamman testaajan projektissa kaikki testausta vaativat arvot olisi kirjattava ylös, jotta niitä varmasti muistetaan käyttää.

Tämän projektin testauksessa ei ensimmäisessä vaiheessa ollut virheraporteissa järjestysnumerointia eli ne piti tunnistaa vain otsikon avulla. Täten niiden käsittely oli työlästä tutkittaessa, mikä kunkin **virheraportin** tila oli. Järjestysnumeroinnin puuttuminen kuitenkin opetti käytännössä, kuinka teoriassa mainittua ohjetta kannattaa noudattaa. Muutenkin ensimmäisen vaiheen aikana käytetty virheraportti muokkaantui. Siihen lisättiin muun muassa virheiden korjaajien nimet kommunikointia helpottamaan. Luvussa 2.3 esitettyyn virheiden luokitteluun tutustumisen myötä raporttiin liitettiin kenttä, jossa arvioidaan virheen korjauksen kiireellisyyttä.

Virheiden raportointiin tulisi kiinnittää muutenkin enemmän huomiota. Raportin osa-alueista varsinkin virheiden luokittelu ja niiden kuvaus olisi tehtävä tarkemmin. Yhdessä suoritetun testauksen vaiheessa ei raportoitu lainkaan tuhoisia virheitä. Kuitenkin myöhemmin tarkastellen voi huomata, että osa virheistä olisi kuulunut tuhoisien virheiden luokkaan. Testaussuunnitelmassa olisi aina määriteltävä tarkkaan virheiden luokittelukriteerit, joita jokaisen testaajan tulisi noudattaa. Joillakin ihmisillä on tapana luokitella virheet liian vakaviksi. Jos testaaja usein luokittelee häiritsevän tasoisen virheen tuhoisaksi, eivät kehittäjät enää huomaa joukosta niitä todella kiireellisiä virheitä.

Yleisesti **V-mallin noudattaminen** auttaisi testausprosessia. Tosin noudattamisen onnistuminen on suuresti riippuvainen myös muun projektin aikataulun toimimisesta. Testausta kannattaisi pyrkiä suorittamaan enemmän ja järjestelmällisemmin jo kehitysvaiheessa käyttäen yksikkö- ja integraatiotestaukseen sopivia lasilaatikkomenetelmiä. Myös hyväksymistestausvaihetta olisi hyvä pyrkiä saamaan järjestäytyneemmäksi. Loppukäyttäjät voivat nimittäin havaita olennaisia käyttöön liittyviä asioita, joita kehitys- ja testiryhmä eivät havaitse. Tosin varsinaiset virheet sovelluksesta olisi kustannusten takia silti pyrittävä löytämään viimeistään systeemitestauksessa.

Luvussa 5.5.2 kehotettiin lisäämään määriteltyjen vaatimusten testausta eli staattista mustalaatikkotestausta. Määrittelydokumenttien testaus onnistuisi paremmin, jos määrittely olisi valmis silloin, kun testaus aloitetaan. Tässä projektissa vaatimukset muuttuivat koko ajan, joten dokumentaatiot eivät olleet ajan tasalla. Täten vaatimusten tarkka testaaminen ei ollut järkevää.

7.4 Olisiko voinut käyttää muita menetelmiä?

Tarkasteltavan järjestelmän testauksessa ei käytetty joitakin olennaisia systeemitestauksen piiriin liittyviä testaustapoja. Luvussa pohditaankin, kuinka muun muassa käytettävyys- ja joustavuustestaukset olisivat tuoneet hyvän lisän käytettyihin testausmenetelmiin.

Käytettävyys on tärkeä osa sovelluksen toimivuutta. Käytettävyystestauksessa (katso luku 4.3.2) olennaisina kriteereinä ovat loppukäyttäjien arviot sovelluksesta. Tulevat loppukäyttäjät saivat CRM Client -sovelluksen ensimmäistä kertaa kokeiltavaksi ensimmäisen kehitysvaiheen jälkeen. He kokeilivat sovelluksen toimintaa sekä kertoivat mielipiteitään ja muutosehdotuksia myöhemmin järjestetyssä tilaisuudessa. Koko järjestelmän he saivat kokeiltavaksi ja testattavaksi hyväksymistestausvaiheessa.

Varsinaisen systeemitestauksen osana ei suoritettu käytettävyystestausta, mutta käytettävyyteen kiinnitettiin kuitenkin huomiota testauksen aikana. Pyrkimyksenä oli rakentaa helppokäyttöinen ja tehokas systeemi. Käytettävyyteen liittyen raportoitiin joitakin virheitä. Muun muassa joidenkin listojen näyttäminen kesti liian kauan käyttäjän kannalta.

Näiden lisäksi olisi kuitenkin ollut hyvä tutkia käytettävyyttä laajemmin. Sovelluksen tulisi olla intuitiivinen ja eri toimintojen välillä tulisi olla helppo liikkua. Jotta sovellusta voidaan sanoa hyväksi käytettävyydeltä, tulisi testata sen tehokkuutta. Tätä varten olisi tarvittu tarkat arvot toivotuille toimintanopeuksille. Myös ohjetiedostojen testaus pitäisi sisällyttää testaukseen, jotta selvitettäisiin niiden olevan helposti käsitettäviä.

Käytettävyyttä testattaessa olisi hyvä saada loppukäyttäjiä mukaan. Näin sovelluksen käytöstä saataisiin loppukäyttäjienkin mielipiteet ja kokemukset näkyviin ennen hyväksymistestausta tai tuotantokäyttöä, jolloin sovelluksen muuttamisen kustannukset eivät ole vielä kohonneet todella korkeiksi.

Järjestelmän **joustavuutta** (katso luku 4.3.6) tullaan testaamaan varsin pian, kun CRM Clien -sovellus on siirrettävä toimimaan taskutietokoneen uusimmalla versiolla. Tällöin huomataan, kuinka paljon muutoksia tarvitaan siirryttäessä sovellusalustan seuraavaan versioon eli kuinka joustava kehitetty sovellus on. Tällä hetkellä CRM Server -sovellus toimii vaatimusmäärittelyn mukaisesti vain Internet Explorer -selaimella, mutta mahdollisesti se on myöhemmissä versioissa saatettava toimimaan myös Netscape -selaimella.

Näiden lisäysten jälkeen myös **kokoonpanotestaus** tulee olennaiseksi. Projektin tässä versiossa oli vaatimusmäärittelyssä rajattu sovelluksen toiminta käytettävissä oleviin kokoonpanoihin, koska kaikki loppukäyttäjät käyttivät vain näitä tiettyjä taskutietokoneen ja Internet Explorer -selaimen versioita. Laajennuksen jälkeen loppukäyttäjien kokoonpanot saattavat vaihdella.

Kolmas tärkeä osa-alue on **käyttöliittymätestaus** (katso luku 5.6.2), jota suoritetussa testauksessa olisi voitu toteuttaa enemmän. Testauksessa kiinnitettiin huomioita standardeihin, eli noudatettiinko yrityksen yleisiä ohjeita ja taskutietokoneen perusohjelmien ulkoasua. Toteutetun lisäksi olisi ollut hyvä käydä läpi kaikki näytöt ja selvittää johdonmukaisuus muun muassa painikkeiden samanlaisuutena sijoittelun suhteen. Testitapausten ulkopuolelta huomioitiin **johdonmukaisuuden** puute, koska eri kehittäjien koodaamat hakutoiminnot toimivat eri tavoin.

Käyttöliittymän **mukavuuteen** liittyy muun muassa se, että sovellus varoittaa ennen kriittisten operaatioiden tekemistä ja on riittävän nopea. Tämä ei toiminut kaikkialla, mutta joissain toiminnoissa yksityiskohtaisia varoituksia ei tässä ensimmäisessä versiossa katsottu aiheellisiksi. Kuitenkin kunnollisessa sovelluksessa tällaiset asiat tulisi huomioida, jos korjaamatta jättämistä ei voida perustella.

Käyttöliittymätestaukseen kuuluu myös **hyödyllisyyden** testaus. Kehitetty sovellus oli ensimmäisenä versiona lähinnä vain päätoiminnallisuudet sisältävä, joten kaikki toiminnot olivat hyödyllisiä. Tätä oli lisätty rajoittamalla eri käyttäjien käyttöoikeuksia vain heidän tarvitsemiinsa toimintoihin. Näin varsinkin uusien käyttäjien on helpompi tutustua sovellukseen.

7.5 Miten projektin koko vaikutti testaukseen?

Toteutettu projekti oli kooltaan pieni. Projektiryhmään kuului vain kuusi ihmistä. Tämä vaikutti osaltaan myös testauksen suorittamiseen. Tässä luvussa vertaillaan toimintatapoja, jotka riippuvat erityisesti projektin koosta, mutta eivät ole varsinaisesti riippuvia kehitetystä sovelluksesta.

Vertailtavien toimintatapojen erot ovat tulleet esille testattuani kahden todella erikokoisen projektin järjestelmiä. Tässä tutkielmassa esitellyssä projektissa oli vain yksi testaaja. Isossa projektissa testiryhmään kuului ryhmän manageri, johtaja ja projektin vaiheesta vaihdellen neljästä kahdeksaan testaajaa.

Testausmenetelmänä käytetty mustalaatikkomenetelmä soveltui yhtä hyvin tämän projektin systeemitestausvaiheeseen kuin se sopi isompaan projektiin. Tosin isommassa projektissa menetelmän soveltamisen toimintatapojen olisi oltava täsmällisempiä, yksityiskohtaisempia ja dokumentoidumpia.

Pienessä projektissa oli kaikkiaan vähän työntekijöitä, joten **kommunikointi** toimi hyvin. Yleensä aina tiesi, keneltä epäselviä asioita tulisi kysyä. Isommassa projektissa taas virheiden korjausprosessi oli joskus todella mutkikas. Virheet kulkivat kehittäjältä toiselle, eikä kukaan tuntenut olevansa vastuullinen korjauksesta. Kommunikointiin liittyy myös se, että isossa projektissa testaajien on toimittava järjestelmällisesti ryhmänä, jotta olennainen tieto välittyy kaikille testaajille. Pienessä projektissa tällaista ongelmaa ei esiinny.

Isossa projektissa on syytä kiinnittää erityistä huomiota **virheiden tarkkaan raportointiin**, jotta virhettä ei hylätä väärinymmärryksen takia. Pienessä projektissa on yksinkertaista käydä tarvittaessa kysymässä tarkennuksia testauksesta. Pienessä projektissa on myös mahdollista tarkkailla virheraporttien kulkemista ja huomioida, että virheet tulevat korjatuiksi. Isommassa projektissa on välttämätöntä käyttää työkalua virheiden käsittelyyn.

Tässä pienessä projektissa kaikki **testitulokset analysoitiin** manuaalisesti. Näistä kirjoitettiin testiraportti, johon laskettiin virheiden jakaantuminen ja niiden elinkaaren vaiheet. Virheiden luokittelussa ja tutkinnassa olisi hyvä olla apuna työkalu jopa pienessä projektissa. Suuressa projektissa ei selvittäisi pelkällä manuaalisella tavalla huolehtia virheiden hallinnasta.

Pienen projektin **testitapauksissa** ei tullut mainittua kaikkia **yksityiskohtia**, koska ne kirjoitettiin yhden henkilön testiryhmälle. Isossa projektissa testitapauksia käyttävät myös muut testaajat, joten yksityiskohdat on kirjattava tarkemmin, eikä mitään saisi jättää arvailujen varaan. Testitapausten tulisi olla yksiselitteisiä ja niin yksityiskohtaisia, että kuka tahansa voi vain ne ohjeenaan suorittaa testauksen vaiheet. Tällaisten testitapausten kirjoittaminen kuitenkin vaatii enemmän aikaa kuin pienessä projektissa on mahdollista käyttää.

Isossa projektissa **testiryhmän johtajan** vastuulla on kaikki hallinnolliset tehtävät ja testaajien tehtävänä on vain suorittaa testitapauksia. Pienessä projektissa taas kaikki yleiset tehtävät vievät aikaa testitapausten kirjoittamiselta ja testauksen suorittamiselta. Mutta toisaalta testaaminen on omatoimisempaa ja siksi sekä haastavaa että opettavaa.

Varsinkin pienen projektin testaamisessa ongelmana on siis se, että kaikkia vaiheita ei voi toteuttaa menetelmien edellyttämässä laajuudessa, vaan on pyrittävä karsimaan tehtävät vain olennaisimpiin. Lisäksi isossa projektissa testaajat voivat vaihtaa tehtäviään ja testata vaihdellen eri osa-alueita. Tunnettuahan on, että tällöin sovellusta katsotaan uudesta näkökulmasta ja saatetaan havaita taas uusia virheitä.

7.6 Johtopäätökset

Testauksen tavoitteena on löytää virheitä ja varmistaa, että ne tulevat korjatuiksi. Virheitä ovat kaikki määrittelyn vastaiset toiminnot. Testattaessa on kuitenkin oltava tarkkana, sillä määrittelyssäkin saattaa olla virheitä. Virheet olisi myös pyrittävä löytämään mahdollisimman aikaisin, jolloin niiden korjauskustannukset ovat mahdollisimman alhaiset ja todennäköisyys niiden korjaamiselle on suurin.

Usein testaus suoritetaan V-mallia käyttäen projektin vaiheiden mukaisesti. Tällöin määrittely ja suunnittelu tehdään ajoissa tarkasti. Testausta varten kirjoitetaan kattava suunnitelma, joka ohjaa testauksen aikataulua, suorittamista ja vaiheita.

Testaus jaetaan neljään vaiheeseen, jotka ovat yksikkö-, integraatio-, systeemi- ja hyväksymistestaus. Yksikkö- ja integraatiotestauksessa kehittäjät testaavat alemman tason toiminnallisuuden lasilaatikkomenetelmän avulla. Ainakin systeemitestauksen suorittaa erillinen testausryhmä käyttäen mustalaatikkomenetelmää. Hyväksymistestaus on asiakkaan suorittama testaus, jonka aikana loppukäyttäjät ovat yleensä mukana.

On hyvä tuntea testausmenetelmien teoriaa ja oppia soveltamaan sitä käytäntöön. Testausmenetelmien pitäisi kuitenkin olla käytössä projektin kaikissa vaiheissa. Tässä projektissa menetelmien käyttöä voidaan pitää onnistuneena, koska asiakas on saanut sovelluksen tuotantokäyttöön ja ollut siihen todella tyytyväinen. Toistaiseksi asiakas ei ole löytänyt virheitäkään lisää, vaan asiakkaalta saadut raportit ovat olleet vain muutosehdotuksia.

7.6.1 Tutkielman ja toteutetun testauksen aikana opittua

Projektin aikana teorian ja käytännön kautta opittiin, että testauksen suunnitteluvaiheessa on luotava **riittävän laaja testiaineisto**. Tällöin virheiden löytämisen siirtyminen myöhemmäksi ei ainakaan johdu testiaineiston riittämättömyydestä. Suoritettua testausta olisi myös auttanut, jos systeemitestauksessa olisi ollut käytössä testiympäristö, jolloin kehittäjien työ ei olisi haitannut testaamista ja päinvastoin.

Tässä projektissa sovitun työnjaon mukaisesti yksikkö-, integraatio- ja hyväksymistestaus olivat täysin erillään systeemitestauksesta. Yksikkö- ja integraatiotestauksen seuraaminen tai ainakin sen raporttien näkeminen auttaisi kuitenkin systeemitestausta. Näin testauksissa ei raportoitaisi päällekkäisiä virheitä ja aiemmista testauksista voisi saada vinkkejä systeemitestauksen suuntaa valittaessa. Hyväksymistestauksen asiakas hoiti omatoimisesti ja ainoastaan alussa testiryhmä oli mukana auttamassa testausta käyntiin.

Projektissa toteutettu yksikkö- ja integraatiotestaus perustui lähinnä kehittäjien kokeiluihin koodin toimivuudesta. Testausta varten ei kirjoitettu suunnitelmaa, sen aikana ei raportoitu virheitä ja mitään varsinaista menetelmää ei käytetty. Testauksessa huomioitiin toiminnan ja ulkoasun oikeellisuutta. Testaus toteutettiin pääasiassa normaaleilla arvoilla, mutta syötteinä kokeiltiin jonkin verran myös raja-arvoja ja epäkelpoja syötteitä. Täten on kuitenkin ymmärrettävää, että näiden vaiheiden **testaus tulisi tulevissa projekteissa toteuttaa järjestelmällisemmin**.

Mustalaatikkomenetelmää kannattaa soveltaa koko laajuudessaan. Tällöin menetelmän käyttö kattaa testaussuunnitelman ja testitapausten kirjoittamisen, jotka sitten antavat suunnan testauksen toteuttamiselle. Tämän projektin tulokset osoittavat, että pelkällä mustalaatikkomenetelmän soveltamisella päästään hyviin tuloksiin. Virheiden kustannukset kuitenkin kohoavat ohjelmistoprojektin edetessä, joten testauksen suorittaminen edes osittain lasilaatikkomenetelmää käyttäen saattaisi aiheuttaa vähemmän kustannuksia.

Mustalaatikkomenetelmää sovellettaessa **kannattaa testiaineisto valita tarkemmin** kuin tässä testauksessa tehtiin. Erityisesti aluksi kannattaa tehdä aineiston jako ekvivalenssiluokkiin, huomioida luokkien raja-arvot ja ottaa testaukseen arvoja rajojen molemmilta puolilta. Testiaineistoon tulee myös valita hylkäämistestejä varten syötteet, joita ovat muun muassa tyhjät ja liian pitkät arvot. Näiden lisäksi tietovirtatestaus kannattaa ottaa mukaan käyttöön. Kaikki testaukset ja syötteet kannattaa kirjoittaa tarkasti testitapauksiin, koska useimmissa projekteissa on monta testaajaa. Projekteissa ei siis voida luottaa siihen, että muut osaisivat automaattisesti lisätä olennaiset asiat kirjoitettuihin testitapauksiin.

Testauskirjallisuudessa todettiin monta kertaa, että sovellukset eivät koskaan ole virheettömiä. Tämä huomattiin käytännössä, koska osa testauksesta suoritettiin muutosten takia myöhemmin uudelleen. Tämän uudelleentestauksen aikana aina löydettiin muutamia virheitä, jotka periaatteessa olisi pitänyt löytää aiemmin. Sovellukseen jäävistä virheistä todistaa myös se, että tutkielmaa kirjoitettaessa kokeiltiin testausta antamalla syötteenä erikoistapauksia ja löydettiin vielä muutama virhe.

7.6.2 Jatkotutkimuksen kohteita

Tulevissa projekteissa kannattaa mustalaatikkomenetelmää hyödyntää järjestelmällisemmin koko systeemitestausvaiheessa aina suunnitelmien kirjoittamisesta toteuttamiseen asti. Periaatteessa tätä menetelmää voidaan käytännössä pyrkiä soveltamaan sovelluksen seuraavassa versiossa syksyllä.

Mustalaatikkomenetelmä on monipuolinen ja kattaa sovelluksen testauksen laajasti. Sen käyttö ei kuitenkaan kokonaisuudessaan onnistu näin pienessä projektissa. Isommassa projektissa päästään kokeilemaan rasitus- ja suorituskykytestausta laajemmin sekä käyttämään muun muassa kokoonpano- ja tietoturvatestejä.

Tutkielmassa käsiteltiin laajahkosti myös lasilaatikkotestausta. Sitä ei kuitenkaan päästy kokeilemaan käytännössä. Nyt kun tämän testauksen tulokset ovat tiedossa, olisi mielenkiintoista soveltaa lasilaatikkomenetelmää yksikkö- ja integraatiotestauksen vaiheissa ja tarkkailla, miten systeemitestausvaiheen testaus ja tulokset muuttuisivat.

Olennainen jatkotutkimuksen kohde olisi myös syventyä testaustyökalujen käyttöön, sillä ne helpottaisivat monia tehtäviä. Erityisesti regressiotestausvaiheessa sekä raporttien kirjoittamisessa ja jäsentelyssä nämä olisivat tarpeellisia.

8 Yhteenveto

Tutkielman alussa käsiteltiin testauksen teoriaa, testauksen määrittelyä, tärkeyttä ja sijoittumista ohjelmistoprojektiin. Seuraavaksi käsiteltiin testauksen jakautumista ohjelmistoprojektin mukaisesti eteneviin testaustasoihin, joita ovat yksikkö-, integraatio-, systeemi- ja hyväksymistestaus. Erityisen mielenkiinnon kohteena olivat testausmenetelmät. Näistä ensin käsiteltiin lasilaatikkomenetelmää, mutta pääpaino oli systeemitestausvaiheen aikana käytetyn mustalaatikkomenetelmän tarkastelussa.

Tutkielman käytännön osuus muodostui sovellusprojektina kehitetyn taskutietokone-sovelluksen testauksesta. Tarkasteltu testaus toteutettiin systeemitestausvaiheessa mustalaatikkomenetelmää hyödyntäen. Valittua menetelmää voidaan pitää hyvänä, koska asiakas on kehitettyyn sovellukseen tyytyväinen ja loppukäyttäjät eivät ole löytäneet sovelluksesta uusia virheitä. Testauksen toteuttamista ja menetelmien käyttöä tulisi kuitenkin tehostaa seuraavissa projekteissa.

Ohjelmistotestauksen teoria ja toteutettu käytännön projekti muuttivat tekijän näkökulmaa testauksen toteutukseen. Seuraavaa sovellusta testattaessa jotkut asiat kannattaa tehdä toisin. Erityisesti esiin nousee systeemitestausta edeltävien vaiheiden järjestelmällisempi toteuttaminen, koska niissä virheiden korjauskustannukset ovat halvemmat.

Yksikkö- ja integraatiotestausvaiheissa menetelmänä kuuluisi olla lasilaatikkotestaus. Toteutetussa projektissa näiden vaiheiden testaus jäi kuitenkin kehittäjien omien arvioiden varaan, eikä mitään menetelmää tietoisesti käytetty. Testauksen järjestelmällisyyden kasvattaminen näissä vaiheissa on eräs tehtävä, jolla ohjelmistoprojektia ja kehitettävän järjestelmän laatua saadaan parannettua.

Tärkeinä tekijöinä testauksen onnistumiselle voidaan pitää järjestelmällisten menetelmien käyttöä. Testaus on suunniteltava ajoissa ja huolellisesti. Testaus-suunnitelman tulee kattaa testaus monipuolisesti. Systeemitestauksen toteutusvaiheessa mustalaatikkomenetelmää on hyödynnettävä mahdollisimman tarkasti ja monipuolisesti. Erityisesti tulee kiinnittää huomiota testiaineiston valintaan ekvivalenssiluokkien ja raja-arvojen avulla, tietovirtakaavioiden käyttöön ja hylkäämistestien hyödyntämiseen.

Testauksen onnistumiseen liittyy tietysti myös sille annettu arvostus. Kun testauksen suunnittelulle annetaan riittävästi aikaa ja voi olla huolellinen koko testausprosessin ajan, päästään hyviin tuloksiin. On kuitenkin huomioitava, että testattaessa ei kaikkia virheitä löydetä systemaattisin menetelmin, vaan aina osa löytyy luovan kokeilemisen avulla.

Tutkielman antina lukijalle on monipuolinen kuva testauksesta. Tämä pohjautuu ensinnäkin lähdekirjallisuuden antamalle teorialle ja toiseksi käytännössä toteutetun sovelluksen testaukselle. Tämä uusi tieto auttaa toivottavasti ymmärtämään testauksenkin olevan olennainen ja välttämätön osa ohjelmistoprosessia sekä nostamaan testauksen arvostusta.

Tutkielman tekeminen on opettanut paljon systemaattista työskentelyä. Näin perusteellinen tutustuminen yhteen ohjelmistotuotannon osa-alueeseen antaa varmasti mahdollisuuksia suunnata työskentelyä testauksen alueelle ja antaa sovelluskehitykselle uutta näkökulmaa.

Tulevana projektina olisi varmasti hyvä tutkia työkalujen hyödyntämistä testauksessa, koska ne voisivat olennaisesti vähentää rutiininomaisten vaiheiden toistoa. Tietysti kiinnostavaa olisi päästä toteuttamaan testausta myös lasilaatikkotestauksen menetelmin ja oppia senkin osa-alueen toteuttamista käytännössä.

Testauksen teorian pohjalta tiedetään, että valitettavasti sovelluksiin aina jää virheitä. Näiden määrä ei yleensä ole kovin suuri etenkin, jos testaus on suoritettu perusteellisesti käyttäen järjestelmällisiä menetelmiä. Kuitenkin kaikkien teoriassa mainittujen testausmenetelmien soveltamisen jälkeen jää tilaisuus pyrkiä vielä parempaan tulokseen. Lopuksi testaajan on siis syytä ottaa käyttöön luovat ideat ja mielikuvitus, jotta sovelluksesta löydettäisiin vielä yksi virhe!

Lähteet

Beizer Boris, “Software System Testing and Quality Assurance”, Van Nostrand Reinhold Company, 1984.

The BCS SIGIST Standards Working Party, “Reliability Testing”, saatavilla HTML-muodossa <URL: http://www.testingstandards.co.uk/ReliabilityTesting23_4_99.htm>, The BCS SIGIST Standards Working Party, luettu 30.4.2002.

Drabick Rodger, “Growth of Maturity in the Testing Process”, saatavilla HTML-muodossa <URL: http://www.softtest.org/articles/rdrabick3.htm >, International Software Testing Institute, 1999.

Gao Jerry, “Software Testing and Strategies”, saatavilla pdf-muodossa <URL: http://www.engr.sjsu.edu/gaojerry/course/ise165/Handouts/Test-strategy.pdf>, San Jose State University, College of Engineer, 2000.

Haikala Ilkka ja Märijärvi Jukka, ”Ohjelmistotuotanto”, Talentum Media Oy, Vantaa, 2002.

IPL Information Processing Ltd, ”Designing Unit Test Cases”, saatavilla pdf-muodossa <URL: http://www.iplbath.com/pdf/p0829.pdf>, IPL Information Processing Ltd, 1996.

Kankaanpää Timo, ”Testaus”, saatavilla HTML-muodossa <URL: http://www.tec.puv.fi/~tka/kurssit/Tietojarjestelmien_suunnittelu/Testaus.doc >, Vaasan ammattikorkeakoulu, 1998.

Kautto Tuomas, ”Ohjelmistotestaus ja siinä käytettävät menetelmät”, Ohjelmistotekniikan seminaari, saatavilla HTML-muodossa <URL: http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/testaus >, Jyväskylän yliopisto, Tietotekniikan laitos, 1996.

Koikkalainen Pasi, ”Ohjelmistotuotanto-kurssi”, luentomateriaali, Jyväskylän yliopisto, Tietotekniikan laitos, 2000.

Marick Brian, “New Models for Test Development”, saatavilla pdf-muodossa <URL: http://www.testing.com/writings/new-models.pdf>, 1999.

Marick Brian, “How to Misuse Code Coverage”, saatavilla pdf-muodossa <URL: http://www.testing.com/writings/coverage.pdf>, 1997.

Melzer Ingo, “Testing of a Computer Program on the Example of a Medical Application with Diversification and Other Methods”, saatavilla HTML-muodossa <URL: http://www.mathematik.uni-ulm.de/~melzer/thesis/ >, University of Ulm, Faculty of Computer Science, 1996.

Paakki Jukka, ”Ohjelmistojen testaus”, luentomoniste, Helsingin yliopisto, Tietojenkäsittelytieteen laitos, 2000.

Patton Ron, “Software Testing”, SAMS Publishing, USA, 2001.

Pressman Roger, “Software Engineering: A Practitioner’s approach”, McGraw-Hill, 1997.

Santanen Jukka-Pekka, "Ohjelmistotuotanto ja projektityö", luentomateriaali, Jyväskylän yliopisto, tietotekniikan laitos, 1997.

Sommerville Ian, “Software Engineering”, Addison-Wesley, 2001.

Liitteet

Liite 1. Tutkimuksessa käytetty virheraporttiesimerkki.

Bug Report

Number:
Date:
Requester:

Title:

Seriousness: 1. Fatal/2. Serious/3. Disturbing/4. Tolerable
Priority: 1. / 2. / 3. / 4.
Detected during: System test client/ System test server/ System integration
Version:
Reproducibility: Yes/No
Steps to reproduce:

Expected Results:

Actual Results:

Comments:

Fix

Date Completed:
Completed by :
Solution:

Retest

Date Tested:
Tested by :
Completed: Yes / No

Liite 2. Esimerkkiohjelma kattavuusmittojen tarkasteluun.

Tässä on esitetty lyhyt esimerkkiohjelma, jota avulla voidaan käyttää kattavuusmittoja tarkasteltaessa.

```
1. void paivays( int kk, int pv){
2. if ( kk=2 and pv=29) {
3.     println("On karkauspäivä. "); }
4. println("Tänään on " + pv + "." + kk + ".");
5. }
```

Taulukossa selvitetään esimerkkiohjelman testauksessa tarvitut testitapaukset kunkin kattavuusmitan tapauksessa.

kk	**pv**	**Ehdon totusarvo**	**Suoritettavat rivit**
Lausekattavuus			
2	29	Tosi	1,2,3,4,5
Päätöskattavuus			
2	1	Epätosi	1,2,4,5
2	29	Tosi	1,2,3,4,5
Ehtokattavuus			
1	29	Epätosi	1,2,4,5
2	1	Epätosi	1,2,4,5
Moniehtokattavuus			
2	29	Tosi	1,2,3,4,5
1	1	Epätosi	1,2,4,5
2	1	Epätosi	1,2,4,5
1	29	Epätosi	1,2,4,5