Ohjelmistojen testaus ja virheenjäljitys

Hannu Virkanen
Pro gradu –tutkielma
Tietojenkäsittelytiede
Kuopion yliopisto
Informaatioteknologian ja
Kauppatieteiden tiedekunta
Joulukuu 2002

TIIVISTELMÄ

KUOPION YLIOPISTO, informaatioteknologian ja kauppatieteiden tiedekunta
Tietojenkäsittelytieteen koulutusohjelma
Tietojenkäsittelytiede

VIRKANEN HANNU, S: Ohjelmiston testaus ja virheenjäljitys
Pro gradu -tutkielma: 77 sivua
Pro gradu -tutkielman ohjaaja: Anne Eerola, FT
Joulukuu 2002

Avainsanat: ohjelmiston testaus, automatisoitu testaus, testaustyökalut, testitapaus

Tutkielmassa esitellään ohjelmistotestauksen teoriaa ja virheiden jäljittämistä ohjelmistoissa kaikilla ohjelmistotuotantoprosessin tasoilla. Teoriaosuudessa selvitetään testauksen käsitteistöä, menetelmiä, testausta ohjelmistotuotannon eri vaiheiden mukaan ja testaustyökalujen käytön kannalta. Suurinta osaa esitellystä teoriasta havainnollistetaan soveltamalla teoriaa tutkielman loppuosassa käytännön ohjelmistotuotannossa vastaan tulleilla ongelmatilanteilla esimerkkisovellusten ja niille suoritetun testauksen ja virheenjäljityksen avulla.

Tutkielmassa on perehdytty ohjelmistotuotannon tuottavuutta parantaviin menetelmiin ja käyttökohteisiin. Ohjelmistojen testauksella pyritään löytämään virheitä mahdollisimman kustannustehokkaasti ja testauksella havaitun virheen aiheuttajan löytämistä pyritään helpottamaan virheenjäljitysmenetelmien avulla. Testausta suoritetaan eri tavoin kaikissa ohjelmistontuotantoprosessin vaiheissa ja testauksen kohde vaihtelee prosessin vaiheiden mukaan. Komponentin sisäistä toimintaa ja sen oikeellisuutta tarkastellaan ohjelmiston komponenttitestausvaiheessa. Ohjelmiston integrointitestauksessa testauksen kohteena ovat komponenttien välisten rajapintojen toiminta. Järjestelmätestauksessa tarkastellaan ohjelmiston toimintaa ja ohjeistusta kokonaisuutena. Itse ohjelmiston testaus tässä vaiheessa suoritetaan sovelluksen käyttöliittymän kautta. Kuhunkin menetelmään, lähestymistapaan ja ohjelmistotuotannon vaiheeseen on perehdytty esimerkkisovelluksen, testitapauksen ja vaiheeseen soveltuvan työkalun käytön kautta. Tutkielmassa esitetyt testitapaukset pohjautuvat ohjelmistotalon sisällä ohjelmistotuotantoprosessissa esiin tulleisiin käytännön testitapauksiin ja niiden suunnittelu- ja määrittelyprosesseihin.

Testauksen teoria testityökalujen osalta on kehittymässä, kuitenkin usein aihetta koskevassa kirjallisuudessa annetaan vain suosituksia ja tehdään havaintoja työkalujen käytöstä ja käyttökohteista. Teoriaa sovelletaan testityökalujen käytön osalta usein siten kuin testityökalu olisi sovellusta testaava ihminen. Kuitenkin päädyttäessä testityökalujen käyttöönottoon sovelluksen testauksessa ja haluttaessa varmentaa työkalun täydellinen hyödyntäminen, on tehtävä muutoksia koko ohjelmistontuotantoprosessiin ja sovelluksen suunnitteluperiaatteisiin. Menetelmien, lähestymistapojen ja apuvälineiden ohella tutkielmassa on esitetty näkemyksiä siitä, millaisia muutoksia edellä kuvatun kaltainen ohjelmistotuotannon tehostamisprosessi vaatii.

ESIPUHE

Tämä pro gradu –tutkielma on tehty Kuopion yliopiston tietojenkäsittelytieteen ja sovelletun matematiikan laitokselle vuonna 2002.

Haluan kiittää tutkielman ohjaajaa FT, professori Anne Eerolaa ohjeista ja maltista tutkielman valmistumisen kanssa. Samoin kiitokset kuuluvat Tietosavo Oyj/Tekla Oyj:lle ja Kuopion yliopiston atk-keskuksen HIS-tutkimusyksikölle mahdollisuudesta työskennellä ja tutustua tutkielmassa esitettyihin välineisiin.

Kiitokset Minnalle avusta ja tuesta.

Kuopiossa 15.12.2002

Hannu Virkanen

SISÄLLYS

1. JOHDANTO

Ohjelmistoprojekteille on asetettu resurssit, joiden puitteissa myös testaus joudutaan suorittamaan: mahdollisimman taloudellisesti, mutta samalla ohjelmiston kokonaislaadusta tinkimättömästi. Ohjelmistojen koon kasvaessa ja niiden toimintojen monipuolistuessa testaus muuttuu jatkuvasti vaikeammaksi suorittaa täysin kattavasti. Testauksen hallinta ja sen kautta laadunvarmistus vaatii enemmän ja enemmän resursseja ohjelmistontuottajalta. Eräs tapa parantaa ohjelmistontestauksen kustannustehokkuutta on ottaa käyttöön oikeita menetelmiä ja apuvälineitä parantamaan testausprosessin laatua ja silti samalla kyetä vähentämään siihen käytettyä työpanosta. Tutkielmassa perehdytään ohjelmistotuotannon tuottavuutta parantaviin menetelmiin ja niiden käyttökohteisiin, joista erityisen tarkastelun kohteena on testauksessa käytetyt apuvälineet automatisointi ja testaustyökalut. Oikein kohdennettu testaus tulee suorittaa ohjelmistotuotannon eri tasoilla rajaamaan ohjelmistossa esiintyvien virheiden vaikutukset mahdollisimman vähäisiksi. Mitä myöhemmin virhe löydetään, sitä kalliimmaksi sen korjauksen toteuttaminen tulee.

Testaus on systemaattinen lähestymistapa ohjelmistoissa esiintyvien virheiden löytämiseksi ohjelmaa suorittamalla. Testattaessa pyritään luomaan kattavia testitapausjoukkoja ja löytämään menetelmiä, joilla ohjelmissa esiintyvät virheet saadaan selvitettyä taloudellisesti ja ohjelman toiminta mahdollisimman luotettavaksi. Ohjelmien testauksen kokonaisuuden hallintaan on tehty erilaisia apuvälineitä: testaustyökaluja ja testausprosessin hallintaohjelmistoja.

Tutkielmassa luodaan katsaus eri menetelmiin, joilla testausta suoritetaan, ja kuvataan testauksen toteutusta ohjelmistotuotannon eri tasoilla. Ohjelmointityökalujen ja -kielien kehittyminen on muodostanut uudenlaisia virhetapauksia verrattuna perinteiseen ohjelmointiin, mikä on aiheuttanut myös uudenlaisia vaatimuksia jo olemassa oleville testausmenettelyille. Käytettävät testaustyökalut pohjautuvat testauksesta luotuun ja jatkuvasti luotavaan teoriaan. Testaukseen liittyviä uusia piirteitä ja vaatimuksia tuodaan esiin perinteisten menetelmien ohella.

Testauksen tarkoituksena periaatteessa on osoittaa, että ohjelma toimii tai toisin päin, että ohjelma ei toimi, kuten sen pitäisi. Testausta suoritettaessa on hyväksyttävä se tosiasia, että kaikkia ohjelmistossa esiintyviä virheitä ei kyetä saamaan esiin, johtuen ohjelmistojen laajuudesta ja suorituspolkujen moninaisuudesta. Ohjelman toiminta voidaan todistaa oikeel-

liseksi vain äärimmäisen triviaaleissa tapauksissa. Oleellista testauksessa onkin valita kuhunkin testitapaukseen sopivat ja mahdollisimman paljon tärkeitä virheitä paljastavat työkalut ja menetelmät. Itse testaus ei tule parantamaan ohjelmaa tai sen laatua millään tavalla, vaan se kertoo, mitä puutteita ohjelmassa on. Ohjelmiston testaus paljastaa ohjelmiston toteutuksen tai suunnitteluvaiheen aikana tulleet virheet ja antaa välillisesti ohjeet ohjelmiston laadun parantamiseen.

Testaamisen voi nähdä yksinkertaisimmillaan ohjelman tai sen osan ajamisena virheitä etsien. Testin kohteen toimintaa ja sen toiminnan oikeellisuutta tarkastellaan antamalla sovellukselle syötteitä ajon aikana, jonka jälkeen sovelluksen toimintaa tarkastellaan sen tuottamien vasteiden kautta. Muita tarkasteltavia asioita testauksessa ovat esimerkiksi: ohjelmiston suorituskyky, käytettävyys, luotettavuus ja toiminta virhetilanteissa. Testausta tapahtuu ohjelmistotuotannon jokaisella tasolla aina sovellukselle tehtyjen määritysten testaamisesta valmiin lopputuotteen hyväksymistestiin. Testaus työnä vaihtelee dokumenttien tarkastelusta sovelluksen loppukäyttäjän toiminnan simulointiin. Testausta joutuvat suorittamaan niin itse koodia tuottavat ohjelmoijat, ohjelmistotalon sisällä omat testaajat kuin tarvittaessa myös sovelluksen loppukäyttäjät. Testaaminen kulkee ohjelman koko elinkaaren ajan sen mukana ja testaus toimii ohjelmistotalon laadunvarmistuksen perustana.

Testattaessa ohjelmistoa tulisi testin paljastaa, selvittää ja tuoda vastauksia toteutetusta ohjelmakoodista esille tuleviin kysymyksiin:

Toimiiko ohjelma väärin suhteessa määrittelyynsä?

Oikein toteutetuista määrittelyistä on helppo luoda tarpeeksi yksityiskohtainen testaussuunnitelma ja laaja testitapausten joukko. Saatujen tulosten perusteella vastaaminen kysymyksiin, toimiiko ohjelma tai ohjelman jokin ominaisuus oikein, on yksinkertainen tehtävä, johon voidaan vastata kyllä tai ei. Oikein ja tarpeeksi yksityiskohtaisesti toteutetut määrittelyt helpottavat myös ohjelmistojen toteuttajien toimintaa ja vähentävät väärinkäsitysten mahdollisuutta.

Toimiiko ohjelma eri ympäristöissä?

Nykyisin ohjelmakoodit joutuvat toimimaan hyvin monenlaisissa laitteistoympäristöissä ja muiden ohjelmistojen kanssa vuorovaikutuksessa. Ohjelmiston testauksessa ei riitä, että ohjelmisto toimii oikein erillisessä testausympäristössä, vaan testattaessa tulee myös varmistaa toiminta sen oikeassa ympäristössä muiden samoista resursseista riippuvaisten ohjelmien ja moninaisten oheislaitteiden ja esimerkiksi niiden ajurien kanssa. Myös erilaisten laitealustojen, käyttöjärjestelmien ja niiden versioiden lisääntyminen voi vahingoittaa ohjelman toimintaa. Edellä kuvattuja ongelmia on pyritty vähentämään erilaisten arkkitehtuurien ja rajapintojen standardoinnilla, mutta virhetilanteiden mahdollisuus kasvaa ohjelmiston toimintaan vaikuttavien komponenttien lisääntyessä.

Tekeekö ohjelma sitä, mitä sen pitäisi?

Ohjelmiston toiminnan oikeellisuuden testaus on tehtävä toiminnallisia määrittelyjä vasten. Usein virheet tapahtuvatkin määrittelyä tehtäessä. Ohjelmiston toteuttaja ei ole ymmärtänyt asiakkaan tarpeita tai asiakas ei ole kyennyt tuomaan tarpeitaan oikein esille, joka tapauksessa ohjelmisto ei tyydytä sen käyttäjän tarpeita. Tilanteen välttämiseksi tulisi kyetä testaamaan myös määrittelydokumentit niiden oikeellisuuden varmistamiseksi. Ohjelman toiminnan ollessa jotain muuta kuin sen pitäisi, voi kyseessä olla myös yksinkertainen ohjelmointivirhe, mutta yleisimmin ohjelmointivirheet ovat havaittavissa ennemminkin ohjelman kokonaisena toimimattomuutena tai virhetoimintoina, kuin vääränlaisina toimintoina.

Tekeekö ohjelma sitä mitä sen ei pitäisi tehdä?

Monimutkaisten ohjelmakoodien implementoinnissa, esimerkiksi käytettäessä erinäisiä valmiita komponentteja ja luokkia voi ohjelmakoodiin tulla mukaan huomaamattomia riippuvuuksia toisista ohjelmistokomponenteista. Ohjelmaa ajettaessa koodi tekee kaiken, mitä sen pitääkin, mutta sivuvaikutuksena ohjelma voi suorittaa myös toimintoja, jotka voivat olla ylimääräisiä ohjelmalle tehdyille toiminnallisille määrittelyille. Ylimääräiset toiminnot voivat olla itse ohjelman kannalta merkityksettömiä, suoranaisesti haitata itse ohjelman suoritusta tai välillisesti jopa vaikuttaa ohjelman toimintaan haitaten ohjelman kanssa yhdessä toimivan laitteiden tai ohjelmistojen toimintaa. Ohjelmoitaessa ei kuitenkaan voida tai edes tulisi tehdä kaikkea itse, vaan on järkevää turvautua kolmansien osapuolien toimit-

tamiin komponentteihin ja laitteiston osiin, joiden toimintaan ei tulisi kuitenkaan luottaa täysin sokeasti, vaan myös valmiina saatujen osa-alueiden toiminta tulee varmistaa järjestelmää integroitaessa.

Tutkielman ensimmäisessä osiossa esitellään ohjelmiston testauksen teoriaa: tutustutaan testauksessa käytettävään käsitteistöön, eri testausmenettelyihin ja menetelmien valintoihin vaikuttaviin tekijöihin ja testausmuotoihin eri ohjelmistotuotantoprosessin vaiheissa. Tutkielman jälkimmäisessä osiossa esitellään erilaisia testaustyökaluja ohjelmistotuotantoprosessin vaihejaon mukaisella ryhmittelyllä. Työkalujen toimintaa havainnollistetaan esimerkein, eli testityökaluja käytetään niitä varten tehdyn sovelluksen testaukseen. Esimerkkitestitapauksilla tutkielman alussa esitettyä teoriaa sovelletaan käytäntöön testityökalujen toimiessa sovelluksen testauksen apuna.

2. TESTAUKSEN KÄSITTEET

Seuraavassa kappaleessa tullaan käsittelemään ohjelmiston testauksessa vakiintuneita käsitteistöä ja eri käsitteiden välisiä riippuvaisuuksia. Käsitteistön pohjana on käytetty BSI:n (British Standards Institution) standardeja Software testing - Part 1 [BSI98a] ja Software testing - Part 2 [BSI98b], jotka ovat yksi standardisoitu määrittely testauksessa käytetylle termistölle. Käsitteistöä ovat olleet luomassa mm. British Computer Society, IEE (Institution of Electrical Engineers), National Computing Centre Ltd. ja Britannian puolustusministeriö. Testauksen ollessa osa-alue ohjelmistotuotannon laadunvarmistusprosessia, muita testauksen käsitteiden kanssa käytettäviä BSI:n standardisoituja termistöjä ovat Quality Vocabulary - Part 2: Quality concepts and related definitions, Data Processing-Vocabulary-Part 1: Fundamental terms ja Information technology - Software life cycle processes.

2.1 Virhe

Virhe (error, fault, failure) eli virhetoiminto määritellään havaittuna eroavaisuutena ohjelmiston toiminnassa ja sille tuotetuissa määrityksissä. Virheet voidaan jakaa esimerkiksi kääntäjän kannalta katsoen kolmeen erityyppiseen kategoriaan: käännöksenaikaisiin, ajonaikaisiin ja loogisiin virheisiin. Toinen tapa jakaa virheet on suorittaa luokittelu testauksen vaihemallin avulla: virheelliseen vaatimusten määrittelyyn, suunnitteluvirheisiin ja ohjel-

mointivirheisiin, joihin tutustutaan tarkemmin testauksen vaihejakoon perehtymisen yhteydessä. Käännöksenaikainen eli kielioppivirhe syntyy esimerkiksi, kun muuttujaa ei ole esitelty, aliohjelman kutsussa on väärät parametrit tai kun kokonaislukumuuttujaan sijoitetaan reaaliarvot eli kun kirjoitettu ohjelmakoodi on ohjelmointikielen syntaksin vastaista. Virheiden löytyminen käännöksen aikana riippuu siitä kuinka tarkat säännöt ohjelmointikieleen tai kääntäjälle on määrätty. *Käännöksenaikaiset virheet* syntyvät ohjelman toteutusvaiheessa ohjelmoijan tehdessä virheellistä ohjelmakoodia. *Ajonaikainen virhe* eli *semanttinen virhe* syntyy kun virheettömästi käännetty ohjelma yrittää esimerkiksi lukea olemattomasta tiedostosta tai jakaa luvun nollalla. Ajonaikaisten virheet ilmenevät, kun ohjelmakoodi on ohjelmointikielen syntaksin mukaista, mutta se yrittää suorittaa toimintoja, joita ei ole mahdollisia toteuttaa laitteen sen hetkisessä tilassa. Useimpiin kääntäjiin on toteutettu ohjelmakoodiin tehtäviä tarkastuksia, joiden avulla edellä mainitun kaltaiset tilanteet voidaan havaita ja tuoda käyttäjän tietoon erilaisin ilmoituksin. *Loogiset virheet* taas syntyvät, kun ohjelma toimii niin kuin sitä käsketään, mutta ei niin kuin sen halutaan määritysten mukaan toimivan. Esimerkiksi jokin muuttuja saattaa olla alustamatta tai laskutoimitukset saattavat olla virheellisiä. Tällaisen virheen löytämiseen ja etsintään virheenjäljitystyökaluista on eniten apua.

2.2 Verifiointi (verification, todentaminen)

Verifioinnilla tarkoitetaan ohjelman oikeellisuuden ja oikean toiminnan todentamista. Verifioinnilla varmennetaan ohjelmakoodi ja sen suoritus tutkimalla, että ohjelma tekee sille vaatimustenmäärittelyssä määritellyt tehtävät. Ohjelmistokehitysprosessin jokaisen vaiheen jälkeen todennetaan eli verifioidaan kunkin vaiheen syötedokumentteja eli vaatimustenmäärittelyjä vaiheen tulosdokumentteihin. Esimerkiksi testauksen suunnittelu- ja määrittelydokumentteja verrataan testeistä saatuihin tuloksiin. *Formaalilla verifioinnilla* tarkoitetaan ohjelman oikeellisuuden todentamista, käytössä olevilla formaaleilla menetelmillä.

2.3 Validointi (validation, kelpoistaminen)

Validoinnilla varmennetaan, että toteutettava järjestelmä vastaa loppukäyttäjän tarpeita. Validointi suoritetaan ohjelman oikeassa ympäristössä tarkoituksena löytää toimintoja, jotka ovat käyttäjän tarpeen vastaisia ja varmistaa, että ohjelmiston toiminnot ovat tarkoituksenmukaisia.

2.4 Virheidenpoisto (debugging)

Virheidenpoisto on prosessi, jonka tarkoituksena on löytää ja poistaa virheiden syitä ja aiheuttajia ohjelmakoodista. Miltei kaikissa nykyaikaisissa ohjelmointityökaluissa on käytettävissä virheenpoistotyökaluja, erilaisia askellus-, jäljitys- ja lokiin tulostuskäskyjä ja työkaluja, ohjelmiston suorituksen esimerkiksi aliohjelmakutsujen ja muuttujien arvojen seuraamisen helpottamiseksi. Yksinkertaisimmillaan debuggausta tulee kukin ohjelmoija suorittaneeksi lisäämällä ohjelmoidessaan ylimääräisiä tulostuslauseita ja lokitiedostojen kirjoittamista ohjelman tilan varmistamiseksi kussakin suoritusvaiheessa. Ohjelman virheenpoisto on ohjelmoijan hallittavissa vielä suhteellisen helposti ohjelman yksikkö- eli komponenttitestausvaiheessa, mutta jo laajempien kokonaisuuksien hallitsemiseen tai vaikka vain esimerkiksi tuhansia kertoja suoritettavan silmukan kulloisenkin kierroksen tuloksen varmentamiseen on pakosta käytettäviä ohjelmointiympäristöön integroituja erilaisia debuggaus- ja virheenpoistotyökaluja. Virheenjäljitysohjelma eli debugger ei löydä itse virheitä vaan ohjelma toimii paikannettaessa virhettä apuvälineenä testaajalle, jonka itse tulee tietää mitä ohjelman suorituksen kussakin vaiheessa tarkkailtavien syötteiden ja tulosteiden tulee olla.

2.5 Staattinen testaus

Ohjelmiston testausta ilman, että ohjelmakoodia suoritetaan kutsutaan *staattiseksi testaukseksi*. Koodin analysointi voidaan suorittaa joko käsin tai automaattisesti staattisen analyysin työkalulla. Menettelyllä pyritään löytämään tutkittavasta ohjelmakoodista loogisia-, suunnittelu- ja koodausvirheitä.

Ohjelmakoodin tarkastelu manuaalisesti järjestettävissä *koodin tarkastus* (inspection), *katselmus* (review) ja *läpikäyntitilaisuuksissa* (walkthrough) tai ohjelmoijan itse suorittamana *pöytätestauksena* (desk checking) ovat tehokkaita testausmenetelmiä tiedettäessä mitä virheitä ja minkä kaltaisia virheitä koodista etsitään. Koodin läpikävijöiden tuleekin tuntea testattava sovellus, käytössä oleva sovelluskehitysväline ja ohjelmointikieli mahdollisimman hyvin, että onnistuneisiin tuloksiin päästään. Useimmiten menettelyillä voidaan paljastaa tyypillisiä ja tunnettuja virheitä, joita automaattisesti ei välttämättä havaita. Suurempia koodikokonaisuuksia sisältävässä tarkastusmenettelyssä koodin tarkastelu tapahtuu

useimmiten ryhmätyönä. Jokainen tehtävään valittu testaaja tutkii ja läpikäy koodin ensin itsekseen, jonka jälkeen pidetään palaveri, jossa koodissa paljastuneita virheitä ja ongelmia tutkitaan. Koodin analysoinnissa ei keskitytä pelkästään tyypillisiin virheisiin, vaan monessa tapauksessa puututaan myös käytettyyn ohjelmointityyliin ja hyvän ohjelmointitavan mukaisiin menettelyihin. Useassa tapauksessa ryhmässä tapahtuva koodin tarkastus tuo tehokkaampia tuloksia virheiden etsinnässä kuin perinteiset ohjelmoijan itse suorittamat pöytätestausmenettelyt. Tarkasteltava koodin on oltava tässä tapauksessa muille katselmukseen osallistuville ymmärrettävää.

Eräs tapa ohjelmakoodin välittömään tarkasteluun ennen sen ajoa on erityisesti eXtreme Programming –ohjelmistonkehitysprosessimallissa korostetussa asemassa oleva *parityöskentely* (Pair Programming), jossa ohjelmointi suoritetaan ohjelmoijapareissa. Ohjelmoija on usein sokea omille virheilleen, mutta vieressä jatkuvasti oleva ylimääräinen silmäpari usein löytää mahdollisia virheitä helpommin jo ohjelmakomponentin toteutusvaiheessa. eXtreme Programming –sovelluskehitysmalli korostaa, myös luodun sovelluksen tai ohjelmistokomponentin välitöntä testausta. Mallissa komponentille suoritettavat testit ohjelmoijapari suunnittelee ja toteuttaa ennen kuin riviäkään koodia on implementoitu, jolloin mahdollisia virhetilanteita tulee mietittyä jo ennen kuin niitä on edes päästy toteuttamaankaan. Menetelmällä voidaan onnistua ohittamaan pahimmat sudenkuopat toteutettavissa järjestelmissä etukäteen ja saadaan ylimääräistä näkemystä yleensä yksilökeskeiseen ohjelmistonkehitysprosessin vaiheeseen. Parityöskentelyn katsotaan nopeuttavan työskentelyä ja parantavan ohjelmiston laatua, niin parantuneen testaamisen kautta kuin virheiden paljastumisella jo koodin implementointivaiheessa [IEE00].

Ohjelmointiympäristöjen kääntäjä toimii staattisena virheenjäljitystyökaluna. Käännettäessä lähdekoodia kääntäjä antaa virheilmoituksia ja varoituksia koodissa olevista virheistä ja ohjelman toimintaa mahdollisesti vahingoittavista kohdista. Tyypillisiä staattisten virheenjäljitystyökalujen tarkastamia kohtia koodista ovat esimerkiksi:

- funktiokutsussa parametrien tyypit,
- parametrien käyttö (esim. parametrien väärä lukumäärä),
- funktion paluuarvon tyyppi,
- samannimiset muuttujat/muuttujan uudelleen määrittely,
- onko ohjelmakoodia, jota ei suoriteta koskaan,
- käytetäänkö esiteltyä muuttujaa,

- onko muuttujaa esitelty ennen käyttöä,
- käytetäänkö esiteltyä funktiota,
- muuttujaan sijoitettavan arvon tyyppi ja
- osoittimien ja indeksien käyttö (esim. viittaukset määriteltyjen rajojen ulkopuolelle).

Koodin tarkastuksissa ja katselmuksissa pyritään löytämään tutkittavasta ohjelmakoodista samoja edellä mainitun kaltaisia virheitä, mutta katsastusmenettelyissä pyritään myös siirtämään ryhmän tietämystä aiheesta kaikille ryhmään kuuluville, jolloin oppiminen tulee sivutuotteena virheiden löytymisen ohella.

2.6 Dynaaminen testaus

Komponentin tai ohjelmiston evaluointia itse ohjelman tai sen osan suorituksen aikana ja perusteella kutsutaan *dynaamiseksi testaukseksi*. Dynaamisen analyysin avulla saadaan testattua ominaisuuksia, jotka staattisessa analyysissä olisivat mahdottomia tai ainakin erittäin työläitä testata. Testattaessa ohjelmistoa dynaamisesti on tärkeintä luoda ajettavalle ohjelmalle oikeanlainen testiympäristö eli ympäristö, jossa ohjelma tulee todellisessa käytössäkin toimimaan. Dynaamisen testauksen apuvälineinä olevat dynaamiset virheenjäljitysohjelmat, kuten edellä mainitut debuggerit, voivat aiheuttaa erinäisiä sivuvaikutuksia testauksessa tarkasteltavalle ohjelmalle, joten on tärkeää myös varmistaa, että ajettaessa ohjelmaa ilman virheenjäljitysohjelmaa, päädytään samoihin lopputuloksiin kuin virheenjäljitysohjelman kanssa. Debuggaus -apuväline voidaan toteuttaa joko symbolista lähdekooditiedostoa käyttävänä virheenjäljitysohjelmana tai lähdekoodia tulkkaavana virheenjäljitysohjelmana. Testauksessa virheenjäljitysohjelmien avulla seurataan eri muuttujien arvojen muuttumista, aliohjelmakäskyjä ja niiden parametrejä. Monissa ohjelmointiympäristöissä päästään seuraamaan muistin ja rekisterien sisältöä binäärikoodina suoritettaessa ohjelmakoodia.

Ohjelman suoritus etenee dynaamisissa virheenjäljitysohjelmissa esimerkiksi askeltaen (step) eli koodia suoritetaan lause kerrallaan testaajan hallitessa ohjelmakoodissa etenemistä. Aliohjelmat suoritetaan kutsuttaessa joko lause kerrallaan tai tapauksissa, joissa esimerkiksi tarvitsee tietää ainoastaan aliohjelmien tuottamat tulosteet, ne ajetaan laitteiston normaalilla suoritusvauhdilla.

Ohjelmakoodin suorituksen tarkastelu voidaan tehdä myös pysäyttämällä koodin suoritus tiettyyn kohtaan, jossa muuttujien arvoja halutaan tarkastella ja asettamalla halutulle koodiriville keskeytys (trap/break). Keskeytysten avulla voidaan haluttujen muuttujien arvoja tarkkailla vain halutuissa kohdissa koodia, mikä nopeuttaa virheenjäljitystä monimutkaisessa ohjelmakoodissa. Samalla säästytään tarkastelemasta koodia, jolla välttämättä ei ole mitään merkitystä jäljitettävän virheen kannalta tai, jonka toiminta on jo varmennettu.

2.7 Testattavuus (testability)

Ohjelmiston *testattavuudella* mitataan ohjelmiston kykyä paljastaa omat virheensä ohjelmakoodissa, komponenteissa tai vaatimuksissa. Testattavuutta käytetään ohjaamaan ohjelmiston verifiointia. Pystyttäessä näyttämään suurella varmuudella toteen, että ohjelmassa olemassa olevat viat ovat ilmiselviä ja tulevat testauksessa esiin, kyetään myös osoittamaan vikojen puuttuminen ja ohjelman toiminnan oikeellisuus varmemmin ja nopeammin. Ohjelman testattavuutta parannettaessa, ohjelmasta poistetaan piirteitä, jotka vaikeuttavat ohjelman ymmärtämistä. Samalla valitaan käyttöön testejä, joilla on paremmat mahdollisuudet paljastaa ohjelmissa esiintyvät viat. Menetelmillä on mahdollista luoda metodiikka, joka käyttää vähemmän testejä, mutta saa aikaan suuremman varmuuden ohjelman luotettavuudesta ja turvallisuudesta. Ohjelman eri osien testattavuutta mitattaessa saadaan tietoa mitkä ohjelmakoodin osat voidaan verifioida testaamalla ja mitkä osat on paras verifioida eri keinoin esimerkiksi pöytätestauksella, analysoinnilla ja mallinnuksella [Fri95].

The IEEE Standard Glossary of Software Engineering Terminology (IEEE, 1990) määrittelee testattavuuden seuraavasti. Testattavuus on

(1) taso, jolla testattava järjestelmä tai komponentti helpottaa määriteltyjen testien suorittamista ja testeille asetettujen kriteerien täyttymisen tarkastelua,

(2) taso, jolla testattavalle ohjelmistolle asetetut vaatimukset helpottavat määriteltyjen testien suorittamista ja testeille asetettujen kriteerien täyttymisen tarkastelua.

3. TESTAUKSEN MENETELMÄT

Seuraavassa kuvatut *lasilaatikkotestaus* (glass-box testing) ja *mustalaatikkotestaus* (black-box testing) ovat testauksen eri päälähestymistapoja eli testauksessa käytettäviä eri menetelmiä. Testauksen eri menetelmät eivät ole sidottuja mihinkään tiettyyn testauksen vaihejaon mukaiseen testausvaiheeseen *yksikkötestiin*, *integrointitestiin* tai *järjestelmätestiin*, vaan niitä voidaan käyttää missä tahansa vaiheessa testauksen toteutuksesta riippuen, mitä ja miten halutaan testata. Testauksen eri lähestymistavat eivät ole toisiaan poissulkevia ja usein käytännössä päädytään käyttämään yhdistelmää näistä kahdesta eri lähestymistavasta pyrittäessä haluttuun lopputulokseen. Käytännön syistä, esimerkiksi valitun testausmenetelmän rajoittaessa halutun testin suorittamista voidaan yhdistää eri testausmenetelmiä ja niiden piirteitä, jolloin suoritettavasta testausmenettelystä käytetään nimitystä *harmaalaatikkotestaus* (gray-box testing).

3.1 Lasilaatikkomenetelmät

Lasilaatikkotestauksessa eli white-box testing, josta käytetään myös nimityksiä clear box tai structural testing eli rakenteellinen testaus, täytyy testaajalla olla tietoa järjestelmän sisäisestä rakenteesta, jonka toiminnan testaamiseen lähestymistavassa keskitytään. Testitapausten suunnittelussa ja toteuttamisessa käytetään hyväksi tietoa ohjelman toteutuksesta eli käytännössä testin suunnittelussa tulee olla käytössä ohjelman tai sen osan lähdekoodi ja toiminnallinen määrittely. British Standards Institutionin mukaan lasilaatikkotestauksen testitapausten valinta tulee perustua testattavan komponentin sisäisen rakenteen arviointiin [BSI98a].

Lasilaatikkotestauksessa pyrittäessä mahdollisimman kattaviin testituloksiin keskitytään periaatteessa suunniteltavassa testissä mahdollisimman täydelliseen *lausekattavuuteen* (statement coverage) eli testitapauksien tulisi käydä jokainen järjestelmän sisäisen rakenteen haara läpi [Mye79]. Lausekattavuutta mitataan prosenttiluvulla, joka kertoo testitapauksella saavutettujen lauseiden määrän suhteessa ohjelmakoodin suoritettavissa oleviin lauseisiin [BSI98b] eli:

$C_s = S_e / S_t * 100\%$

missä

C_s on lausekattavuus prosentteina

S_e on suorittujen lauseiden määrä testitapauksessa

S_t on suoritettavien lauseiden kokonaismäärä.

Käytännön syistä, eli ohjelmistojen kompleksisuudesta, johtuen täydelliseen lausekattavuuteen päästään harvoin. Edellä mainittu tavoite on saavutettavissa ainoastaan käytettäessä lasilaatikkotestausmenetelmää yksinkertaisten ohjelmakomponenttien yksikkötestausvaiheessa. Lasilaatikkotestausta suorittamaan tulee valita käytössä olevan ohjelmointiympäristön ja ohjelmointikielen hyvin tunteva kehittäjä eli käytännössä valinta kohdistuu ohjelmoijiin. Kokoelma ohjelmakoodin eri osa-alueille tarkoitettuja lasilaatikkomenetelmän mukaisia testitapauksia on esitelty taulukossa 1.

Taulukko 1. Lasilaatikkomenetelmän mukaisia tyypillisiä testitapauksia lähdekoodissa oleville rakenteille, ohjausrakenteille ja muuttujille.

Lauseet	lausekattavuus: kaikki lauseet suoritettava vähintään kerran
Silmukat	jokaiselle silmukalle suoritetaan testi, jossa toisto suoritetaan: - nolla kertaa - yhden kerran - maksimiarvo kertaa - jokin arvo väliltä 0 – maksimi kertaa
Suorituspolut	kaikki suorituspolut tulee suorittaa kerran
muuttujat	numeeriselle muuttajalle sijoitettavat arvot (jos lukualueen ylittävät tai alittavat arvot mahdollisia): - liian pieni arvo - minimi arvo – 1 - minimi arvo - minimi arvo + 1 - jokin arvo väliltä minimi – maksimi - maksimi –1 - maksimi - maksimi +1 - liian suuri arvo bittikombinaatio: - kaikkia bitit nollia - jokainen bitti vuorollaan yksi - jokainen bitti vuorollaan nolla - kaikki bitit ykkösiä merkkijono: - merkkijonon pituus: nolla (tyhjä merkkijono) - merkkijonon pituus: yksi

	- merkkijono pituus: maksimi - merkkijonon pituus: ylittää maksimin - normaalimittainen merkkijono - numeroiden, kirjaimien ja erikoismerkkien yhdistelmät merkkijonossa
Parametrit	koska parametrit ovat edellä luetellun kaltaisia muuttujia niille suoritetaan samat testit kuin muuttujille
Taulukot	taulukon ylivuototilanteet esiintyvät vähintään kerran

3.2 Mustalaatikkomenetelmät

Mustalaatikkotestausta kutsutaan myös funktionaaliseksi testaamiseksi. Muita käytettyjä ja kuvaavia nimityksiä menetelmästä ovat behavioral testing, functional testing, opaque-box testing facility testing, feature testing tai closed-box testing. Lasilaatikko testauksen salliessa testaajan tarkastella terminologiassa ohjelmaa kuvaavan laatikon sisältöä, mustalaatikko testauksessa suljetaan tämä mahdollisuus pois. Mustalaatikkomenetelmällä ohjelmaa testataan ilman tietoa ohjelman sisäisestä rakenteesta, sen lähdekoodista yleensä tai algoritmista, jonka pohjalta ohjelma on toteutettu. Mustalaatikkotestauksessa keskitytään testitapausten luomisessa ohjelmiston toiminnallisiin määrittelyihin [Mel96], joiden pohjalta valitaan tarvittavia testitapauksia, joista esimerkkejä taulukossa 2. Testaus pohjautuu mustalaatikkomenetelmässä sovellukselle suunnitteluvaiheessa tehtyihin määrityksiin ja niiden vertaamiseen testattavalta sovellukselta testin ajon aikana saataviin tulosteisiin ja vasteisiin. Menetelmän käyttö edellyttää ohjelman suunnittelussa tehdyiltä määrittelyiltä täsmällisyyttä ja oikeaa toteutusta.

Mustalaatikkotestauksessa testattavalle sovellukselle annetaan haluttu syöte, jonka jälkeen sovelluksen toimintaa tutkitaan sen tuottaman tulosteen perusteella. Edellä mainitun kaltainen ohjelman toiminnan tarkastelu on nimennyt mustalaatikkotestaamisen myös toisella nimellä eli funktionaalinen testaaminen, joka kuvaa paremmin tapaa, jolla testaus suoritetaan. Menetelmä soveltuu parhaiten sovelluksen testauksen sen käyttöliittymän kautta, ohjelmiston järjestelmätestauksen ja taantuma- eli regressiotestauksen toteutukseen. Mustalaatikkotestauksen toteuttajaksi ei tulisi valita testattavan ohjelman tekijää, tai edes henkilöä, joka tietää paljon testattavan ohjelman, sisällöstä vaan sovelluksen kannalta täysin ulkopuoleisen tahon käyttö testien suorittamiseen on suotavaa.

Taulukko 2. Mustalaatikkoperiaatteen mukaisia tyypillisiä testitapauksia ohjelman eri toiminnoille.

Toimintopolut	kaikkien toimintopolkujen täydellinen läpikäynti
Syötteet Syötteet	syötettävien tietotyyppien ollessa samat kuin lasilaatikkotestauksen muuttujat: numeeriselle muuttajalle sijoitettavat arvot (jos lukualueen ylittävät tai alittavat arvot mahdollisia): - liian pieni arvo - minimi arvo – 1 - minimi arvo - minimi arvo + 1 - jokin arvo väliltä minimi – maksimi - maksimi –1 - maksimi - maksimi +1 - liian suuri arvo bittikombinaatio: - kaikkia bitit nollia - jokainen bitti vuorollaan yksi - jokainen bitti vuorollaan nolla - kaikki bitit ykkösiä merkkijono: - merkkijonon pituus: nolla (tyhjä merkkijono) - merkkijonon pituus: yksi - merkkijono pituus: maksimi - merkkijonon pituus: ylittää maksimin - normaalimittainen merkkijono numeroiden, kirjaimien ja erikoismerkkien yhdistelmät merkkijonossa
Päivitys	tiedostot: - tyhjän tiedoston päivitys tietokannat: - tietueen lisäys (alkuun, loppuun, keskelle) - tietueen poisto (alkuun, loppuun, keskelle) - duplikaatit - olemattoman tietueen poisto - eri tapahtumat tietueelle ja niiden yhdistelmät (lisäys, muutos, poisto, palautus)
Käsittely	- laskentatarkkuus - erikoistilanteet (vuoden vaihde, vuosituhannen vaihde, lukujen etumerkin muuttuminen)

Tulostus	- eri tulostusmuodot - otsikointi, tunnisteet, sivunumerot - erikoistilanteet (vuoden vaihde, vuosituhannen vaihde, lukujen etumerkin muuttuminen)

3.2.1 Aluetestaus (Domain testing)

Jotta mustalaatikkotestauksella päästäisiin täyteen varmuuteen testattavan komponentin toiminnasta tulisi testi suorittaa kaikilla komponentin syöteavaruuden arvoilla. Suurimmissa osissa testattavista sovelluksista jo sovelluksen yksittäisen komponentin saamien syötearvojen joukon (*input domain*) ja tuottamien tulosarvojen joukon (*output range*) laajuudesta johtuen on käsin testattaessa mahdotonta saavuttaa täysi kattavuus. *Aluetestaamisella* (domain testing) pyritään ratkaisemaan syötealueen laajuuden ongelmaa jakamalla komponentille annettavien syötearvojen joukko eri alueisiin.

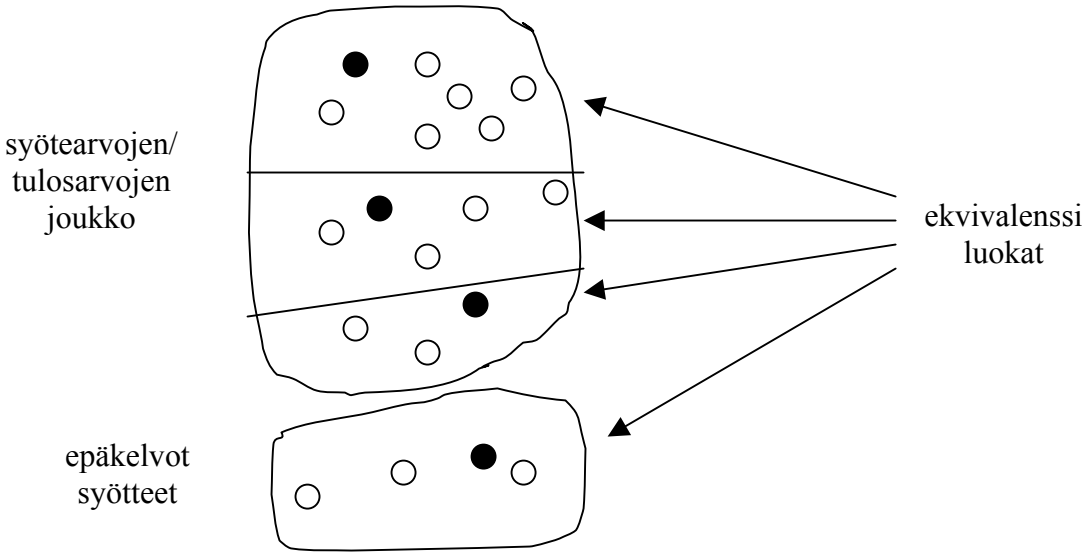
3.2.2 Ekvivalenssiositus (Equivalence partition)

Komponentille annettava syötealue (input domain) tai sen tuottamat tulosarvot (output domain) pyritään jakamaan toisiaan vastaaviin luokkiin eli suoritetaan mahdollisille syötearvoille tai tulosarvoille *ekvivalenssiositus* (equivalence partition). Ekvivalenssiosituksella syötearvojen joukko pyritään jakamaan erillisiin joukkoihin eli *ekvivalenssiluokkiin* (equivalence class). Saman ekvivalenssiluokan sisältä minkä tahansa joukkoon kuuluva arvon tulisi antaa saman vasteen syötettynä testattavalle komponentille tai käydä läpi samanlaisen käsittelyn komponentin sisällä (kts. Kuva 1). Vastaava ositus voidaan tehdä testattavan komponentin tuottamien tulosteiden kohdalla. Ekvivalenssiositus perustuu komponentin oletettuun toimintaan, jonka tulisi selvitä testitapausten suunnittelijalle komponentille laadituista spesifikaatioista [BSI98a]. Suoritettavalla ekvivalenssiosituksella pyritään pienentämään sovelluksen testaamiseen tarvittavien syötteiden määrää. Koska kunkin ekvivalenssiluokan edustajan tulisi antaa sama vaste tai samaan vasteavaruuden joukkoon kuuluva tulos syötteelle, riittää sovelluksen testaus ainoastaan yhdellä tai kahdella luokkansa edustajalla. Luokasta valittujen edustajien testaaminen tulee antamaan varmuuden siitä, toimiiko sovellus oikein kyseisellä syötearvo-alueella [Kan01]. Menetelmän pohjalta luotujen testien kattavuutta mitataan *ekvivalenssiluokan kattavuusmitalla* (equivalence class coverage) eli mitataan kuinka suurta osuutta ekvivalenssiluokista on käytetty suoritettaessa menetelmän mukaista testiä [BSI98b] eli:

$C_{ep} = P_c / P_t * 100\%$,

missä:

C_{ep}	on ekvivalenssiluokan osituksen kattavuus prosentteina
P_c	on testin kattamien luokkien määrä
P_t	on kaikkien luokkien kokonaismäärä.



Kuva 1. Ekvivalennssiositus suoritettuna syöte- tai tulosarvoalueelle, kustakin luokasta valittu yksi testattava arvo (tummennettu ympyrä).

3.2.3 Raja-arvotestaus (Boundary analysis)

Raja-arvotestaus (Boundary analysis) perustuu kahteen oletukseen. Ekvivalenssiluokkiin eli testattavan sovelluksen syöteavaruus tai tulosteavaruus on jaettavissa komponentin määrittelyiden perusteella ekvivalenssiluokkiin, joiden edustajia testattavan komponentin tulisi käsitellä samalla tavalla. Toinen oletus raja-arvotestauksen määrittelyn mahdollistamiseksi on, että jaetuille ekvivalenssiluokille on löydettävissä selkeät *raja-arvot* (boundary). Toisin sanoen syötealueen edustajat on kyettävä asettamaan lineaarisesti järjestykseen keskenään. Kohdassa, jossa esimerkiksi lukusuoralla ekvivalenssiluokka vaihtuu toiseen sijaitsee kyseisen luokan raja-arvo. Ekvivalenssiluokissa sopiva luokkansa edustaja on mi-

kä tahansa arvo luokan sisältä, kun taas raja-arvoanalyysissä testattavaksi syötteiksi valikoidaan luokan pienin ja suurin edustaja.

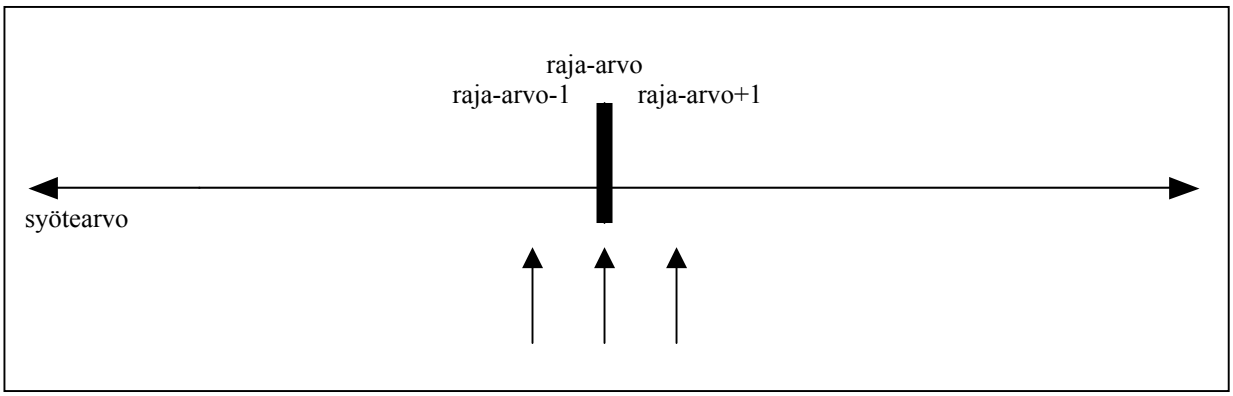
Ohjelmoijalla on taipumuksia tehdä virheitä juuri raja-arvojen käsittelyssä, joten arvot molemmin puolin raja-arvoa tulee myös sisällyttää testattavien syötearvojen joukkoon suunniteltaessa raja-arvoanalyysin mukaista testitapausta. Tyypillisimmillään virhe tapahtuu suoritettaessa vertailuja eri arvoilla, esimerkiksi sekaannus merkkien ≥ ja > välillä. Seuraavassa esimerkissä ohjelman suorituksen haarautuminen riippuu siitä, onko annettu syöte x, syötteen ollessa kokonaisluku, arvoltaan positiivinen. Ehto voidaan kirjoittaa ohjelmakoodiin muotoon:

```
if (x > 0) then
        käsittely positiivisille kokonaisluvuille
else
        käsittely muille kuin positiivisille kokonaisluvuille.
```

Ohjelmoija kirjoittaa virheellisesti ehdon muotoon:

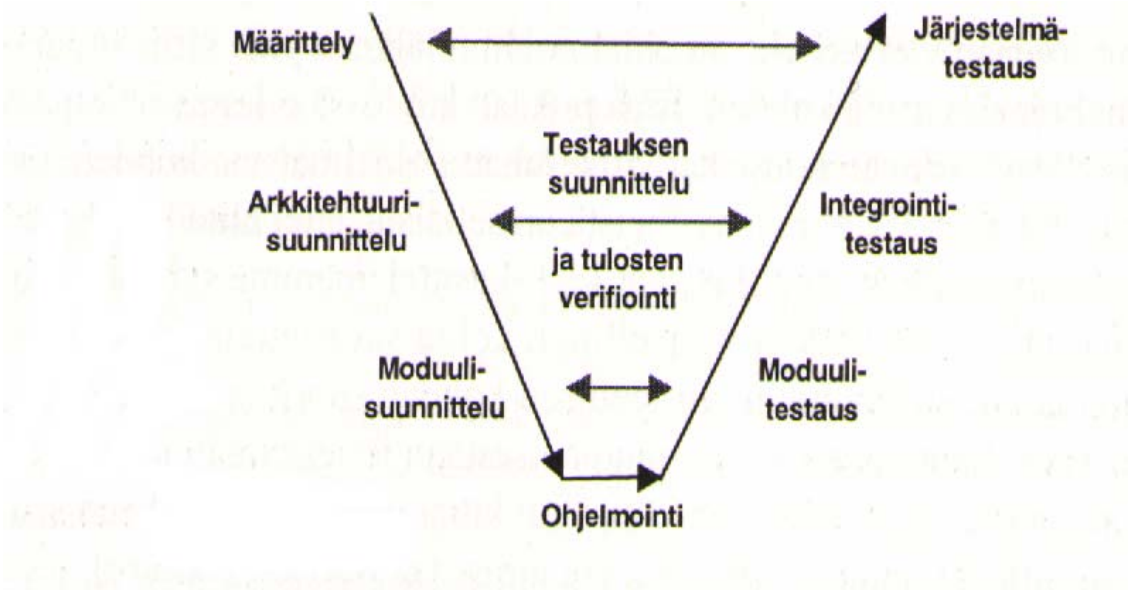
```
if (x ≥ 0) then
        käsittely positiivisille kokonaisluvuille
else
        käsittely muille kuin positiivisille kokonaisluvuille.
```

Ohjelman suoritus päätyy virheellisesti else-haaraan annettaessa syötteen x arvoksi 0. Arvolle 0 suoritetaan sama käsittely kuin positiivisille kokonaisluvuille vaikka se ei kuulukaan joukkoon. Esimerkin tapauksessa positiivisten kokonaislukujen joukon raja-arvo eli pienin arvo on 1. Virhe ei paljastu käytettäessä ekvivalenssiositusta kuin ainoastaan hyvällä onnella eli valittaessa muiden kuin positiivisten kokonaislukujen joukon edustajaksi juuri arvo 0. Virhe paljastuu kuitenkin sovellettaessa raja-arvotestausta, tarkasteltaessa raja-arvon välittömässä läheisyydessä olevia arvoja eli tarvittavien syötearvojen ollessa: raja-arvo-1, raja-arvo ja raja-arvo+1 eli 0, 1 ja 2 (kts. Kuva 2).



Kuva 2. Raja-arvoanalyysin mukaiset testitapaukset raja-arvon kohdalla syötejoukossa.

4. TESTAUKSEN VAIHEJAKO



Kuva 3. Testauksen V-malli kuvaa ohjelmistotuotannon eri tasoilla tapahtuvaa ohjelmiston ja sen testauksen suunnittelua [Hai96].

Testauksen V-mallissa (kts. Kuva 3) testaus suunnitellaan jokaisella ohjelmiston toteutusta vastaavalla suunnittelutasolla. Vaihejakomallin vasemmassa haarassa suoritetaan testien suunnittelu ja määrittely ja oikeanpuoleisessa haarassa tapahtuu testien suorittaminen. Vasemmassa haarassa suoritettavien ohjelmistontuotantoprosessin osa-alueiden lopputuotteina on ohjelmiston määrittelydokumentteja ja spesifikaatioita, joiden oikeellisuus tulisi varmistaa dokumenttien testausmenettelyjen ja tarkastuksien avulla. Tutkielmassa keskitytään kuitenkin tarkemmin V-mallin oikean puoleisen haaran, jossa vaiheiden lopputuottei-

na syntyy ohjelmakoodia: kokonainen järjestelmä tai sen yksittäinen osa eli ohjelmistokomponentti. Kullakin testauksen toteutuksen tasolla saatuja testituloksia verrataan vastaavalla tasolla luotuihin määrittelyihin. Yksikkötestaus, kuvassa nimellä moduulitestaus, määritellään kunkin ohjelmistokomponentin suunnitteluvaiheessa, integrointitestaus määritellään arkkitehtuurisuunnitteluvaiheessa ja järjestelmätestaus suunnitellaan määrittelyvaiheessa. Mallin mukaisissa testauksen vaiheissa testaus nähdään eri näkökulmasta ja eri vaiheet paljastavat erilaisia virheitä testauksen kohteesta. Tärkeä kysymys on, missä vaiheessa kukin testi tulisi suunnitella ja suorittaa ohjelmistotuotantoprosessissa.

Testauksen eri vaiheissa paljastuvat virheet voidaan luokitella vaihejakomallin mukaisesti:

1) Yksikkötestauksen paljastamat virheet ovat ohjelmointivirheitä.
2) Integrointitestauksessa esiin tulevat virheet johtuvat suunnitteluvirheistä.
3) Järjestelmätestaus vaiheessa paljastuvat virheet, voivat pahimmassa tapauksessa johtaa koko prosessin alkuun eli virheellisesti suoritettuun määrittelyvaiheeseen.

Testausvaiheiden menetelmien tulee olla mahdollisimman hyvin vaiheensa tyypillisiä virheitä paljastavia sillä, mitä ylemmälle tasolle virheet testausmallissa kulkevat ohjelmassa mukana, sitä kalliimmaksi ja monimutkaisemmaksi niiden korjaaminen tulee.

4.1 Komponenttitestaus

Komponenttitesti eli moduulitesti tai yksikkötesti (unit test) tarkoittaa ohjelmiston yksittäisten komponenttien testaamista välittömästi niiden valmistuttua suorituskelpoisiksi. Terminologiassa on tehtävä selväksi ero yksikkötestin ja komponenttitestin välille vaikka kyseessä ovatkin periaatteessa samat ohjelmistotuotannon vaiheet. Yksikkötestissä testattavan yksikön ympäristön muodostavat sitä varten tehty testiympäristö, kun taas komponenttitestauksessa komponentin ympäristön muodostaa sen oikea ympäristö. Näitä kahta termiä käytetään synonyymeinä [Faq01]. Ohjelmistokomponentti on oikein suoritetun komponenttisuunnittelun jälkeen 100-1000 koodirivistä koostuva ohjelman erillinen osa esimerkiksi matemaattinen funktio. Testitapauksia suorittaessaan komponenttien testaajalla on oltava käytössään ohjelman lähdekoodi ja kyseisen komponentin määrittelyt eli spesifikaatiot. Testauksen tarkoitus on verrata komponentin toimintaa komponentin toiminnallisiin määrittelyihin ja komponentin rajapintojen spesifikaatioihin eli Haikalan ja Märijärven testauksen vaihejaon mukaisiin moduulisuunnittelun ja arkkitehtuurisuunnittelun tuloksiin.

Ohjelmakoodin läpikäynti ja testaus tapahtuu lasilaatikkoperiaatteella eli ohjelmakoodi on testaajan saatavilla.

Komponenttitestauksen on katettava mahdollisimman tarkkaan komponentin sisältämä ohjelmakoodi. Erilaisten *kattavuusmittojen* käyttö testauksen laajuuden ja käytettävien resurssien määrittelemiseksi on helpoiten toteutettavissa yksikkötestivaiheessa, johtuen koodin ja eri suorituspolkujen vielä suhteellisen rajallisesta määrästä. Kuitenkin ohjelmistoissa olevien syötteiden, tilojen ja toimintoketjujen moninaiset kombinaatiot tekevät ohjelmakoodin täydellisesti kattavasta testauksesta ja analysoinnista vaikeasti suoritettavan ja hallittavan. Esimerkiksi ohjelmalla, joka lukee syötteeksi saamansa kaksi kokonaislukua, on yli neljä miljardia eri syöttö tilaa. Sata koodiriviä pitkä ohjelman pätkä voi sisältää 32 miljoonaa eri suorituspolkua [Fri95].

Useimmissa tapauksissa testattavan yksikön oikea ympäristö ohjelmistossa ei vielä ole valmis. Komponenttiin liittyvät muut komponentit eivät vielä ole valmiita liitettäväksi testattavaan komponenttiin tai niiden toiminta on vielä testaamatta. Tällöin joudutaan luomaan komponenttitestausta varten komponentille sen varsinaista ympäristöä simuloiva testiympäristö eli testipeti (testbed). *Testiympäristö* koostuu ohjelmakomponentin ympäristöä jäljittelevistä *testiajureista* (test drivers), jotka kutsuvat komponentin funktioita tai metodeja ja mahdollistavat saatavien tulosteiden tarkastelun. Testipetiin on usein lisättävä myös tarvittavat *testityngät* (test stubs), jotka korvaavat testauksen aikana komponenttiin liitettävät muut komponentit, syöttölaitteet ja testattavan komponentin kanssa yhdessä toimivat komponentit.

4.2 Integrointitestaus

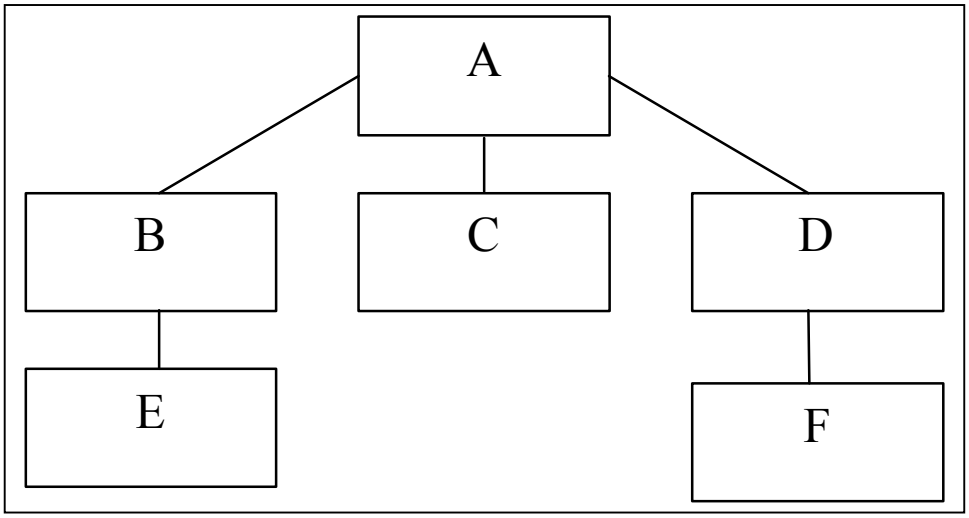
Integrointitestaus on harvoin helposti erotettavissa komponenttitestauksesta, useimmiten kyseessä on ohjelmiston tuotantoprosessissa sama tai keskenään päällekkäiset vaiheet. Integrointitestauksessa yksittäiset ohjelmistokomponentit kasataan arkkitehtuurisuunnitteluvaiheessa laaditun teknisen määrittelyn mukaiseksi kokonaisuudeksi. Jokaisen uuden komponentin liitämisen jälkeen tulee uuden komponentin liittymät eli rajapinnat ja niiden toimivuus testata. Pääpainopisteenä integrointitestausvaiheessa onkin yksittäisen komponentin ja siihen liittyvien komponenttien väliset rajapinnat. Komponentin sisäinen toiminta tulisi olla jo varmennettu yksikkötestausvaiheessa, mutta harvemmin kuitenkaan selvitään

ilman taantumia ja palaamista kokoamisvaiheessa olevien komponenttien sisäisten virheiden korjaamiseen ja uudelleen testaamiseen. Kyseessä olevat kaksi vaihetta lomittuvat väkisin ja testaus tapahtuu päällekkäin. Vaiheiden erottamista toisistaan hämärtää myös valmiiden komponenttien ja prosessien käyttäminen osana komponenttitestiä, jolloin kyseessä on jo osa integrointitestiä.

Testiajurit korvaavat komponenttihierarkiassa komponenttien yläpuolelle sijoittuvat eli komponenttia kutsuvat tai käyttävät ohjelmistokomponentit. Testiajureiden tulee simuloida korvaamansa komponenttien antamia kutsuja hierarkiassa alemmille komponenteille, mutta myös kyetä vastaanottamaan ja tallettamaan kutsumiltaan komponenteilta saamiaan paluuarvoja ja parametrejä testattavan yksikön toiminnan verifioimiseksi.

Testityngät (test stub) ovat komponentteja, jotka korvaavat testattavien komponenttien kutsumia komponentteja. Testitynkien toteuttaminen ei ole yhtä helppoa kuin testiajurien toteutus, testityngän tulee kyetä antamaan oikeanlaisia paluuarvoja sitä kutsuvalle testattavalle komponentille. Esimerkiksi, jos kutsutun komponentin tulee palauttaa luvun n kertoma ei yhdenlainen paluuarvo (esim. ainoastaan kahden kertoma) riitä testattavalle komponentille, vaan testityngän on kyettävä palauttamaan oikea arvo, muuten testattava komponentti ei läpäisee testiä. Eräs mahdollisuus ratkaista ongelma on toteuttaa testitynkä, joka palauttaa oikeita arvoja lukemalla niitä esimerkiksi testitietokannasta, jota simuloimaan voidaan laitaa vaikka tekstitiedosto.

Ohjelmistokomponenttien integrointi voidaan suorittaa neljällä eri tavalla: *kokoava-* (*bottom up)*, *jäsentävä-* (top down) tai *big bang* -lähestymistapa. Luvussa esitetään eri integrointistrategia kaaviona, joka on mukailtu esimerkki Gilford Myersin kirjasta The Art of Software Testing. Kuvassa 4 on esitetty kuudesta komponentista koostuvan ohjelman komponenttihierarkia.



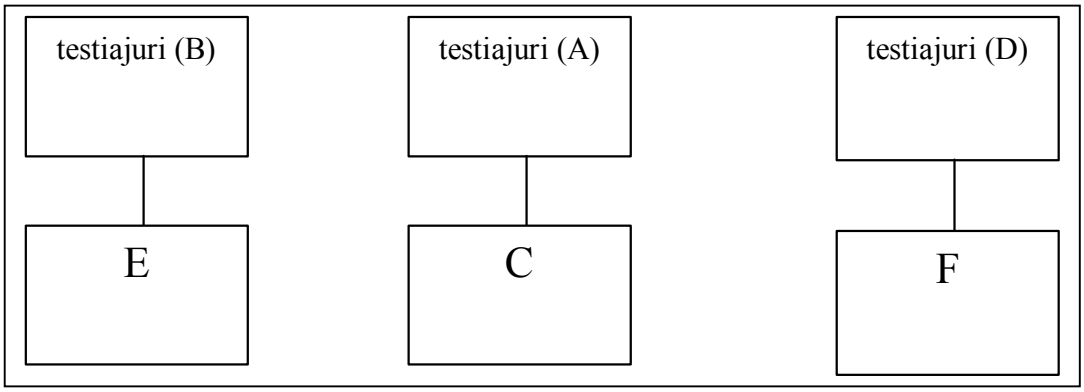
Kuva 4. Esimerkin komponenttihierarkia.

Kirjaimilla A-F nimetyt laatikot esittävät ohjelman komponentteja, joita yhdistävät viivat kuvaavat ohjelmistokomponenttien välisiä riippuvuuksia. Komponenttihierarkiassa yläpuolella olevat komponentit kutsuvat niihin liitettyjä alempia komponentteja, esimerkiksi komponentti A kutsuu komponentteja B, C, D, komponentti B kutsuu komponenttia E ja komponentti D kutsuu vastaavasti komponenttia F. Kuvassa esitetyssä komponenttihierarkian testauksessa komponenttien funktioita kutsuvat testiajurit sijoitetaan tarvittaessa kutsuvan komponentin tilalle testattavan komponentin yläpuolelle ja kutsuttavia komponentteja korvaavat testityngät sijoitetaan vastaavasti testattavan komponentin alle.

4.2.1 Kokoava-lähestymistapa

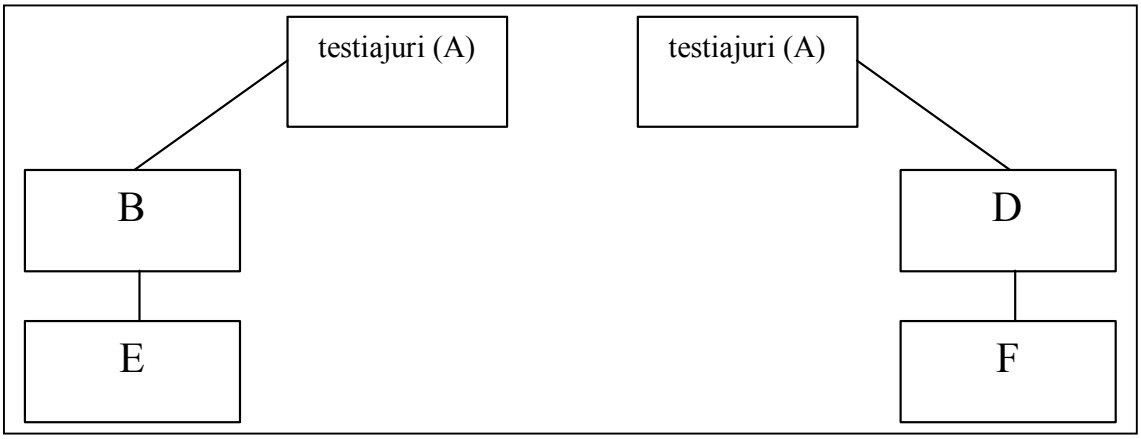
Yleisin tapa toteuttaa integrointi on kokoava, bottom up -menetelmä. Lähestymistavassa testataan ensin komponentit, jotka eivät ole riippuvaisia muista komponenteista eli komponentit, jotka eivät kutsu toisia komponentteja. Ensimmäisten komponenttien testaamisen jälkeen siirrytään testaamaan edellä mainituista komponenteista riippuvaisia komponentteja eli komponentteja, jotka käytävät ensimmäiseksi testattuja komponentteja. Alimpien, toisista riippumattomien komponenttien testauksen jälkeen, ei etenemisjärjestyksellä ole väliä kunhan komponenttien alapuolella olevat komponentit on testattu. Kuvassa esitetyssä komponenttihierarkiassa suoritus järjestys on seuraava: ensimmäisenä testataan alimman tason komponentit E, C ja F. Jokaisessa edellä mainitun komponentin testauksessa tarvi-

taan testiajuri, joka kutsuu testattavaa komponenttia (kts. Kuva 5). Komponentille E asetetaan komponenttia B simuloiva testiajuri suorittamaan komponentille E suunnattuja kutsuja ja vastaanottamaan E:n antamia paluuarvoja. Ohjelmistokomponentille C asetetaan A:ta simuloiva testiajuri ja vastaavasti F:lle D:tä simuloiva testiajuri.



Kuva 5. Integrointitestauksen ensimmäinen vaihe.

Seuraavaksi alimpiin komponentteihin E ja F kyetään integroimaan komponentit B ja D, joita kutsumaan asetetaan komponenttia A simuloiva testiajuri. Komponenttia A ei kyetä integroimaan ja testaamaan ennen kuin kaikki sen alapuolella olevat komponentit ovat C:n lisäksi liitetty toisiinsa ja liittymät testattu (kts. Kuva 6).



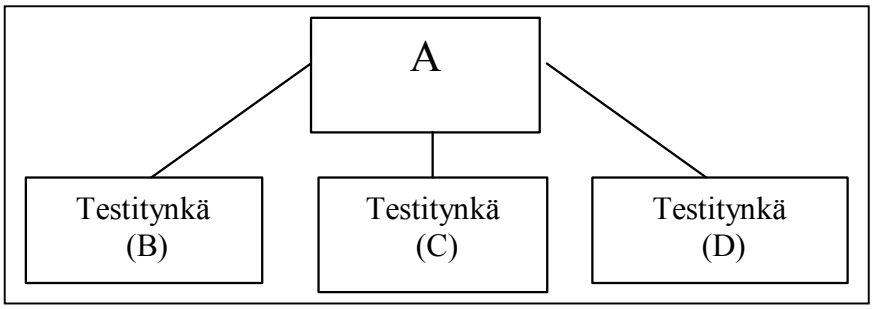
Kuva 6. Integrointitestauksen toinen vaihe.

Viimeisenä vaiheena (kts. Kuva 6) päästään integroimaan hierarkiassa ylinnä oleva komponentti A jo integroituihin osajärjestelmiin: B-E, C, D-F. Testiajureita tai testitynkiä ei enää tarvita, vaan kaikki komponentit ovat omilla paikoillaan niiden yhteistoimintaa testat-

taessa. Kokoavassa -menetelmässä huomataan, että vasta integroinnin viimeisessä vaiheessa päästään testaamaan haluttuja toimintoketjuja, komponentin A toimiessa hierarkian ylimpänä komponenttina. Menetelmän mukaisesti kuvan esimerkki komponenttihierarkialle täytyy toteuttaa viisi eri testiajuria.

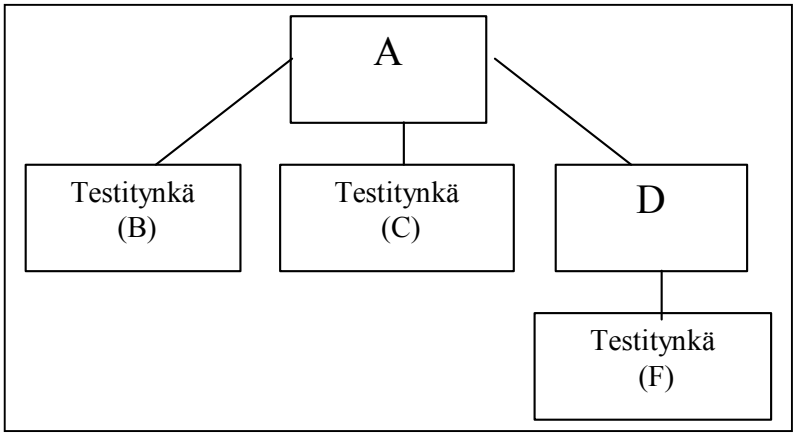
4.2.2 Jäsentävä-lähestymistapa

Jäsentävässä (Top-Down) menetelmässä ensimmäiseksi testattavaksi valitaan komponenttihierarkian ylimmäinen tai toiminnaltaan keskeisin ohjelmistokomponentti eli kuvion hierarkiassa ensimmäinen testattava on komponentti A. Testauksen suorittamiseksi tulee komponentille asettaa sen alapuolella olevien komponenttien toimintoja korvaavat testityngät (kts Kuva 7).



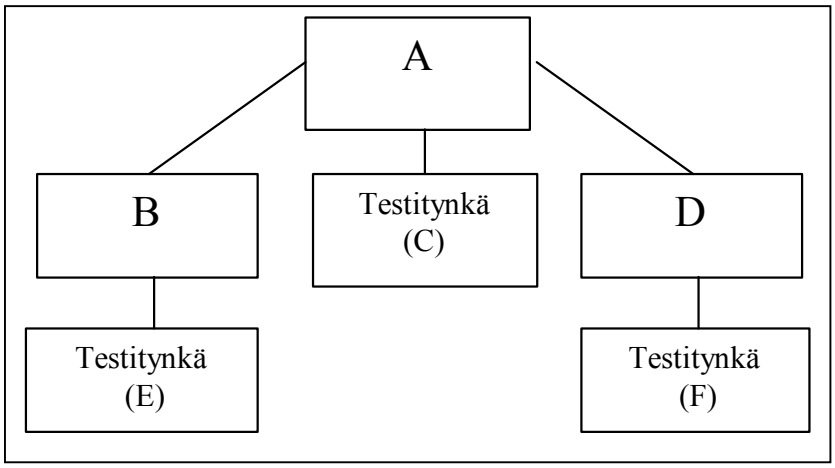
Kuva 7. Integrointitestauksen ensimmäinen vaihe.

Seuraavan testattavan komponentin valintaa ei ole tarkemmin määritetty, ainoa ehto on, että vähintään yksi testattavan komponentin yläpuolella hierarkiassa oleva, eli testattavaa komponenttia kutsuva komponentti, on testattu. Kuviossa seuraavaksi testattavaksi valitaan, joku komponentin A alapuolella olevista komponenteista: B, C ja D (kts. Kuva 8).



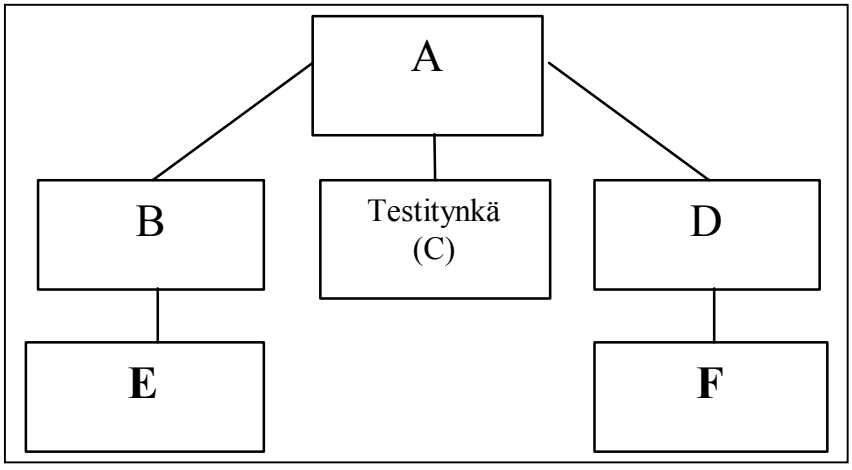
Kuva 8. Integrointitestauksen toinen vaihe.

Jäsentävä lähestymistapa mahdollistaa komponenttien integrointijärjestyksen muokkaamisen tarvittaessa haluttuun järjestykseen, esimerkiksi sovelluksen kannalta kriittisen osajärjestelmän mahdollisimman nopeasti testattavaksi ja valmiiksi saamiseksi. Kuviossa ohjelman syöte- ja tulostuskomponentit ovat komponentit E ja F. Komponentti B hankkii tiedot syötekomponentin E kautta ja komponentti D tuottaa raportin, jonka tulostaminen suoritetaan ohjelman tulostuskomponentilla F. Esimerkkitapauksen projektin integrointitestausvaiheessa katsotaan ohjelman syöttö- ja tulostustoimintosekvenssi valmistumisen kannalta kaikkein kiireisimmäksi ja tärkeimmäksi, joten kaikki käytössä olevat integrointi ja yksikkötestaukseen tarkoitetut resurssit laitetaan työskentelemään osajärjestelmän valmiiksi saamiseksi. Syöte- ja tulostuskomponenttien valmistaminen ensimmäisenä parantaa myös ohjelman testattavuutta, helpottaen syötteiden antamista ohjelmalle ja saatavien tulosten analysointia. Kuvan mukaisessa hierarkiassa ohjelman I/O-komponenttien valmistumisen nopeuttaminen toteutetaan esimerkiksi sijoittamalla ensimmäisessä vaiheessa komponentteja B ja D korvaavien testitynkien tilalle varsinaiset komponentit B ja D ja lisätään niille hierarkiassa alapuolella olevia komponentteja simuloivat testityngät (testitynkä (E) ja testitynkä (F)) (kts. Kuva 9). Seuraavaksi testataan komponenttien toiminta erikseen hierarkian ylimmän komponentin A kanssa, jättäen osajärjestelmän kannalta merkityksetön komponentti C vielä testityngäksi.



Kuva 9. Integrointitestauksen kolmas vaihe.

Neljännessä vaiheessa (kts. Kuva 10) liitetään erikseen testityngän (E) ja testityngän (F) tilalle oikeat komponentit E ja F, joiden toiminta testataan olemassa olevan osajärjestelmän kanssa. Tässä vaiheessa ohjelman syöttö-/tulostuskomponentit on saatu integroitua yhteen ja ohjelman testausta päästään suorittamaan ohjelman omien syöttö- ja tulostusmahdollisuuksien kautta.



Kuva 10. Integrointitestauksen neljäs vaihe.

Viimeisenä vaiheena on testityngän C korvaaminen oikealla komponentilla C ja komponentin integroinnin testaus ohjelman muiden osien kanssa, jolloin päädytään kuvan 4

näköiseen komponenttihierarkiaan. Integrointitestauksen suorittamiseen top-down menetelmällä tarvitaan yhteensä viisi erilaista komponentteja korvaavaa testitynkää.

Esimerkin ohjelmistokomponenttien integrointi tapahtui järjestyksessä:
A, B, D, E, F, C
Vaihtoehtoinen tapa olisi ollut tehdä osajärjestelmä syöttöpuolelta A-E valmiiksi ja sitten edetä tulostuspuolelta A-F saaden aikaan vastaava osajärjestelmä eli toinen mahdollinen järjestys:
A, B, E, D, F, C

Jäsentävä integrointilähestymistapa mahdollistaa myös monia muita vaihtoehtoisia integrointijärjestyksiä riippuen siitä, mitkä komponentit ja toiminnot halutaan ensimmäiseksi valmiiksi ja testattaviksi. Hyviä syitä nopeuttaa osajärjestelmien kokoamista on esimerkiksi testata ensin komponentit, joiden epäillään olevan virheille alttiita tai toiminnot, joiden tulee olla äärimmäisen luotettavia ja jotka tarvitsevat enemmän aikaa ja resursseja testaukseen, mikä takaa tarpeellisen testauksen ohjelman osalle.

4.2.3 Big Bang -lähestymistapa

Big bang -integroinnissa kaikki komponentit ja osajärjestelmät kasataan yhteen kerralla. Menetelmä on harvoin käyttökelpoinen, mutta ideaalitapauksessa ohjelmiston eri komponenttien integroinnin tulisi onnistua kaikkien komponenttien toimiessa täysin oikein eristettynä ja yhdessä, kuitenkin käytännössä menetelmän käyttö onnistuu ainoastaan komponenttien välisten rajapintojen ollessa vähäisiä ja niistä huolehtii vain muutama hyvin testattu funktio. *Big bang* -lähestymistapa vaatii huomattavasti enemmän töitä yksikkötestaus vaiheessa. Kaikille komponenteille, joita pidetään eristyksissä ennen niiden integrointia, täytyy toteuttaa oma testausympäristö eli tarvittavat testityngät ja testiajurit, varmistamaan kunkin komponentin täydellinen toimivuus erillään osajärjestelmästä. Esimerkeissä esitetyssä tapauksessa tarvitsee toteuttaa Big Bang -integrointia varten ohjelmassa oleville komponenteille kaikille omat testiympäristöt, yhteensä viisi testiajuria ja viisi testitynkää, ennen kuin ohjelman osien integrointi voidaan suorittaa.

4.2.4 Toimialakomponenttien integrointi

Olio-ohjelmoinnissa suositeltava lähestymistapa olioiden keskinäiseen integrointiin on kokoava bottom-up lähestymistapa. Olio-ohjelmoinnissa käytetyistä menetelmistä on myös johdettu Padmal Vitharanan ja Hemant Jainin esittämä *toimialakomponenttien* (business component) integrointiin soveltuva lähestymistapa Component assembly and testing strategy [Vit00]. Menetelmä on periaatteeltaan lähellä edellä kuvattua bottom-up integrointia, mutta integrointijärjestystä määriteltäessä kiinnitetään huomiota komponenttien väliseen liikenteeseen eli tarkemmin niiden toisilleen välittämien kutsujen määrään. Menetelmällä pyritään löytämään virhetilanteet ja ristiriitaisuudet komponenttien rajapinnoissa mahdollisimman aikaisessa vaiheessa ja välttämään testattavan osajärjestelmän kompleksisuuden kasvamista samaan aikaan.

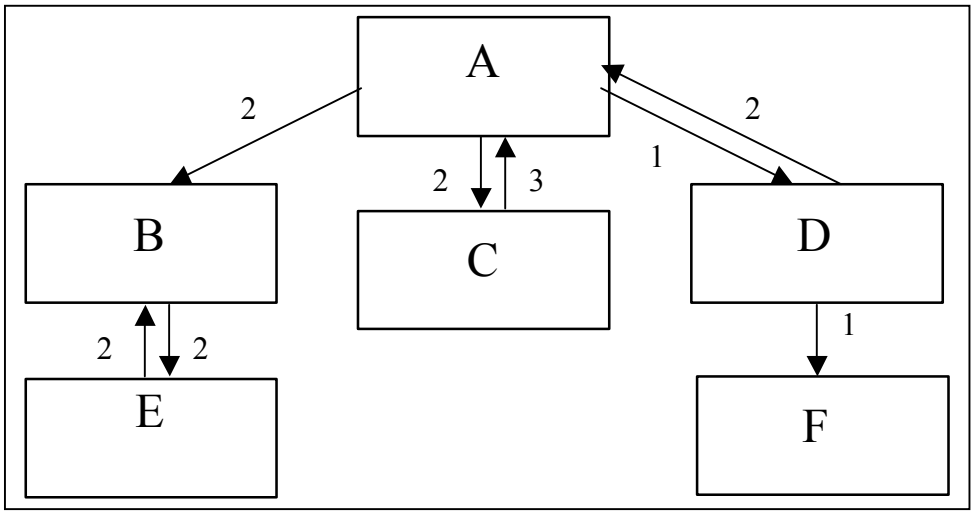
Menetelmä soveltuu valmiiden toimialakomponenttien keskinäiseen integrointiin eli lopputuloksena syntyvä integroitu ja testattu järjestelmä koostuu joukosta keskenään yhteensopivia ja toimintaansa valittuja toimialakomponentteja, jotka liikennöivät keskenään rajapintojensa kautta. Esitetty integrointijärjestys ja siten myös komponenttien välisten rajapintojen testausjärjestys koostuu kahdesta vaiheesta:

Vaihe 1: Valitaan joukko komponentteja, joilla on suurin määrä kutsuja keskenään eli komponenteista muodostuvan alijoukon sisällä, mutta pienin määrä kutsuja ulos omasta alijoukostaan.

Vaihe 2: Jäljelle jäävistä eli keskenään integroitujen komponenttien muodostaman alijoukon ulkopuolelle jäävistä komponenteista valitaan seuraavaksi aiemmin integroidun alijoukon kanssa integroitavaksi komponentti tai joukko komponentteja. Valintaperusteena seuraavaksi integroitavalla komponentilla on, että sillä tulee olla kaikkein suurin määrä kutsuja jo aiemmin integroidun joukon kanssa. Vaihetta 2 toistetaan niin kauan kuin kaikki järjestelmään kuuluvat komponentit ovat tulleet integroiduiksi ja toiminta testatuksi jo aikaisemmissa vaiheissa integroidun alijärjestelmän kanssa.

Esimerkkinä integrointijärjestyksestä edellä kuvatulla tavalla muutetaan luvussa käytettyä esimerkkikomponenttiarkkitehtuuria siten, että komponentteja liittäviin kutsurajapintoihin lisätään niiden välillä kulkevien eri kutsujen määrä ja suoritettavien kutsujen suunta (kts.

Kuva 11). Komponentti, josta kutsua kuvaava nuoli lähtee kutsuu komponenttia, johon nuoli osoittaa.



Kuva 11. Esimerkin komponenttiarkkitehtuuri, johon on lisätty komponenttien välisten kutsujen määrä ja suunta.

Sovellettaessa menetelmää integrointijärjestys muodostuu seuraavaksi:

Vaihe 1: Ensimmäisessä vaiheessa todetaan, että komponenttien D ja F välillä on vain yksi kutsu ja niistä kumpikaan ei suorita kutsuja muille komponenteille eli ensimmäisiksi integroitaviksi valitaan komponenttien muodostama alijärjestelmä eli komponentti DF.

Vaihe 2:
Ensimmäinen iteraatiokierros: Selvitetään kutsujen määrä DF-alijärjestelmän ja muiden hierarkiaan kuuluvien komponenttien välillä (kts Taulukko 3). Koska komponentilla A on ainoana komponenttina yhteyksiä DF-komponentin kanssa eli sillä on suurin määrä kutsuja jo integroidun joukon kanssa, on se seuraava integroitava, eli muodostuu alijärjestelmä ADF.

Taulukko 3. Kutsujen määrä alijärjestelmän DF ja muiden komponenttien välillä.

komponentti	A	B	C	E
DF	3	0	0	0

Toinen iteraatiokierros: Muodostetaan tilannetta vastaava taulukko kuten ensimmäisellä kierroksella ja havaitaan, että komponentti C:llä on eniten yhteyksiä jo integroituun alijärjestelmään ADF (kts. Taulukko 4). Suoritettavan integroinnin jälkeen muodostuu alijärjestelmä ACDF testattavaksi.

Taulukko 4. Kutsujen määrä alijärjestelmän ADF ja muiden komponenttien välillä.

komponentti	B	C	E
ADF	2	5	0

Kolmas iteraatiokierros: Koska ainoastaan komponetilla B on yhteyksiä alijärjestelmään ACDF (kts. Taulukko 5), integroidaan se seuraavaksi olemassa olevaan osajärjestelmään.

Taulukko 5. Kutsujen määrä alijärjestelmän ACDF ja muiden komponenttien välillä.

komponentti	B	E
ACDF	2	0

Neljäs iteraatiokierros: Ainoa jäljellä oleva komponentti E integroidaan loppujärjestelmään, jolloin kaikki järjestelmän muodostavat komponentit on saatu integroiduiksi keskenään ja niiden välinen toiminta on tullut testatuksi jokaisessa vaiheessa toteutetulla osajärjestelmän ja integroitavan komponentin välisellä integrointitestillä.

4.3 Järjestelmätestaus

Järjestelmätestauksessa eli systeemitestauksessa (system testing) altistetaan valmistunut järjestelmä testattavaksi kokonaisuudessaan ennen kuin se toimitetaan asiakkaalle. Järjestelmätestauksen päätavoite on tutkia täyttääkö valmis systeemi sille määrittelyssä asetetut vaatimukset. Toisena tavoitteena voidaan taas pitää, että kyetään löytämään ja korjaamaan mahdollisimman paljon ristiriitoja valmistuneen systeemin ja sen määrittelyn väliltä. Systeemitestauksessa testataan koko ohjelmaa ja sen koko ympäristöä, joten systeemitestaus on tärkeää toteuttaa täsmälleen samankaltaisessa tai samassa ympäristössä, jossa valmista ohjelmaa tullaan todellisuudessa käyttämään. Täydellisesti suoritetun yksikkötestauksen ja integrointitestauksen jälkeen järjestelmätestausvaiheessa ei tulisi enää paljastua muita kuin

virheellisestä vaatimusten määrittelystä johtuvia virheitä, muuten edellä menneet testausvaiheet ovat epäonnistuneet ja niihin joudutaan palaamaan ennen kuin järjestelmätestausta päästään suorittamaan. Testitapausten suunnittelu ei pohjaudu ainoastaan ohjelman määrittely dokumentteihin, vaan testitapausten suunnittelussa apuna tulisi käyttää myös jo laadittuja ohjelman toimintoja kuvaavia, sen loppukäyttäjille suunnattuja manuaaleja ja ohjeistuksia [Mye79]. Käyttäjille suunnatun opastuksen käyttäminen testitapausten luomiseen vaatimusten määrittelyn ohella saa aikaan myös käyttäjille suunnatun dokumentaation tarkastuksen ohjelman toimintaa vasten. Näin dokumentaatio tulee testatuksi järjestelmätestausvaiheen hyödyllisenä sivutuotteena.

Seuraavassa luettelo järjestelmätestauksen testauskohteista MSQH:n mukaan ja testeissä asetettavista kysymyksistä [STR91]:

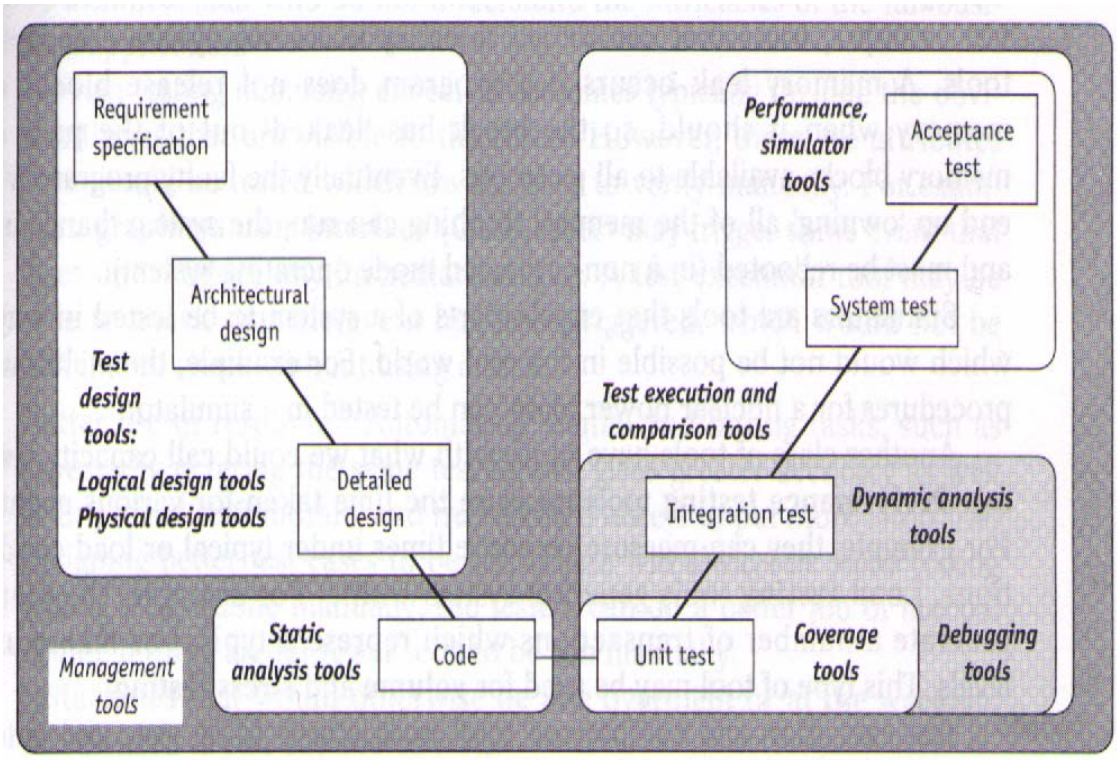
- Tekeekö ohjelmisto sen mitä sen pitää tehdä? (ohjelman toiminnallisuus)
- Onko ohjelma helppokäyttöinen ja vikasietoinen? (käyttöliittymä)
- Vastaavatko opasteet ja Help -tiedostot ohjelmaa? (opasteet)
- Vastaavatko laaditut manuaalit ohjelmaa? (dokumentaatio)
- Toimiiko järjestelmä kuormitettuna? (kuormitus)
- Täyttääkö ohjelmiston toiminta sille asetetut suorituskyky-vaatimukset? (suorituskyky)
- Onnistuuko lopettaminen virhetilanteissa? (lopetus virhetilanteissa)
- Toimiiko järjestelmä suurilla syötteillä jne.? (suuret määrät dataa)
- Toimiiko käyttäjän tunnistus ja tietoturvallisuus? (turvallisuus)
- Toimiiko ohjelmisto oikeassa ympäristössään? (liitettävyys)
- Pysyykö ohjelma pystyssä vaadittuja aikoja? (luotettavuus)
- Sietääkö ohjelma virhetilanteita eli varmistustoimintojen toimivuus? (toipumiskyky)
- Onnistuuko asennus asennusohjeiden mukaan? (asennettavuus)

5. TESTAUSTYÖKALUT

Ohjelmistotuotannossa testauksen täydellinen suorittaminen on prosentuaalisesti erittäin suuri tekijä prosessiin budjetoituja resursseja ajatellen. On järkevintä pyrkiä jättämään pois mahdollisimman suuri osa käsin suoritettavasta testauksesta automatisoimalla testauksen

suoritus. Testauksen automatisointi voi myös parantaa ohjelmistojen laatua suoranaisesti lisäämällä testaajien aikaa perehtyä testaamaan toimintoja, joita ei voida automatisoida, ja automatisoimalla se, mikä voidaan. Eli samalla määrällä resursseja tulisi pystyä lisäämään testauksen määrää ja kattavuutta, *automatisoidun testausjärjestelmän* avulla. On kuitenkin huomattava, että automatisoidun testijärjestelmän luominen on useissa tapauksissa monin verroin enemmän resursseja vaativa toimenpide verrattuna manuaaliseen testaamiseen. Ajan kanssa testauksen automatisoiminen alkaa tuottaa itseensä sijoitettuja työmääriä takaisin.

Ohjelmistotuotannon eri tasoilla on käytössä eri *testaustyökalut*, kuten Software Test Automation teoksessa todetaan (kts. Kuva 12) [Few99]. V-mallissa kuvataan samalla tavalla kuin Haikalan-Märijärven teoksessa testauksen suunnittelua ja toteutusta ohjelmistotuotannon vaihejaon mukaisilla tasoilla. Oheiseen Fewsterin V-malliin on myös lisätty eri testityökalujen käytön sijoittuminen ohjelmistotuotannon tasoille. Usein eri työkalut ja niiden toiminnot eivät ole niin tarkoin rajattuja, etteivätkö ne olisi käyttökelpoisia jossain muussakin vaiheessa testausta. V-malli antaa kuitenkin suhteellisen tarkat suuntaviivat, siitä minkä tyyppinen työkalu on käytössä missäkin vaiheessa testausta.



Kuva 12. V-malli testaustyökalujen kannalta.

Kuvassa esiintyvistä testityökaluista tutustutaan tarkemmin itse ohjelmakoodin toteuttamisen yhteydessä tarvittaviin työkaluihin ja sen jälkeen ohjelmistotuotantoprosessissa seuraaviksi tarvittaviin testityökaluihin eli koodin implementoinnissa käytettäviin staattisiin virheenjäljitystyökaluihin, yksikkö- ja integrointitesteissä käytettäviin dynaamisiin virheenjäljitystyökaluihin. Lopuksi tarkemmin perehdytään järjestelmätestauksessa käytettäviin testien nauhoitus- ja toistotyökaluihin.

6. STAATTISET VIRHEENJÄLJITYSTYÖKALUT

Staattisten virheenjäljitystyökalujen tarkoitus on löytää virheitä ohjelmakoodista ilman varsinaisen ohjelman suorittamista. Yksinkertaisin esimerkki staattisen analyysin apuvälineestä on kääntäjä, joka ohjelmakoodin kääntämisen yhteydessä ilmoittaa koodissa havaituista virheistä. Kääntäjä (compiler) käy ohjelman lähdekoodia läpi rivi riviltä ja tuottaa tuloksena suorituskelpoisen ohjelman eli ohjelmointikielellä kirjoitetusta lähdekielisestä ohjelmasta tuotetaan konekielinen ajettava koodi esimerkiksi Windows -ympäristössä .exe -tiedosto. Kääntäjä analysoi käännettävää lähdekoodia ja kykenee tuottamaan virheilmoituksia ja varoituksia koodissa esiintyvistä käytettävän ohjelmointikielen syntaksin vastaisista virheistä ja myös yksinkertaisimmista ohjelmakoodissa esiintyvistä ohjelmarakenteellisista virheistä. Kääntäjät kykenevät ilmoittamaan ohjelmakoodissa ilmenevistä käännösaikaisista virheistä eli kirjoitetun ohjelman kielioppivirheistä. Ohjelman mennessä käännöksestä läpi, siinä voi olla kuitenkin loogisia virheitä, joiden takia ohjelma ei toimi sille tarkoitetulla tavalla. Staattiset virheenjäljitystyökalut löytävät osan ohjelmakoodissa olevista loogisista virheistä esimerkiksi ehtolauseet, jotka ovat aina tosia, tai ohjelmarakenteessa olevan saavuttamattoman koodin. Kuitenkin ohjelmaan voi jäädä myös koodia, joka ei tee mitään tai aiheuttaa virheitä ajon aikana, mutta on silti täysin ohjelmointikielen syntaksin mukaista. Tällaisten virheiden havaitseminen voi tapahtua vasta koodin suorittamisen aikana tai suoritettaessa ohjelmakoodin staattista analysointia käsin esimerkiksi pöytätestauksessa.

Yleisimmin kääntäjä tuottaa käännöstä tehdessään kahdentyyppisiä ilmoituksia: varoituksia (warnings) ja virheilmoituksia (errors). Kääntäjän ilmoittaessa virheistä ohjelma ei ole ohjelmointikielen syntaksin mukainen, eikä ajettavaa ohjelmaa voida tuottaa. Varoitukset eivät estä suorituskelpoisen ohjelman tuottamista ja ohjelman ajamista. Kääntäjän tuottamien varoitusten avulla ilmoitetaan ohjelmassa olevista epäilyttäviltä vaikuttavista piirteis-

tä. Tyypillisiä tarkastettavia kohtia käännettävästä ohjelmakoodista ja kääntäjien tuottamia varoituksia ja virheilmoituksia ovat:

- samannimiset muuttujat/muuttujan uudelleen määrittely,
- ei käytetyt muuttujat,
- muuttujan käyttö ennen sen määrittelyä,
- väärän tietotyypin sijoittaminen muuttujaan,
- viittaus indeksiin, joka on määrättyjen rajojen ulkopuolella,
- funktiokutsun parametrien lukumäärä,
- funktiokutsun parametrien tyyppi,
- funktion paluuarvon tyypit ja
- ohjelmakoodi, jota ei suoriteta koskaan.

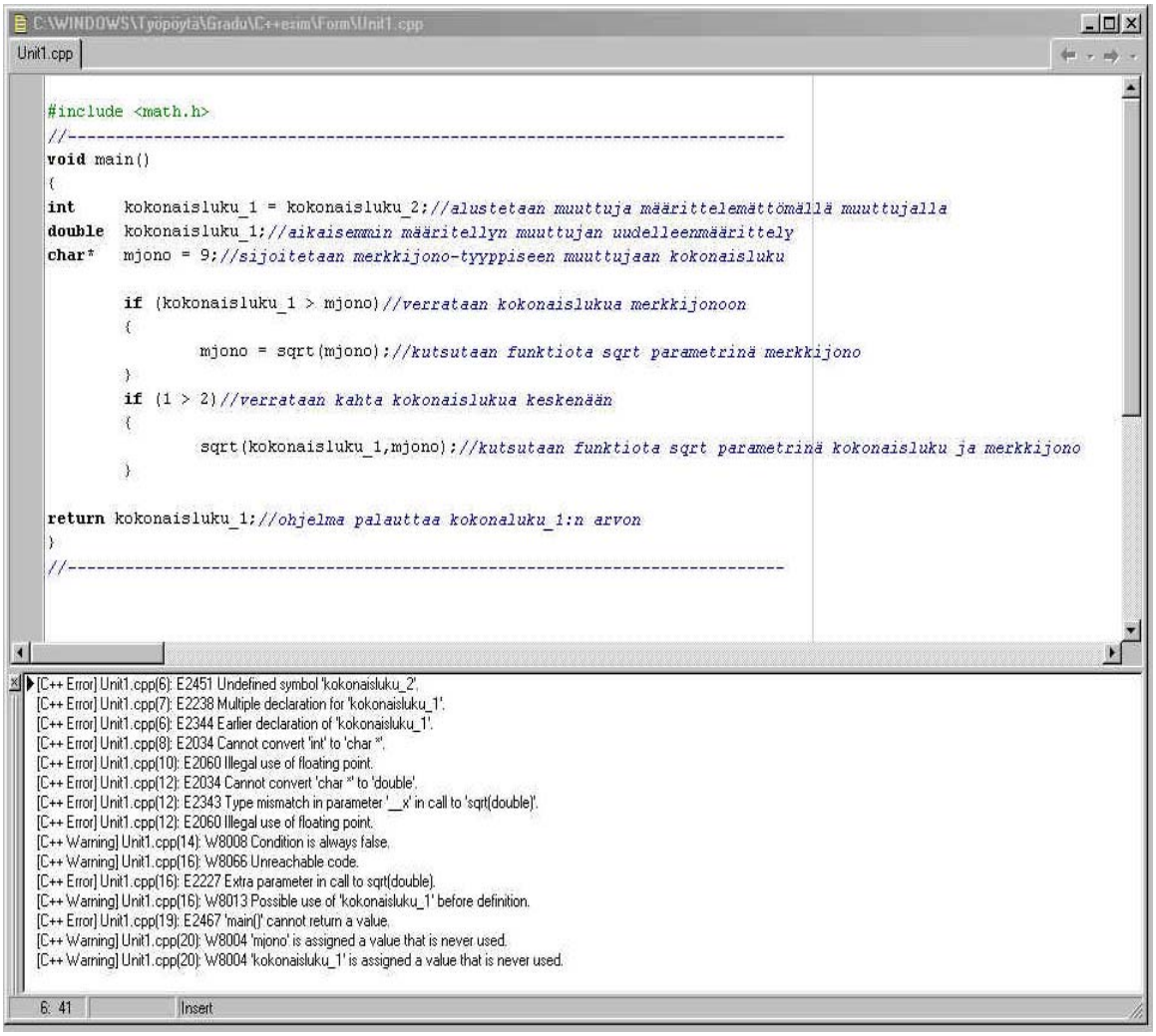
6.1 Kääntäjä esimerkki

Seuraavassa lyhyessä C-kielisessä esimerkkiohjelmassa on kirjoitettu erilaisia määrittelyitä muuttujille, funktiokutsuja ja yksinkertaisia ohjelmarakenteita, joita tutkitaan Borland C++-Builder 4.0 ohjelmointiympäristössä olevalla staattisella virheenjäljitystyökalulla eli kyseisen sovelluskehittimen kääntäjällä. Lähes kaikissa nykyisissä ohjelmointiympäristöissä on käytössä pitkälti samantyyppisiä virheenjäljitystyökaluja kuin käytetyssä esimerkkiympäristössä. Ohjelmistotalo Borlandin muissa tuotteissa mm. Delphi- ja J-Builder-ympäristöissä työkalut ovat C++-Builderin vastaavien kanssa lähes yhteneväisiä. Samoin C++-Builderissa esitellyt toiminnot ovat hyvin samankaltaisia Microsoftin tarjoaman Visual Studio –sovelluskehitysympäristön kanssa, erot löytyvät lähinnä eri toimintojen nimiöinnistä ja työkalujen ulkoasuista.

Esimerkkikoodissa on toteutettu yksinkertainen C-kielen syntaksin mukainen ohjelma, joka määrittelee muuttujia, vertailee niitä ehtolauseissa ja kutsuu funktioita. Koodi sisältää mahdollisimman paljon virheitä muuttujien määrittelyssä, funktiokutsuissa ja ohjelmarakenteissa. Ohjelmassa on käytetty sqrt-funktiota, joka kuuluu C++-Builder:in luokkakirjastoon math.h. Sqrt-funktion tulee saada parametrinä double-tyyppinen luku eli desimaaliluku, jolle funktio laskee neliöjuuren ja palauttaa sen arvon double-tyyppisenä paluuarvona. Muita ohjelmakoodissa suoritettuja toimintoja on kuvailtu kunkin rivin kommenteissa (//-merkin jälkeen).

```
1 #include <math.h>
2 //------------------------------------------------------------------------------
3 void main()
4 {
5
6 int     kokonaisluku_1 = kokonaisluku_2;//alustetaan muuttuja määrittelemättömällä muuttujalla
7 double  kokonaisluku_1;//aikaisemmin määritellyn muuttujan uudelleenmäärittely
8 char*   mjono = 9;//sijoitetaan merkkijono-tyyppiseen muuttujaan kokonaisluku
9
10       if (kokonaisluku_1 > mjono)//verrataan kokonaislukua merkkijonoon
12      {
13              mjono = sqrt(mjono);//kutsutaan funktiota sqrt parametrinä merkkijono
14      }
15      if (1 > 2)//verrataan kahta kokonaislukua keskenään
16       {
17              sqrt(kokonaisluku_1,mjono);//kutsutaan funktiota sqrt parametrinä kokonaisluku ja merkkijono
18      }
19
20 return kokonaisluku_1;//ohjelma palauttaa kokonaluku_1:n arvon
21 }
22 //------------------------------------------------------------------------------
```

Kääntäjä tuottaa edellä kuvatun kaltaisesti kahdenlaisia virheilmoituksia: Error- ja Warning-tyyppisiä. Error- tyyppiset virheilmoitukset estävät koko ohjelman suorituksen, koska niissä määrättyjä käskyjä ja toimenpiteitä ei pystytä toteuttamaan. Kääntäjän tuottamat Warning- tyyppiset virheilmoitukset eivät estä ohjelman suoritusta, mutta ovat kääntäjän kannalta epäilyttävää koodia ja voivat aiheuttaa virhetilanteita ohjelmaa suoritettaessa (kts. Kuva 13).



Kuva 13. C++-Builderin käännösikkuna.

Esimerkissä kääntäjän antamat virheilmoitukset ja varoitukset:

- Virhe rivi 6: määrittelemätön muuttuja.
- Virhe rivi 7: saman muuttujan uudelleenmäärittely.
- Virhe rivi 6: em. muuttujan aikaisempi määrittely.
- Virhe rivi 8: tyypinmuunnos, jota ei voida toteuttaa.
- Virhe rivi 10: tietotyypin väärä käyttö.
- Virhe rivi 12: tyypinmuunnos, jota ei voida toteuttaa.
- Virhe rivi 12: väärän tyyppinen parametri funktiokutsussa.
- Virhe rivi 12: tietotyypin väärä käyttö.
- Varoitus rivi 14: ehtolause, joka on aina epätosi.
- Varoitus rivi 16: saavuttamaton koodi.
- Virhe rivi 16: ylimääräinen parametri funktiokutsussa.
- Varoitus rivi 16: muuttujan mahdollinen käyttö ennen määrittelyä.

- Virhe rivi 19: funktio ei voi palauttaa arvoa eli pääohjelman käännös ei onnistu.
- Varoitus rivi 20: muuttujalle alustetaan arvo, jota ei koskaan käytetä.
- Varoitus rivi 20: muuttujalle alustetaan arvo, jota ei koskaan käytetä.

7. DYNAAMISET VIRHEENJÄLJITYSTYÖKALUT

Dynaamisella testauksella tarkoitettaan virheiden etsimistä ohjelmasta ohjelmaa tai sen osaa suorittamalla. Suorituksenaikaisen testauksen ja analysoinnin mahdollistamiseksi, tarvitaan ohjelmalle toimiva ympäristö, joka mahdollistaa ohjelmaa ajamalla saatavien tulosten havainnoinnin. Dynaamisessa testauksessa, suoritettaessa yksikkötestiä tai integrointitestiä, testattavan ohjelmakomponentin ympäristö muodostetaan yleensä testiajureiden ja testitynkien avulla. Testiajurit ja -tyngät voivat olla joko itsenäisiä ohjelmia tai testattavaan ohjelmaan yhdistettyjä erillisiä osia.

Testiympäristön avulla testattavasta komponentista voidaan saada ulos paljon erityyppistä informaatiota testausta suoritettaessa. Testattavasta sovelluksesta saatavat tulosteet voidaan tallentaa erillisiin lokitiedostoihin, joista haluttua informaatiota voidaan tarkastella. Käytössä olevan sovelluskehittimen virheenpoistotyökaluja käyttäen muuttujien arvoja ja ohjelman tuottamia tulosteita voidaan seurata suoraan työkalun tuottamista tulosteista ja ajonaikaisesta seurannasta. Dynaamisen analyysin työkaluilla saadaan myös tietoa testattavan ohjelman vaikutuksesta ympäristöönsä, esimerkiksi tietoa ohjelman muistin ja resurssien käytöstä. Testaus suoritetaan joko käsin syöttäen testitapahtumia ja ohjelman tarvitsemia testisyötteitä tai antaen testiajurin lukea annettavat syötteet testattavan järjestelmän ulkopuoliselta tulostuslaitteelta, joka suorittaa tai syöttää testattavalle ohjelmalle sille määritellyt testitapahtumat.

7.1 Debugger esimerkki

Useimmissa nykyaikaisissa ohjelmointiympäristöissä on käytettävissä sovelluskehittimeen integroitu virheenpoistotyökalu eli debugger, joista on otettu esimerkiksi edellä mainitun Borland C++Builder 4.0 -kehitysvälineessä olevat virheenjäljitystyökalut, joilla kyetään toteuttamaan kaikki yleisimmät ajonaikana tapahtuvat ohjelmakoodin tarkasteluun ja testaukseen liittyvät toimenpiteet, kuten keskeytysloukut, askellus, jäljitysloukut ja muuttujien arvojen seuranta ja muuttaminen kesken ajon.

Dynaamisen virheenjäljitystyökalun toiminnan esittelyä varten on luotu äärimmäisen yksinkertainen Kertolaskin -sovellus C++-Builder ympäristössä (kts. Kuva 14). Ohjelman lomakkeella on kaksi syöteteksti kenttää (Luku 1, Luku 2), tekstikenttä, johon kertolaskun tulos (Tulos-kenttä) saadaan painamalla Laske-painiketta. Ohjelman toiminnallisuus on kokonaisuudessaan yhdessä funktiossa (Laske_painikeClick), jota kutsutaan Laske-painikkeen painalluksella.

```
void__fastcall TForm1::Laske_painikeClick(TObject *Sender)
{
editTulos->Text = StrToInt(editLuku1->Text) * StrToInt(editLuku2->Text);
}
```

Ohjelmointiympäristön virheenjäljitystyökalun avulla päästään tarkkailemaan muuttujien arvoja sovelluksessa suorituksen aikana seuraavaksi kuvattavien keskeytysloukkujen ja seurantatyökalujen avulla.

Kuva 14. Kertolaskin-sovellus.

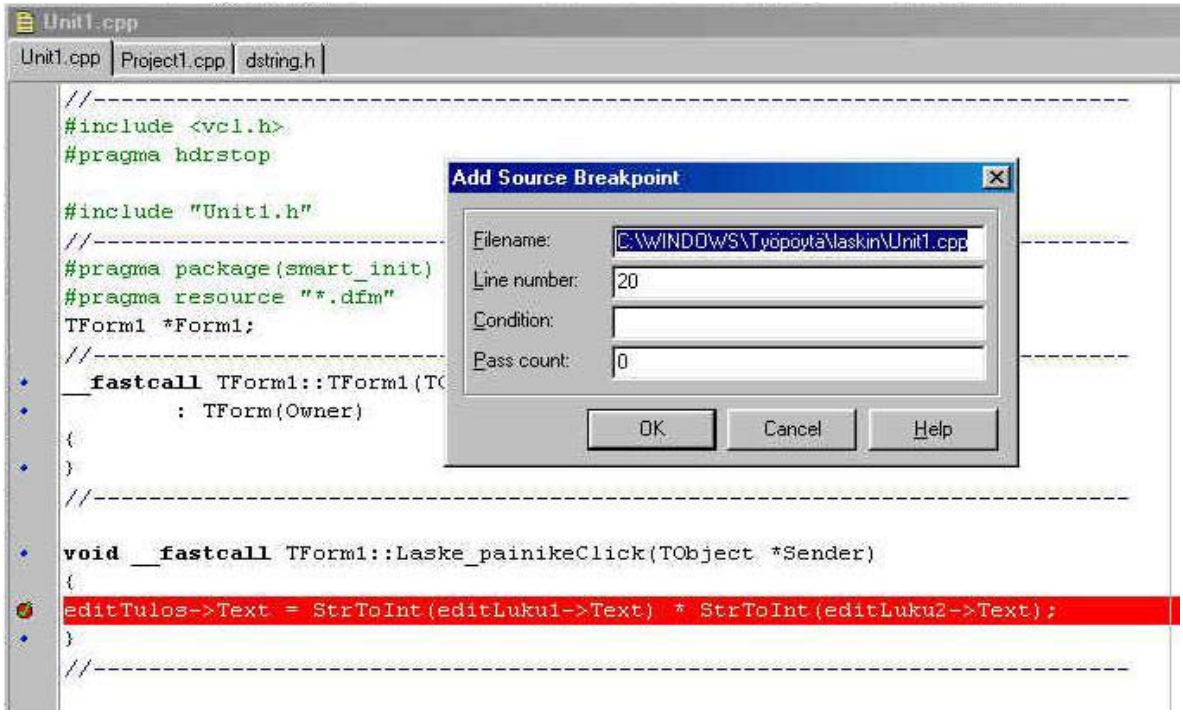
7.1.1 Keskeytysloukku (breakpoint/trap)

Keskeytysloukku tarkoittaa lähdekoodissa valittua riviä, jonka kohdalle ohjelman suoritus pysähtyy keskeytysloukun lauetessa esimerkiksi saavuttaessa koodiriville, koodirivillä olevan muuttujan saavuttaessa halutun arvon tai saavuttaessa riville ennalta määrätyn toistokerran jälkeen. Näin koodirivillä olevien muuttujien arvoja on mahdollista tarkastella. Ohjelman suorituksen edettyä keskeytysloukun kohdalle, keskeytetään ohjelman ajo ja korostuspalkki osoittaa, missä lauseessa ohjelman suoritus on. Tämän jälkeen eri muuttujien arvoja voidaan tutkia, jatkaa koodin suoritusta rivi kerrallaan eli askeltaen tai jatkaa ohjelman suoritusta seuraavaan keskeytysloukkuun. On huomattava, että keskeytysloukut ovat koodissa vain virheiden etsinnän aikana eli niitä ei tallenneta esimerkiksi ajettavaan .exe-

tai .dll-tiedostoon käännettäessä ohjelmasta ajettava versio. On myös otettava huomioon, että keskeytysloukkujen käyttö voi vaikeuttaa muiden testattavaan järjestelmään kuuluvien normaalinopeudella toimivien osajärjestelmien toimintaa eli keskeytysloukkuja ei pystytä käyttämään läheskään kaikkiin testitapauksiin.

7.1.2 Keskeytysloukkujen asettaminen

Keskeytysloukkujen asettaminen testattavalle ohjelmalle tapahtuu kyseisessä ohjelmointiympäristössä yksinkertaisesti naputtamalla haluttua lähdekoodin riviä editori-ikkunassa, jossa kyseinen ohjelma on auki tai syöttämällä haluttu rivinumero ja mahdollisesti muita tarvittavia arvoja kunkin tyyppisen keskeytysloukun omassa editointi/ominaisuus-ikkunassa. Keskeytysloukun poistaminen tapahtuu päinvastaisesti napauttamalla halutun poistettavan keskeytysloukun koodiriviä. Keskeytysloukkuja ei kuitenkaan ole järkevää sijoittaa sellaisille riveille, joilla ei ole suorittavaa lausetta tai joissa arvoille ei tapahdu mitään, esimerkiksi kommenttiriveille tai muuttujien esittelyjä sisältäville riveille. Seuraavaksi lyhyt esittely C++-Builderissa olevista erityyppisistä keskeytysloukkutyökaluista (kts. Kuva 15) ja niiden käytöstä: keskeytysloukun asettaminen, arvojen seuranta keskeytysloukkujen avulla ja koodissa eteneminen (kts. Taulukko 6).



Kuva 15. Keskeytyksen asettaminen lähdekoodiriville eli *Add Source Breakpoint* -ikkuna.

Source Breakpoint -keskeytysloukulla kyetään tarkastelemaan koodirivillä tapahtuvien käskyjen vaikutusta ohjelmassa määriteltyjen muuttujien arvoihin. Keskeytysloukku laukeaa ohjelman suorituksen tullessa valitulle riville.

Taulukko 6. *Add Source Breakpoint Properties* -dialogin avulla kyetään editoimaan ja määrittelemään seuraavia koodiriville asetetun keskeytysloukun ominaisuuksia.

Filename	Tiedostonimi, johon keskeytysloukku on asetettu.
Line Number	Koodirivi, johon keskeytysloukku on asetettu.
Condition	Ehtolause, jonka toteutumista tarkastellaan joka kerta keskeytysloukkuun saavuttaessa, ehtolauseella voidaan esimerkiksi pysäyttää ohjelman suoritus vasta, kun tietty muuttuja saavuttaa halutun arvon.
Pass Count	Laskuri, jonka avulla kyetään laukaisemaan keskeytysloukku vasta halutulla suorituskerralla.

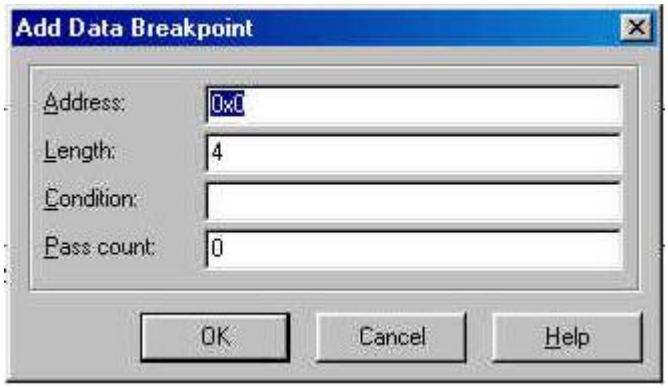
Address Breakpoint -keskeytysloukulla kyetään tarkastelemaan kutakin koodiriviä vastaavien konekielisten käskyjen muistipaikkojen arvoja (kts. Kuva 16 ja Taulukko 7), jotka ovat nähtävissä koottuna erillisessä ohjelmaa ajavan koneen rekisterien arvoja kuvaavassa CPU-ikkunassa.

Kuva 16. Keskeytyksen asettaminen muistiosoitteeseen eli *Add Address Breakpoint* -ikkuna.

Taulukko 7. *Add Address Breakpoint Properties* -dialogin avulla kyetään editoimaan seuraavia koodiriville asetetun keskeytysloukun ominaisuuksia.

Address	Määrittelee keskeytysloukun muistiosoitteen.
Condition	Ehtolause, jonka toteutumista tarkastellaan joka kerta keskeytysloukkuun saavuttaessa, ehtolauseella voidaan esimerkiksi pysäyttää ohjelman suoritus vasta kun tietty muuttuja saavuttaa halutun arvon.
Pass Count	Laskuri, jonka avulla kyetään laukaisemaan keskeytysloukku vasta halutulla suorituskerralla.

Data Breakpoint -keskeytysloukulla kyetään tarkastelemaan kutakin koodiriviä vastaavien konekielisten käskyjen muistipaikan arvoja keskeytysloukun lauetessa muistipaikkaan kirjoitettaessa (kts. Kuva 17 ja Taulukko 8).

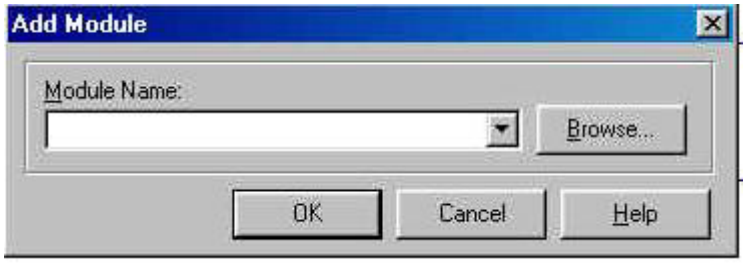


Kuva 17. Keskeytyksen asettaminen muistipaikkaan ehdollisesti eli Add Data Breakpoint -ikkuna.

Taulukko 8. *Add Data Breakpoint Properties* -dialogin avulla kyetään editoimaan seuraavia koodiriville asetetun keskeytysloukun ominaisuuksia.

Address	Määrittelee keskeytysloukun muistiosoitteen, myös muuttujien nimet käyvät osoittamaan vastaavaa muistipaikkaa.
Length	Määrittelee perustietotyypeille automaattisesti laskettavan keskeytysloukun osoitteen pituuden muistipaikassa.
Condition	Ehtolause, jonka toteutumista tarkastellaan joka kerta keskeytysloukkuun saavuttaessa, ehtolauseella voidaan esimerkiksi pysäyttää ohjelman suoritus vasta, kun tietty muuttuja saavuttaa halutun arvon.
Pass Count	Laskuri, jonka avulla kyetään laukaisemaan keskeytysloukku vasta halutulla suorituskerralla.

Module Load Breakpoint -keskeytysloukulla kyetään tarkastelemaan muuttujien arvoja keskeytysloukun lauetessa ladattaessa ohjelman ulkopuoleinen moduuli eli esimerkiksi Windowsin dynaamisesti linkitetty kirjasto (dll-tiedosto) ensimmäistä kertaa muistiin (kts. Kuva 18 ja Taulukko 9).



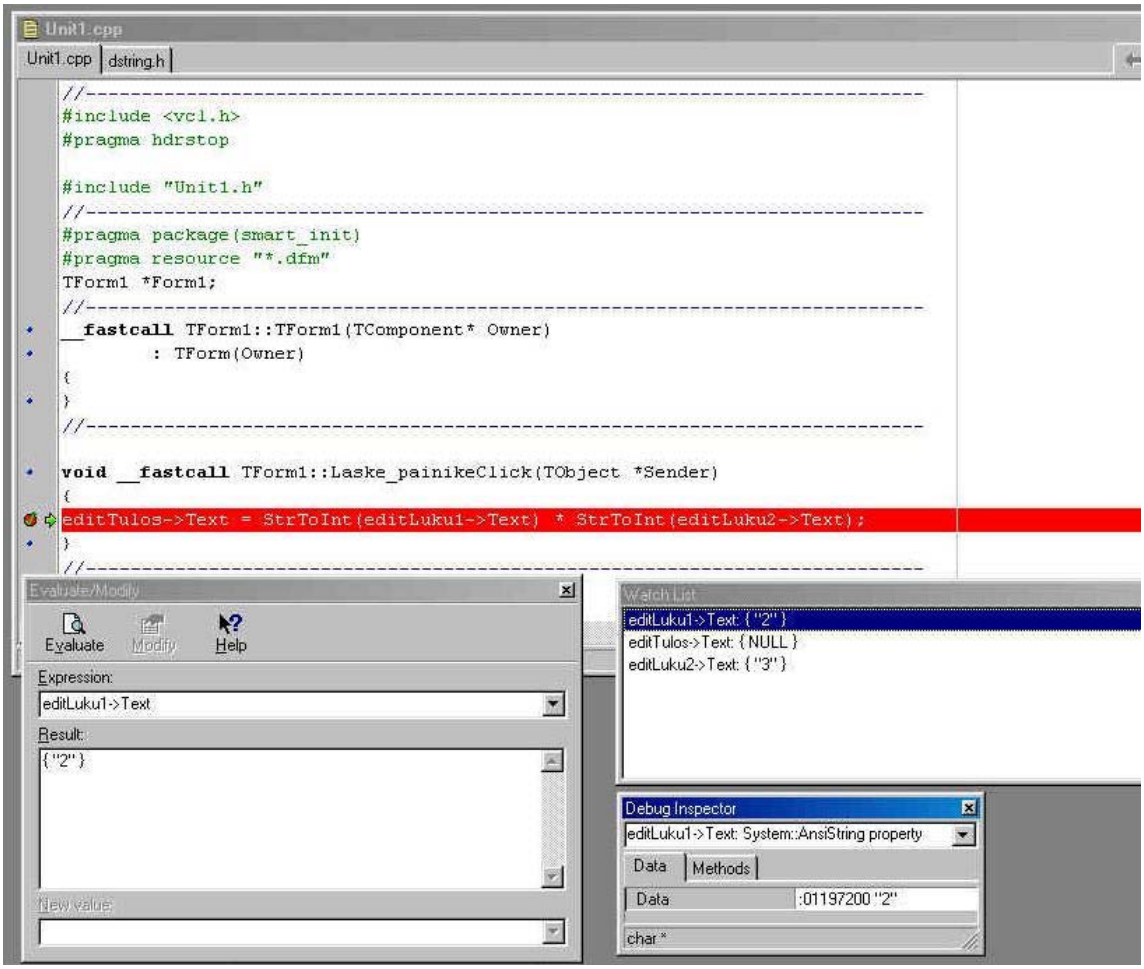
Kuva 18. Keskeytyksen asettaminen moduulin lataukseen eli *Add Module* -ikkuna.

Taulukko 9. *Add Module* -dialogin avulla kyetään määrittelemään ohjelman ulkopuoleiset moduulit, jotka laukaisevat keskeytysloukun:

Module name	Muistiin ladattavanmoduulin nimi (esim. .dll/.bpl).

7.1.3 Keskeytysloukkujen käyttö

Keskeytysloukun laukeamisen jälkeen on testaajalla käytössä erilaisia virheidenetsintämahdollisuuksia. Arvoja ja kutsuja päästään tarkastelemaan käyttämällä askellusta, tutkimalla ja muuttelemalla muuttujien arvoja, lisäämällä lausekkeita *Watch list* -, *Debug Inspector*- ja *Evaluate/Modify* –työkaluikkunoiden avulla (kts. Kuva 19). Lähdekoodiin voidaan myös lisätä uusia keskeytysloukkuja ja poistaa entisiä ajon aikana.



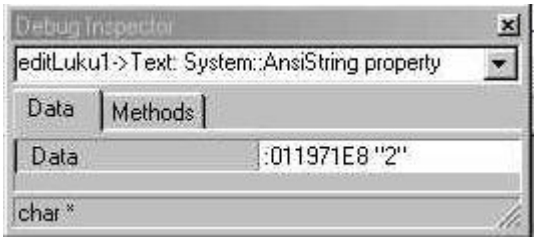
Kuva 19. *Evaluate/Modify* -, *Watch list* – ja *Debug Inspector* -työkalut lähdekoodi-ikkunan päälle asetettuna ohjelman suorituksen ollessa pysäytettynä funktiokutsuun.

Keskeytysloukkujen asettamisen jälkeen, voidaan testauksen ajo käynnistää (Run). Ohjelman ajo tapahtuu normaalisti, kunnes suoritus saavuttaa ensimmäisen keskeytysloukun. Keskeytyksen kohdalla ohjelman suoritus pysähtyy tarvittavan lähdekielisen tiedoston ollessa luettuna muistiin. Kohdistinpalkki näyttää koodieditori-ikkunassa keskeytysloukun sisältävän rivin ja erillisellä nuolella osoitetaan, millä rivillä ohjelman suoritus on menossa. Keskeytysloukkujen kanssa voidaan käyttää kaikkia virheidenetsintäkomentoja. Sovelluskehitysympäristö sisältää edellä mainittuja virheenpoistotyökaluja, joiden toiminnat ovat osittain ominaisuuksiltaan päällekkäisiä. Niiden kaikkien tärkeimpänä tehtävänä on päästä seuraamaan eri muuttujien ja lausekkeiden arvoja ohjelman suorituksen aikana, suorituksen edetessä käyttäjän haluamalla tavalla ja nopeudella. Suorituksen etenemisen hallitsemiseen voidaan käyttää askellusta. Lähdekoodissa olevien muuttujien arvoja voidaan tutkia ja vaihtaa, lausekkeita ja muuttujia voidaan editoida työkaluikkunoissa. Esimerkiksi halutun muuttujan arvoa kyseisellä hetkellä päästään tarkastelemaan yksinkertaisimmillaan viemäl-

lä kursori muuttujan nimen ylle lähdekoodi-ikkunassa, jolloin ohjetekstikenttään ilmestyy muuttujan sen hetkinen arvo (esim. editLuku1->Text = {data: "2"}) tai erinäisten arvojen muuttumista helpottavien työkalujen avulla kuten kuvassa 19 olevilla työkaluilla, joiden toimintaa kuvataan seuraavassa luvussa.

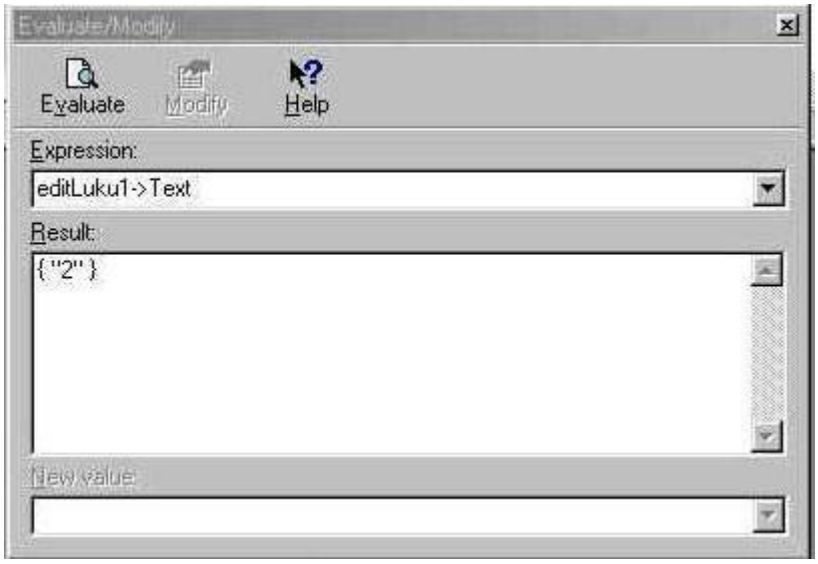
7.1.4 Arvojen seuranta keskeytysloukkujen avulla

Muuttujien arvoja ja muistipaikkoja, joihin muuttujien arvoja on talletettu voidaan seurata seuraavien kolmen eri toiminnon avulla.



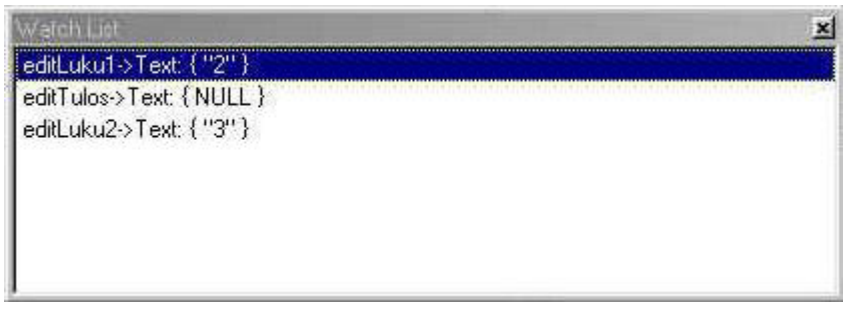
Kuva 20. *Debug Inspector* -työkalun ikkuna.

Debug Inspector –työkalun avulla kyetään seuraamaan muun muassa seuraavia tietotyyppejä: taulukoita, luokkia, vakioita, osoittimia ja rajapintoja. Kuvan 20 tilanteessa Debug Inspector -ikkunassa on tutkittavana editLuku1 -kentän ominaisuus Text, johon on talletettu arvo "2".



Kuva 21. *Evaluate/Modify* -työkalun ikkuna.

Evaluate/Modify -työkalulla (Kts. Kuva 21) kyetään, samalla tavalla *Object Inspector* –ikkunassa, seuraamaan muuttujien arvoja kullakin suorituksen hetkellä, mutta myös kokonaisia lauseita kyetään evaluoimaan samalla kertaa. *Evaluate/Modify* -ikkunassa kyetään myös muokkaamaan testattavien muuttujien arvoja kesken suorituksen haluttaessa. Kuvan tilanteessa *Evaluate/Modify* -ikkunassa on tutkittavana *Expression* -kenttässä määritelty muuttuja eli editLuku1 -kentän ominaisuus (property) Text, johon on talletettu ominaisuuden sen hetkinen arvo, joka saadaan tarkasteltavaksi *Result* -kenttään. Expression-kentässä määritellyn muuttujan arvoa päästään muuttamaan kesken debuggaamisen asettamalla muuttujan uusi arvo *New Value* -kenttään.



Kuva 22. *Watch list* -työkalun ikkuna.

Watch list -ikkunaan (Kts. Kuva 22) päästään lisäämään useampia eri muuttujia, joiden arvoja ja niiden muuttumista kyetään seuraamaan ohjelman suorituksen edetessä esimer-

kiksi askeltaen. *Watch list* -työkaluun asetettujen tarkasteltavien muuttujien ja lausekkeiden arvot tulostuvat ikkunaan suorituksen kulloisellakin hetkellä.

7.1.5 Koodissa eteneminen

Ohjelmaa voidaan suorittaa rivi kerrallaan ja pysähtyä sen jälkeen tarkastelemaan rivillä suoritettavien toimenpiteiden tuloksia. Kuitenkaan kaikissa tapauksissa ei ole järkevää käydä läpi kaikkia osia ohjelmakoodissa esimerkiksi jos läpikäytävä koodi ei vaikuta virheitä aiheuttavaan kohtaan millään tavalla. Toisaalta, joissakin tapauksissa askellus täytyy suorittaa huomattavasti yksityiskohtaisemmin, jopa seuraamalla ohjelman suoritusta konekielisellä tasolla. Tällaisissa tapauksissa on askellus toteutettava eri tavoilla, joita sovelluskehitysympäristön virheenpoistotyökaluvalikosta löytyy useita erilaisia. Esimerkiksi ohjelman kutsuessa aliohjelmaa, tulee voida valita, suoritetaanko kutsu yhdellä kertaa, jolloin päästään tarkastelemaan suoraan aliohjelmakutsun aikaansaamia tulosteita, vai käydäänkö myös aliohjelma läpi rivi riviltä. Askeltamisen aloittamisen ja lopettamisen kohta lähdekoodissa on oltava testaajan määriteltävissä ja normaalin ajon jatkaminen on myös oltava hallittavissa. Taulukossa 10 on esitelty askelluksessa tarvittavia komentoja ja selitetty lyhyesti kunkin toiminnon sisältö.

Taulukko 10. Virheenjäljityskomentoja edettäessä koodissa käyttäen askellusta.

Step Over	Ohjelma suoritetaan koodirivi kerrallaan. Kaikki koodirivillä määritellyt suoritettavat toiminnot toimitetaan kerralla eli ohjelman rivi suoritetaan, mutta aliohjelmaan ei siirrytä.
Trace Into	Ohjelma suoritetaan koodirivi kerrallaan. Kaikki koodirivillä määritellyt suoritettavat toiminnot toimitetaan erikseen seuraamisen helpottamiseksi esim. muuttujien alustus muuttujan omassa luokassa eli aliohjelman kutsu aiheuttaa hypyn ko. aliohjelmaan.
Trace to Next Source Line	Ohjelma suoritetaan koodirivi kerrallaan. Kaikki koodirivillä määritellyt toiminnot suoritetaan ja siirrytään seuraavalle koodiriville, aliohjelmakutsut suoritetaan normaalia vauhtia.
Run to Cursor	Ajaa ohjelman suorituksen riville, jolla kohdistin on eli asettaa keskeytyksen riville, jolla kursori on koodieditorissa.
Run Until Return	Ajaa ohjelmaa, kunnes suoritus palautuu sen hetkisestä funktiosta takaisin.

Aliohjelmien kutsujen seuraamiseen, aliohjelmakutsujen parametrien tarkistamiseen ja niissä esiintyvien virheiden jäljittämiseen tarkoitettu työkalu on esitellyssä sovelluskehi-

tysympäristössä *Call Stack* –työkalu (kts. Kuva 23). Ikkunassa näkyvät kaikki aliohjelma- ja funktio-kutsut sekä niiden parametrien arvot, jotka muuttuvat automaattisesti ohjelman suorituksen aikana. Aliohjelmien nimet ja kutsuissa välitetyt parametrit ilmestyvät ikkunaan ohjelman etenemisen myötä, kunkin kutsun suorittamisen yhteydessä.

Kuva 23. Call Stack -työkalun ikkuna.

8. AUTOMATISOITU TESTAUS

Testauksen automatisoinnin työkaluilla tarkoitetaan tässä yhteydessä niin kutsuttuja testauksen suoritus- ja vertailutyökaluja (test execution and comparison tools) [Few99], muita samoista työkaluista käytettyjä nimityksiä ovat esimerkiksi *nauhoitus/toisto –työkalut* (capture/replay -tools). Periaatteeltaan nämä testityökalut ovat yksinkertaisesti ohjelmia, jotka suorittavat testin käyttämällä testattavaa sovellusta eli käytännössä simuloivat tavallista testaajaa ja testaajan testattavalle sovellukselle manuaalisesti suorittamia toimintoja. Testityökalu suorittaa testattavalle ohjelmistolle sille määriteltyjä toimintoja samalla etsien virheitä ohjelman vasteista. Testauksen automatisoinnissa testityökalu on testaajan työtä helpottava apuväline, jolla voidaan nopeuttaa ja vähentää testaajan työtä. Testityökalun avulla päästään myös parempaan testikattavuuteen työkalun mahdollistaessa suuremmat testiaineistot verrattuna testin manuaaliseen suorittamiseen. Oikein suunnitelluilla testitapauksilla ja testityökalujen käytön kohdentamisella mahdollistetaan testaukseen panostettujen resurssien tehokkaampi käyttö. Automatisoidun testauksen implementoinnilla voidaan huomattavasti vähentää testaukseen käytettyä työmäärää. Joissakin yrityksissä on raportoitu saavutetun jopa 80% vähennys testaukseen käytetyissä resursseissa [Few99].

Teoria testauksen työkalujen ja testauksen automatisoinnin osalta on samaa, kuin perinteisin menetelmin testattaessa: testityökalujen avulla suoritetaan testattavalle sovellukselle samoja testejä kuin manuaalisessakin testauksessa. Testauksen käytännön osalta on kuitenkin ongelmana suurten ohjelmistojen testikattavuuden saaminen vaaditulle tasolle, johtuen ohjelmien syötteiden ja toimintojen usein muodostamista lukemattomista kombinaatioista. Tähän ongelmaan testityökalut tuovat osittain helpotusta, testityökalut mahdollistavat suu-

rempien testiaineistojen käytön, testien nopeamman suorittamisen ja saatujen tulosten tarkistamisen. Työkalut tekevät myös vähemmän virheitä toimittaessa suurten testiaineistojen kanssa.

9. AUTOMATISOIDUN TESTAUKSEN SUORITTAMINEN

Testauksen automatisointityökalu suorittaa testattavalle sovellukselle samat testaustoimenpiteet kuin testaaja normaalistikin suorittaisi. Työkalulla suoritettavat testit on kirjoitettu työkalun omalla testiskriptikielellä, joka sisältää yleensä tavallisimmat ohjelmointirakenteet: toistot, haarautumat, funktiokutsut, muuttujat ja muuttujien väliset operaatiot.

9.1 Testiskriptit

Testiskriptit ovat sarja käskyjä ja ohjeita testaustyökalulle siitä, mitä testattavalle ohjelmalle tulisi tehdä halutun testin aikaansaamiseksi. Lisänä skripteissä on logiikkaa testattavalta ohjelmalta saatujen vasteiden tulkitsemiseksi ja saatujen paluuarvojen ja tulosteiden tallentamiseksi niin tulosten tarkasteluun kuin tulosten varmentamiseen siihen tarkoitetulla vertailutyökalulla.

Testiskriptikielet ovat testaustyökalujen ohjelmointikieliä, jotka sisältävät samanlaista logiikkaa kuin tavalliset ohjelmointikielet. Testiskriptit ovat testaustyökalujen testiskriptikielellä toteutettuja testien suorittamiseen käytettyjä ohjelmakoodeja. Skriptit sisältävät käskyjä, tietoa ja ohjeita testaustyökalulle testin suorituksen etenemisestä esimerkiksi: mitä syötetään mihinkin kenttään tai mitä painiketta painetaan seuraavaksi.

Testiskriptikielessä ja skriptien ajossa tarvittavia ominaisuuksia ovat esimerkiksi [Few99]:

- synkronointi (esim. seuraavan syötteen syöttöajankohta),
- tietoa vertailusta (mitä tulosteiden tulisi olla, minkä kanssa tulosteita verrataan),
- mitä tietoa ruudulta tulee kaapata ja mihin se talletetaan tai verrataan,
- milloin lukea tietoa muulta laitteelta ja mistä lukea sitä (esimerkiksi tiedostosta, tietokannasta tai joltain muulta oheislaitteelta) ja
- kontrollitietoa (esimerkiksi testin toistojen määrä tai päätöksenteko ohjelman tulosteiden perusteella).

Testiskriptejä pystytään luomaan pelkästään testaustyökalun nauhoitus/toisto -välineen avulla, mutta useimmiten järjellisten testien aikaansaamiseksi tulee testiskriptit ohjelmoida työkalulle. Testin suorituksen nauhoituksen avulla saadaan kuitenkin usein helpommin ja nopeammin luotua tarvittava pohja koodille testiskriptin aikaansaamiseksi. Nauhoitus/toisto –apuvälinettä voidaan useimmissa tapauksissa käyttää avuksi luotaessa raamit varsinaiselle testiskriptille. Nauhoitus toimintoa hyväksikäyttäen päästään nopeammin luomaan varsinaista testitapausta. Käyttäjän suorittamiin ja nauhoitettuihin toimintoihin esimerkiksi: kenttien syöttöihin, lomakkeiden avauksiin jne. kyetään lisäämään testaustyökalussa monimutkaisempaa logiikkaa, editoimalla nauhoitettua skriptiä, lisäämällä skriptiin esimerkiksi tulosteiden tarkistuksia, toistoja eri syötteillä tai muita testattavalle ohjelmalle suoritettavia toimintoja.

Testaustyökalun nauhoitus/toisto -toimintoa voi verrata tekstieditorin leikkaa/liimaa toimintoon, se helpottaa itse testiskriptin tekemistä, mutta toiminnan logiikka on lisättävä itse puhtaasti käsin koodaten. Testiskripteihin voidaan lisätä myös mahdollisuus tulostaa ajonaikana tapahtuvia virheitä selvittäviä kommentteja, jotka testauksen suorituksen aikana kirjoitetaan testityökalun tekemään testiloki-tiedostoon.

Testaustyökalun käytössä aikaavievintä on testiskriptien koodaus. Ei ole järkevää luoda automatisoituja testejä, joita tullaan käyttämään vain kerran ohjelmiston testaukseen. Useimmissa tapauksissa kertakäyttöisen testin suorittaminen manuaalisesti on huomattavasti nopeampaa, kuin testin koodaaminen skriptikielelle. Testiskriptien tekemisessä on valittava kohteet, joissa testaustyökalun käyttö on järkevintä ja joissa työkalun käyttö tulee säästämään testaajan aikaa ja siten myös ohjelmistotuotantoprosessille määrättyjä resursseja. Parhaiten työkalut soveltuvat testaamaan ohjelmistoja ja ohjelmistojen osa-alueita, joiden toimintaan liittyy paljon toistoa, esimerkiksi mahdollisten syötekombinaatioiden testaamiseen. Järkevää on automatisoida ohjelmistoversioiden regressiotestaus eli testi, joka tullaan ajamaan jokaisen samaan ohjelmistoversioon suoritetun korjauksen tai päivityksen jälkeen.

Testiskriptien kirjoittamisen viemän ajan takia on oleellista, että testiskriptit ovat mahdollisimman suurelta osin uudelleenkäytettäviä. Koodin uudelleenkäytettävyyden varmistamiseksi tulisi käyttää samoja keinoja kuin tavallisissa ohjelmointimenettelyissä. Testikoodin tulisi muodostua mahdollisimman pienistä ja abstrakteista ohjelmistokomponenteista, joita

liittämällä yhteen muodostetaan varsinainen testiskripti. Esimerkiksi uudelleenkäytettävästä skriptikomponentista voidaan ottaa monien ohjelmien sisältämät useat päivämääräkentät, joiden toimintalogiikka on samanlainen ja joihin käytettävät syötteet ovat samanmuotoisia. Tällaisiin tapauksiin on järkevää luoda päivämääräkenttien tarkastukseen tarkoitettu oma testiskriptiohjelmakomponentti. Päivämääräkenttientarkistuskomponentti tunnistaa päivämääräkentän ohjelman graafisen käyttöliittymän kuvauksen sisältävän GUI mapin perusteella. Testi suoritetaan päivämääräkentälle esimerkiksi parametrinä välitettävän tarkistettavan kentän nimen avulla, josta tarkemmin seuraavassa aliluvussa. Useita lomakkeita sisältävässä graafisessa sovelluksessa eri lomakkeet tunnistetaan myös GUI mapien perusteella, joten testityökalu pystyy avaamaan haluamansa lomakkeen ja suorittamaan määrätyille kentille halutut testit käyttäen yhtä ja samaa komponenttia.

9.2 Graafisen käyttöliittymän kautta testaaminen (GUI mapit)

GUI (graphical user interface) mapilla tarkoitetaan graafisessa käyttöliittymässä lomakkeella olevien komponenttien kuvauksia ja koordinaatteja eli niiden sijaintia sovellusikkunassa, minkä mukaan testaustyökalu osaa suorittaa testiskriptissä määritellyt toimenpiteet oikeille ikkunakomponenteille. Esimerkiksi koodissa Button_Press("Ok") suorittaa Ok -painikkeen painamisen määrätyllä lomakkeella. GUI mapit talletetaan tiedostona, joka ladataan suoritettavan testin käyttöön testin ajon aikana. Työkalut tunnistavat graafiset komponentit lomakkeelta ja luovat valmiita GUI mapeja, joita on myös mahdollisuus päästä editoimaan: esim. nimeämällä komponentteja loogisemmin testiskriptien koodausta varten tai poistamalla tiedostosta testin suorituksen kannalta tarpeettomia komponentteja.

Yhden ohjelmistotalon sisällä päivämääräkomponentteja voi olla käytettynä useissa eri ohjelmistoissa ja usein niiden toiminta on myös standardisoitu vähintään yrityksen sisällä. Testaaminen graafisen käyttöliittymän kautta mahdollistaa saman skriptikomponentin käytön saman toiminnallisuuden omaavien ohjelmistokomponenttien testaukseen, jopa laitteistoympäristöstä riippumatta. Esimerkeissä käytetty WinRunner 6.0 –testityökalulla kyetään testaamaan niin kaikkia Windows-ohjelmistoja kuin myös selainpohjaisia Web-käyttöliittymiä. Eri ympäristöt vaativat kuitenkin erilliset GUI mapit ja joissakin tapauksissa muita lisäkomponentteja testialustaan eri ympäristöjen komponenttien tunnistamiseksi oikein.

9.3 Automatisoidun testijärjestelmän luonti

Kuten edellä mainittiin testausta ei kannata automatisoida vain kertakäyttöisiin testeihin, testien automatisoinnin vaatiman tavallista testausta suuremman työmäärän takia, tiettyjä erikoistapauksia lukuunottamatta. Tällaisesta tapauksesta esimerkkinä voidaan ottaa valtavan testiaineiston syöttäminen hyvin yksinkertaisen käyttöliittymän kautta sovellukselle. Testausta kannattaa automatisoida vasta kun testin manuaalisen suorittamisen työmäärä ylittää testiskriptin luomiseen menevän ajan ja yleensä yhdellä tai vain muutamalla testauksen suorituskerralla tähän ei päästä.

Testien automatisointi on hyödyllisintä testaajalta paljon aikaa vievillä, paljon manuaalista työtä vaativilla, osa-alueilla. Esimerkiksi, kun kyseessä on suurien mekaanisten testien suorittaminen tai kyseessä on testi, jota tullaan tarvitsemaan toistuvasti, on vakavasti harkittava testauksen suorituksen toteuttamista jollakin työkalulla. Automatisoitua testijärjestelmää luotaessa on tärkeää tunnistaa testattavasta sovelluksesta osa-alueet, joihin voidaan soveltaa tarkoitukseen sopivaa testityökalua. Kaikkia testejä ei voida suorittaa testityökaluilla testattavan sovelluksen ennalta arvaamattomuuden ja testityökalujen asettamien rajoitusten takia. Testattavasta sovelluksesta kannattaa hakea ohjelmiston kannalta tärkeimmät ja useimmin käytetyt toimintopolut, joiden eteneminen on sovelluksen ja sen käyttäjän kannalta oleellisinta ja tutkia ovatko valitut toimintoketjut automatisoitavissa ehkä jo käytössä olevalla testaustyökalulla vai, onko edellä mainittuja tapauksia varten kannattavaa hankkia niihin sopivampi testityökalu. Toimintoketjuista tulee etsiä mahdollisimman samankaltaisia toimintosekvenssejä, pienempiä kokonaisuuksia, joita varten luodaan edellä kuvatun uudelleenkäytettävyyden periaatteiden mukaan omat skriptikomponenttinsa, jolloin säästetään paljon aikaa testiskriptikokonaisuuksien luomisessa myös tulevaisuudessa.

9.4 Testitapauksen luonti WinRunner 6.0 -ympäristössä

Esimerkkinä yksinkertaisen testin luomisesta WinRunner 6.0 –testaustyökalulle ja työkalun käyttämästä Test Script Languagesta (TSL) eli testin ohjelmointikielestä, luodaan aikaisemmin testatulle Kertolasku-sovellukselle (kts. Kuva 14) mustalaatikkoperiaatteen mukainen automatisoitu testi. Testillä verifioidaan sovelluksen ainoaa toimintoa eli kertolaskun toiminnallisuutta. Testissä tullaan soveltamaan luvussa kolme esiteltyjen mustalaatikkotestauksen suoritusmenetelmien: aluetestauksen, ekvivalenssiosituksen ja rajaarvoanalyysin yhdistelmää.

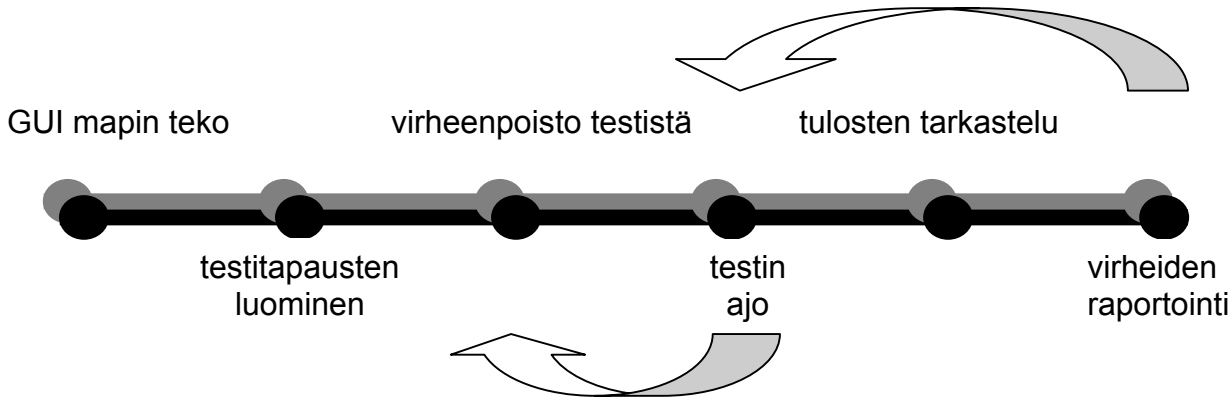
Tehdyssä testissä sovellukselle syötetään kokonaislukuja testiaineiston sisältävästä tietokannasta, tai tässä tapauksessa yhdestä testiä varten luodusta Excel-taulusta, ja tarkastellaan ohjelman toiminnan oikeellisuutta vertaamalla laskutoimituksesta saatuja tuloksia testikannassa oleviin varmennettuihin oikeisiin laskutoimitusten tuloksiin. Lopuksi käyttäjälle näytetään testin tulokset testityökalun testin ajon aikana generoiman raportin muodossa. Kyseistä testimuotoa, jossa testityökalu hakee testiaineiston ja varmennetut tulokset toiminnan oikeellisuuden selvittämiseksi tietokannasta tai muusta tietolähteestä kutsutaan *Data Driven* –testiksi eli datan ohjaamaksi testaukseksi [Few99]. Datalla tarkoitetaan tässä tapauksessa sovellukselle syötettäviä ja sovelluksen vasteisiin verrattavia arvoja. Toinen tapa luokitella tehtävä testi pohjautuu Cem Kanerin esittämään luokitteluun, joka perustuu tulosten oikeellisuuden arviointiin (evaluation-based). Esimerkin testaustekniikkaa kutsutaan *itse-verifioivaksi testaukseksi* (Self-verifying data test) eli testissä käytetään vertailuarvoja sisältävää dataa ulkopuolisesta tietolähteestä tulosten oikeellisuuden varmistamiseksi. Tässä tapauksessa voitaisiin myös käyttää vertailuarvoina vasteita, jotka on saatu ajamalla testi sovelluksella aikaisemmin, esimerkiksi suorittamalla regressiotestausta. Kyseistä testausmuotoa kutsutaan Kanerin terminologiassa nimellä *Comparison with saved results* eli vertailuksi tallennetuilla arvoilla [Kan01].

Seuraavalla testiesimerkillä tulevat WinRunner työkalun tärkeimmät toiminnalliset ominaisuudet testin tekemisestä sen suorittamiseen esitellyiksi eli: käyttäjän toimintojen nauhoitus- ja editointimahdollisuudet, testiaineiston syöttö ja vertailuarvojen hakeminen ulkopuolisesta lähteestä, testitulosten vertailu ja oikeellisuuden tarkistaminen ja viimeisenä testitulosten raportointi ominaisuudet.

Yksittäisen testin luominen WinRunner-testityökalulle koostuu kuvan 24 mukaisesti kuudesta eri vaiheesta [Win99]:

1) Gui mapin teko:	Työkalun on tunnistettava testattavan sovelluksen käyttöliittymän komponentit, että se kykenee myös käyttämään niitä.
2) Testitapausten luominen:	Testi voidaan luoda nauhoittamalla, ohjelmoimalla tai yhdistelemällä molempia tekniikoita.
3) Virheenpoisto testistä:	Testiä kyetään ajamaan debug-tilassa, kuten missä tahansa sovelluskehitysympäristössä (kts. luku 5.2). Testissä voidaan asettaa keskeytyksiä, tarkkailla muuttujien arvoja ja kontrolloida suorituksen etenemistä.
4) Testin ajo:	Testattavan skriptin ajon aikana suoritetaan testattavalle sovellukselle toimintoja, joiden aiheuttamat vasteet tallennetaan.
5) Tulosten tarkastelu:	Käyttäjä suorittaa arvioinnin testin onnistumisesta työkalun tuottaman virheraportin perusteella.
6) Virheiden raportointi:	Virheraportit voidaan lähettää eteenpäin esimerkiksi yrityksen laatujärjestelmään/tietokantaan.

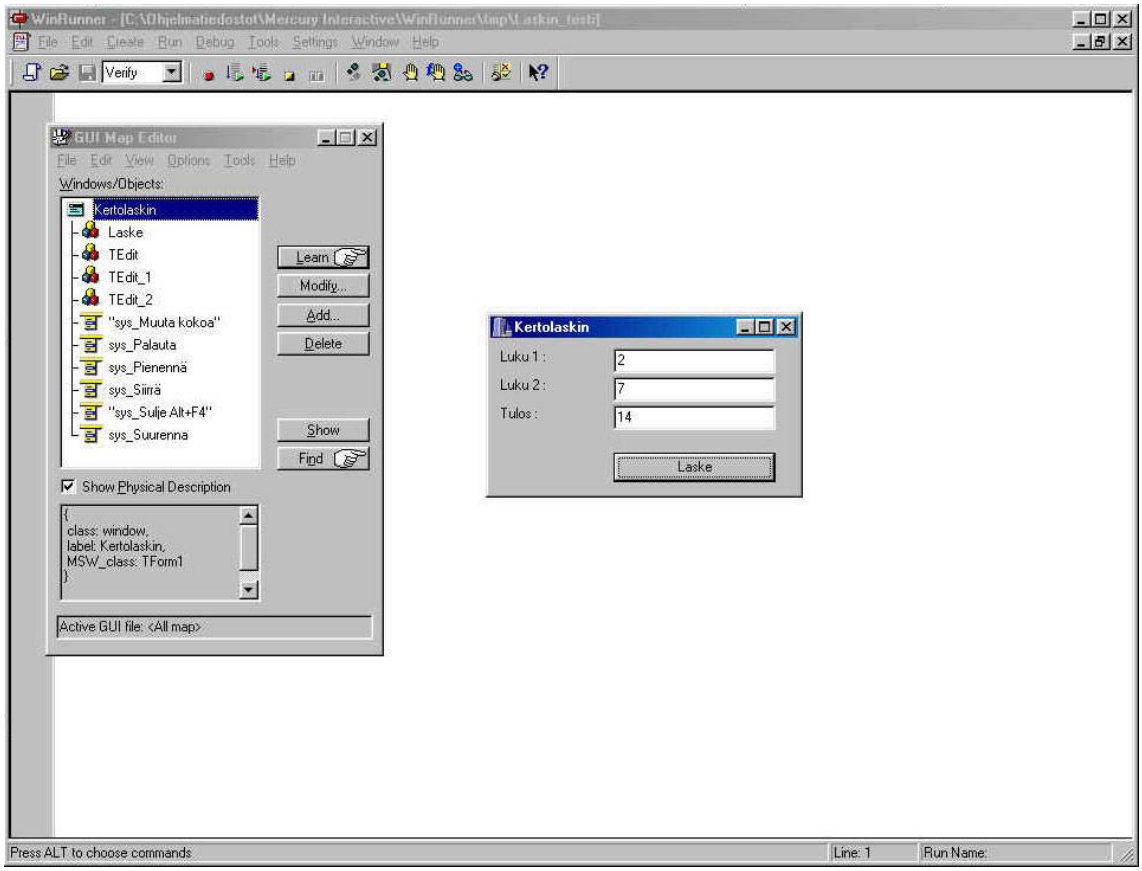
Testin lopullisesti valmistuttua päädytään suorittamaan testausprosessin kolmea viimeistä vaihetta, esimerkiksi ajettaessa toistuvia regressiotestejä sovellukselle. Testiä toteutettaessa joudutaan usein palaamaan virheidenpoistovaiheeseen, koska testin tekijä voi päätyä samoihin virhetilanteisiin kuin millä tahansa ohjelmointityökalulla ohjelmoitaessa.



Kuva 24. Testausprosessi WinRunner-työkalulle.

9.4.1 GUI Mapin tekeminen

Ensimmäisenä testiympäristön (WinRunner) ja testattavan sovelluksen (Laskin) käynnistämisen jälkeen tulee suorittaa testattavan sovelluksen graafisen käyttöliittymän komponenttien tunnistus eli WinRunnerin GUI mapin tekeminen. Komponenttien tunnistus käynnissä olevalta sovellukselta tapahtuu yksinkertaisesti käynnistämällä WinRunnerista graafisten käyttöliittymien hallintaan tarkoitettu työkalu GUI Map Editor (kts. Kuva 25).

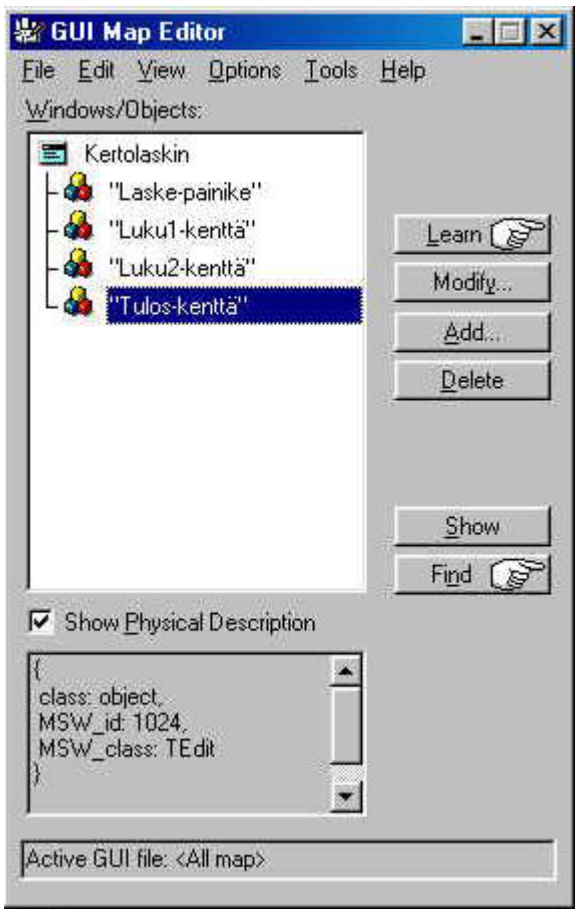


Kuva 25. Esimerkissä käytettävä GUI Map Editor -työkalun ikkuna ja testattava Laskin-sovellus WinRunner -testiympäristön perusikkunan päälle avattuna. Kuvan tilanteessa GUI Map Editorilla on suoritettu Laskin-sovelluksen käyttöliittymän komponenttien tunnistus. GUI Map Editor on nimennyt komponentit oman nimeämiskäytännön mukaan eli Laskin-sovelluksen ikkuna on editori-ikkunassa nimetty sovelluksen nimipalkin mukaan Kertolaskimeksi, Laske-painike on nimeltään Laske ja tekstikentät on nimetty Luku1: TEdit, Luku2: TEdit_1 ja Tulos: TEdit2. Editori-ikkunassa näkyvät myös Windows-sovelluksille ominainen tiedostovalikko nimettynä: ”sys_Muuta kokoa” -, ”sys_Palauta” - jne. toimintoineen, jonka GUI Map Editor on osannut avata, tutkia ja listata sovellusikkunan nimipalkissa Kertolaskin-tekstin vieressä olevasta ikonista.

Kuvassa olevalla työkalulla kyetään suorittamaan graafisen käyttöliittymän komponenttien tunnistus automaattisesti *(Learn*-painike) ja myös tarvittaessa editoimaan saatua kuvaustiedostoa (*Add*-, *Delete*- ja *Modify*-toiminnot). GUI Map Editoria voidaan käyttää myös lomakkeella olevien komponenttien hakemiseen editorin komponenttiluettelosta (*Find*–toiminto) ja komponentin etsimiseen testattavan sovelluksen lomakkeelta (*Show*-toiminto).

9.4.2 GUI Mapin editoiminen

Seuraavana vaiheena tapahtuu saadun GUI Mapin editointi helpommin ymmärrettävään muotoon eli tunnistetut kentät ja painikkeet nimetään vastaamaan hyvän ohjelmointitavan mukaisesti kutakin komponenttia, koska komponentteja tullaan käyttämään kuten muuttujia tavallisessa ohjelmoinnissa, jolle sijoitetaan arvo, eli tässä tapauksessa annetaan arvo syötekenttään. Nimien editointi helpottaa nauhoitettavan testiskriptin ymmärtämistä ja ehkäisee väärinkäsityksiä graafista käyttöliittymää käsiteltäessä kuten oikein suoritettu nimiöinti ohjelmakoodissakin tekee (kts. Kuva 26). Testin kannalta ylimääräisten komponenttien poistaminen komponenttiluettelosta selkeyttä myös saatua käyttöliittymän kuvausta. Komponenttien lisääminen kuvaustiedostoon voidaan suorittaa myöhemminkin uusien ominaisuuksien testaamista varten. Luotavan testin on tarkoitus testata sovelluksen kertolasku funktiota, jolloin tarpeellisiksi komponenteiksi jäävät vain syöttö- ja tuloskentät sekä *Laske*-painike.



Kuva 26. GUI Map editoinnin jälkeen. Komponentit ovat nimetty kuvaavammin niiden nimikenttien (label) ja tyypin (tekstikenttä/painike) mukaan ja testin kannalta tarpeettomat komponentit on poistettu. Uudelleen nimeäminen tapahtuu *Modify* -toiminnolla ja poistaminen *Delete* -toiminnolla.

9.4.2 Skriptin nauhoitus

Gui mapin tekemisen jälkeen on vuorossa WinRunnerin testausprosessissa toinen vaihe eli testitapausten luominen, joka aloitetaan skriptin nauhoituksella. Skriptin nauhoitus tapahtuu yksinkertaisesti käyttämällä testattavaa sovellusta testityökalun tehdessä aikaisemmin tunnistetuille komponenteille suoritettavia tapahtumia. Sovelluksen testaaminen aloitetaan sovelluksen ollessa päällä, jos ei erikseen haluta testityökalun avaavan sovellusta, esimerkiksi vaihtuvilla käynnistysparametreilla. WinRunner -ohjelma pyörii testiympäristössä samanaikaisesti taustalla. Nauhoitus aloitetaan testityökalun valikossa olevalla *Record*-komennolla, jolloin työkalu alkaa tallentaa testattavalle sovellukselle tehtäviä toimenpiteitä ja luo niistä dynaamisesti ajettavaa ja toistettavaa testiskriptiä. Testin nauhoitus keskeytetään haluttujen toimenpiteiden suorittamisen jälkeen *Stop*-komennolla WinRunnerista.

Testinauhoituksessa suoritetaan seuraavat toimenpiteet testattavalle sovellukselle työkalun luoman nauhoituksen kannalta (kts. Kuva 27):

1) Toiminnan kohdistaminen testattavaan sovellukseen napauttamalla hiirellä sovellusikkunaa.

   nauhoitetussa skriptissä:

```
set_window ("Kertolaskin", 8);
```

2) Syötekenttien Luku1 ja Luku 2 tyhjennys niissä mahdollisesti olevista syötteistä eli kentän koko alueen maalaus hiirellä ja Delete-näppäimen painallus näppäimistöltä.

   nauhoitetussa skriptissä:

```
obj_drag ("Luku1-kenttä", 131, 7, LEFT);
win_drop ("Kertolaskin", 47, 18);
obj_type ("Luku1-kenttä","<kDel_E>");
obj_drag ("Luku2-kenttä", 131, 10, LEFT);
win_drop ("Kertolaskin", 39, 47);
obj_type ("Luku2-kenttä","<kDel_E>");
```

3) Toiminta kohdistetaan kenttään Luku1 napauttamalla hiirellä kenttää ja syötetään kenttään arvoksi 3 ja seuraavaksi kohdistetaan toiminto kenttään Luku2 ja syötetään kenttään arvo 4.

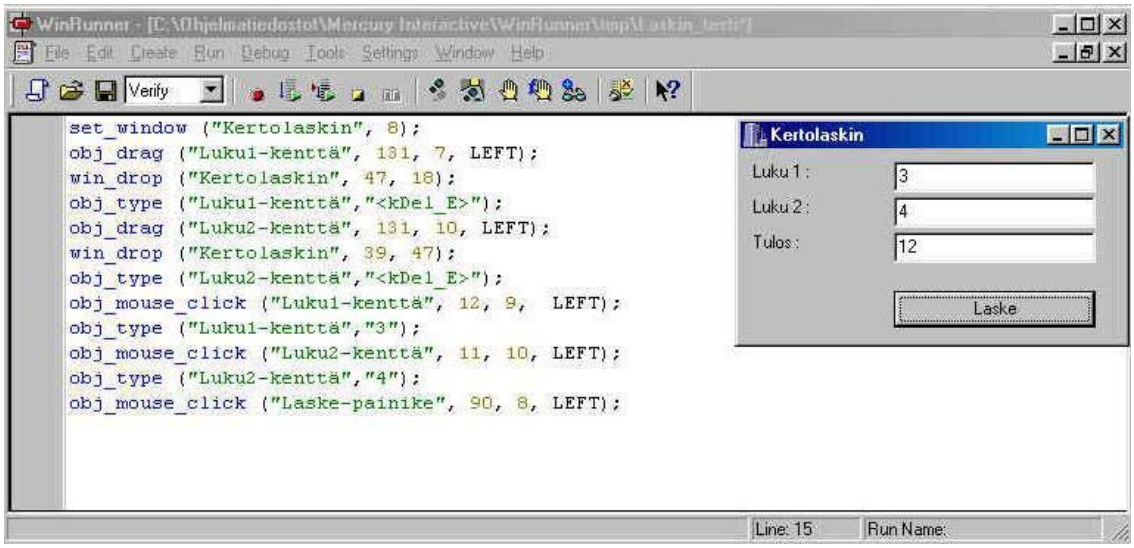
   nauhoitetussa skriptissä:

```
obj_mouse_click ("Luku1-kenttä", 12, 9,  LEFT);
obj_type ("Luku1-kenttä",”3"));
obj_mouse_click ("Luku2-kenttä", 11, 10, LEFT);
obj_type ("Luku2-kenttä",”4"));
```

4) Painetaan Laske-painiketta.

   nauhoitetussa skriptissä:

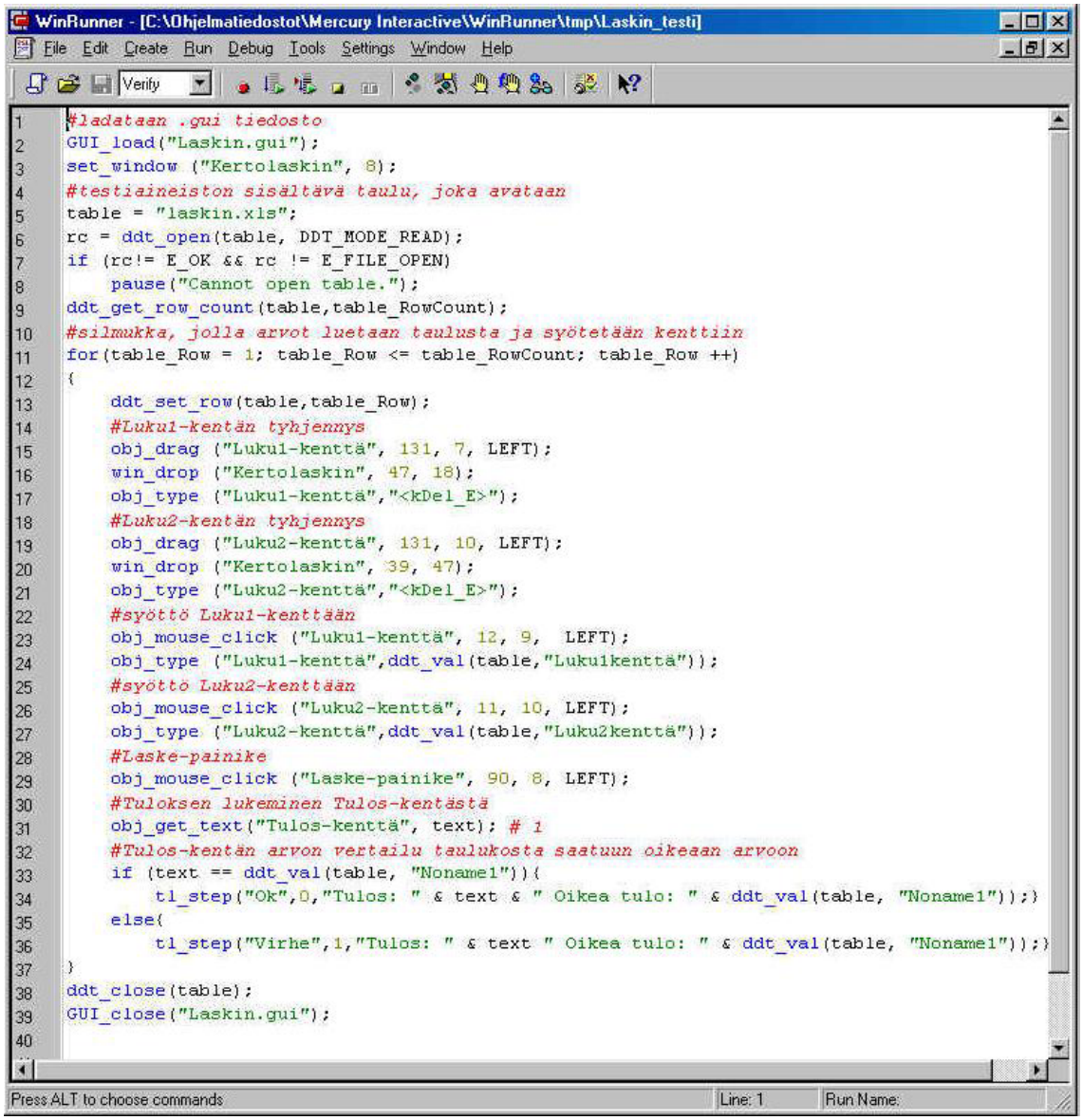
```
obj_mouse_click ("Laske-painike", 90, 8, LEFT);
```



Kuva 27. Nauhoitettu skripti. Kuvassa testattava sovellus testin suorituksen jälkeisessä tilassa WinRunner -ikkunan päällä ja alla WinRunnerin koodieditorissa nauhoituksella aikaansaatu testiskripti.

9.4.4 Nauhoitetun skriptin editointi

Nauhoituksella saatu testiskripti on jo itsestään ajettava valmis testi, mutta on hyödytöntä testata vain kahden kokonaisluvun syöttöä ja painikkeen painallusta testattavalla sovelluksella ilman saatujen tulosten oikeellisuuden tarkastamista ja annettujen syötteiden useampia eri kombinaatioita. Seuraavaksi nauhoituksella saatuun skriptin ”raakaversioon” lisätään toiminnallista logiikkaa eli skriptiä aletaan editoida WinRunnerin koodi-ikkunassa (kts. Kuva 28). Koodiin lisätään talletetun GUI Mapin avauskäsky, syötettävät tiedot parametrisoidaan, luodaan syötteet ja laskutoimituksille oikeat tulokset sisältävä tietokannan taulu, lisätään tarkastelu, vertailu ja raportinti saaduille tuloksille ja lopuksi kaikki edellä mainittu toteutetaan silmukassa, joka toistetaan niin useasti kuin syötettävää dataa riittää sitä sisältävässä taulussa. Eli edellisillä vaiheilla päädytään tekemään testistä Data Driven - tai itse-verifioiva testi, riippuen mitä terminologiaa halutaan käyttää. Samaan aikaan on menossa myös WinRunnerin testausprosessin kolmas vaihe eli virheenpoisto testistä, koska testin koodaajalla on taipumusta päätyä samankaltaisiin virhetilanteisiin kuin millä tahansa ohjelmointityökalulla ohjelmoitaessa.



Kuva 28. Editoitu skripti lopullisessa muodossaan.

Ensimmäisenä lisätään skriptiin oikean GUI Mapin avaavat ja sulkevat käskyt. editoidussa skriptissä (kts. Kuva 28):

avaus:

```
GUI_load("Laskin.gui");.
```

sulkemien

```
GUI_close("Laskin.gui");
```

Testattaessa sovellusta sen käyttöliittymän kautta testaajalla ei ole käytössä tietoa ohjelman sisäisestä rakenteesta, kuten virheenjäljitystyökaluja esittelevässä esimerkissä, vaan ainoas-

taan valmis sovellus. Tällöin mustalaatikkoperiaate on luonteva valinta testin toteutustavaksi. Kertolaskin ohjelmalle annettavina syötteinä on kaksi merkkijonoa (tyyppi: String), jotka konvertoidaan koodissa vastaaviksi kokonaisluvuiksi (tyyppi: Int) funktiolla StrToInt (luvussa 5 olevassa Kertolaskin-sovelluksen ohjelmakoodissa StrToInt(editLuku1->Text) ja StrToInt(editLuku2->Text)). Merkkikonversion takia sovellus ei hyväksy syötteiksi kertolasku toiminnolle muuta kuin numeerisista arvoista muodostuvia kokonaislukuja merkkijonoina, joten esimerkiksi kirjaimien, reaalilukujen tai kokonaislukujen arvoalueen ylittävillä arvoilla syöttö on mahdotonta, eikä niillä voida siten myöskään testata syötteille tehtävien tarkistusten takia. Lähdekoodista tietämättömälle mustalaatikkotestaajalle kyseiset rajoitteet selviävät kokeiluilla.

Testaajalla tässä tapauksessa on tiedossa, että käytettävässä sovelluskehitysympäristössä (Windows 95 ja uudemmat) kokonaisluvut esitetään 32-bittisinä eli lukujen arvoalue on väliltä –2147483648 – 2147483647. Tiedon avulla päästään tekemään testattavan sovelluksen käyttämälle lukualueen niin syöte kuin tulosteiden joukolle ekvivalenssiositus. Molemmat syötekentät ovat tyypiltään samoja, joten niiden ekvivalenssiluokat ovat keskenään yhteneväisiä, kuten myös tulosarvojen joukko edellä mainittujen kanssa. Lukuavaruudelle suoritetulla osituksella saadaan aikaiseksi kolme ekvivalenssiluokkaa:

{*kokonaisluku* x | x < -2147483648} eli x < arvoalueen minimi

{*kokonaisluku* x | -2147483648 ≤ x ≤ 2147483647} eli x = validi arvo

{*kokonaisluku* x | x > 2147483647} eli x > arvoalueen maksimi.

Ekvivalenssiositusta tehtäessä on jako luokkiin tehtävä myös, jos on syytä olettaa, että luokan edustajia ei käsitellä samalla tavoin. Lukuarvojen muuttuessa negatiivisista positiivisiksi kohdassa nolla, voidaan epäillä lukuja käsiteltävän eri tavoin rajan eri puolilla. Todellisuudessa lukujen käsittelyllä ei ole itse sovelluksessa eroa, ovatko luvut negatiivisia vai positiivisia, mutta mustalaatikkotestausmenetelmällä testausta suorittavalla testaajalla ei ole tietoa sovelluksen sisäisestä rakenteesta perustuvat testitapausten valinnat oletuksiin ohjelman toiminnasta. Arvojen käsittely voi erota myös ohjelman suorituksen alemmilla tasoilla, esimerkiksi käännösvaiheessa, mistä ei edes sovelluksen kehittäjällä ole välttämättä tietoa, joten arvoalueen jako kohdassa nolla on menetelmän käytön kannalta perusteltua. Saadaan uusi ekvivalenssiluokittelu:

{*kokonaisluku* x | x < -2147483648}

{*kokonaisluku* x | -2147483648 ≤ x ≤ -1}

{*kokonaisluku* x | x = 0}

{*kokonaisluku* x | 1 ≤ x ≤ 2147483647}

{*kokonaisluku* x | x > 2147483647}.

Syötteiksi saaduilta arvo-alueilta valitaan kultakin alueelta sattumanvaraisesti yksi arvo (kts. Taulukko 11).

Taulukko 11. Jokaisesta ekvivalenssiluokasta on valittu satunnaisesti arvo testattavaksi.

arvoalue	testattava arvo	validi
X < -2147483648	-3000000000	ei
-2147483648 ≤ x ≤ -1	-10000	kyllä
X = 0	0	kyllä
1 ≤ x ≤ 2147483647	10000	kyllä
X > 2147483647	3000000000	ei

Haluttaessa varmentaa sovelluksen toiminnan oikeellisuus myös jaettujen ekvivalenssiluokkien rajoilla otetaan käyttöön raja-arvoanalyysi eli testi suoritetaan vielä arvoille kunkin ekvivalenssiluokan minimi- ja maksimiarvojen molemmin puolin. Saatavat testattavat arvot raja-arvoanalyysin jälkeen löytyvät taulukosta 12. Arvo alueiden raja-arvot menevät päällekkäin testattaessa ekvivalenssiluokkien maksimi ja minimi arvoja ja niiden viereisiä arvoja, joten niiden testaamiseen riittää monessa tapauksessa yksi ja sama arvo.

Taulukko 12. Kertolaskin–sovellukselle annettavat testisyötteet ekvivalenssiosituksen ja raja-arvoanalyysin jälkeen selityksineen.

testattava arvo	selite	arvoalue	validi
-3000000000	satunnainen arvo ekvivalenssiluokasta	X < -2147483648	ei
-2147483650	arvoalueen alittavien arvojen max – 1	X < -2147483648	ei
-2147483649	arvoalueen alittavien arvojen max, negatiivisen arvoalueen min – 1	X < -2147483648 -2147483648 ≤ x ≤ -1	ei
-2147483648	arvoalueen alittavien arvojen max +1, negatiivisen arvoalueen min	X < -2147483648 -2147483648 ≤ x ≤ -1	kyllä
-2147483647	negatiivisen arvoalueen min + 1	-2147483648 ≤ x ≤ -1	kyllä
-10000	satunnainen arvo ekvivalenssiluokasta	-2147483648 ≤ x ≤ -1	kyllä
-2	negatiivisen arvoalueen max – 1	-2147483648 ≤ x ≤ -1	kyllä
-1	negatiivisen arvoalueen max, 0 - arvoalueen min – 1 ja max – 1	-2147483648 ≤ x ≤ -1 x = 0	kyllä
0	negatiivisen arvoalueen max + 1, 0 - arvoalueen min ja max positiivisen arvoalueen min –1	-2147483648 ≤ x ≤ -1 x = 0 1 ≤ x ≤ 2147483647	kyllä
1	0 - arvoalueen min + 1 ja max + 1, positiivisen arvoalueen min	x = 0 1 ≤ x ≤ 2147483647	kyllä
2	positiivisen arvoalueen min + 1	1 ≤ x ≤ 2147483647	kyllä
10000	satunnainen arvo ekvivalenssiluokasta	1 ≤ x ≤ 2147483647	kyllä
2147483646	positiivisen arvoalueen max – 1	1 ≤ x ≤ 2147483647	kyllä
2147483647	positiivisen arvoalueen max arvoalueen ylittävien arvojen min –1	1 ≤ x ≤ 2147483647 x > 2147483647	kyllä
2147483648	positiivisen arvoalueen max + 1 arvoalueen ylittävien arvojen min	1 ≤ x ≤ 2147483647 x > 2147483647	ei
2147483649	arvoalueen ylittävien arvojen min + 1	x > 2147483647	ei
3000000000	satunnainen arvo ekvivalenssiluokasta	x > 2147483647	ei

Taulukkoon 12 merkityillä arvoilla *ekvivalenssiluokkien kattavuus* eli testissä käytettyjen ekvivalenssiluokkien määrä muodostuu täydeksi eli kattavuus on:

$C_{ep} = P_c / P_t * 100\% = 100\%$, missä

P_c testin kattamien luokkien määrä: 4,

P_t kaikkien luokkien kokonaismäärä: 4

eli testin syötearvot kattavat kaikki sovelluksen syötearvoavaruuden ekvivalenssiluokat.

Testattavan sovelluksen asettamien rajoitusten, eli lukuarvoalueen ylittävien ja alittavien syötteiden syöttäminen on tehty mahdottomaksi em. sovelluskehittimen tyyppitarkastuksissa, mikä ehkäisee sovelluskehittäjän puolesta virhetilanteiden syntymistä estäen ylivuotoja muistinalueella. Näin ollen joudutaan jättämään arvoalueen minimin ja maksimin ylittävät ja alittavat syötteet (kts. Taulukko 12, ei validit arvot) testaamatta. Testaaja voi joka tapauksessa päästä testaamaan sovelluksen toimintaa arvoalueen ylittävillä arvoilla sovelluksen tuottamilla tulosarvoilla, joiden arvoalue on jaettavissa samoihin ekvivalenssiluokkiin kuin syötearvojen joukko. Yleisesti ekvivalenssiositusta käytetään vain syötearvojen jakamiseen erillisiin luokkiin, koska sovelluksen toimintaa päästään manipuloimaan vain syötteiden kautta. Menetelmää voidaan sen määritelmän mukaan soveltaa myös tulosarvoavaruuden jakamiseen *ekvivalenssiluokkiin* edellä kuvatun luokkajaon mukaisesti [BSI98a]. Tällöin havaitaan, että esimerkiksi laskinsovelluksella kerrottaessa kaksi täysin validia syötettä keskenään (esimerkiksi syöte 1: arvoalueen maksimi ja syöte 2: 2) lukujen muodostama tulo ylittää reilusti käytetyn arvoalueen ylärajan. Joten testiin lisätään myös syötteet, joiden tulo ylittää ja alittaa kelvollisten tulosteiden arvoalueen *ekvivalenssiluokkien* rajat. Sovellukselle käyttöliittymän kautta syötettävissä olevat, ekvivalenssiosituksella ja rajaarvoanalyysiä käyttäen saadut ja siten myös testattavat arvot ja niille saatavat tulokset on listattu taulukkoon 13.

Taulukko 13. Kertolaskin–sovellukselle annettavat testisyötteet ekvivalenssiosituksen ja raja-arvoanalyysin jälkeen, joukosta poistettu myös arvot, joita sovellus ei anna syöttää.

Luku1	Luku2	Tulo(Luku1,Luku2)
0	0	0
0	-1	0
1	-1	-1
1	1	1
0	-2	0
1	-2	-2
0	2	0
1	2	2
1	10000	10000
1	-10000	-10000
1	2147483646	2147483646
1	2147483647	2147483647
2	2147483647	4294967294
1	-2147483647	-2147483647
1	-2147483648	-2147483648
2	-2147483648	-4294967296

Taulukkoon 13 merkityillä arvoilla syötearvojen *ekvivalenssiluokkien kattavuus* muodostuu:

$C_{ep} = P_c / P_t * 100\% = 50\%$, missä

P_c testin kattamien syöteavaruuden luokkien määrä: 2,

P_t kaikkien syöteavaruuden luokkien kokonaismäärä: 4

eli testin syötearvot kattavat vain puolet sovelluksen syötearvoavaruuden ekvivalenssiluokista.

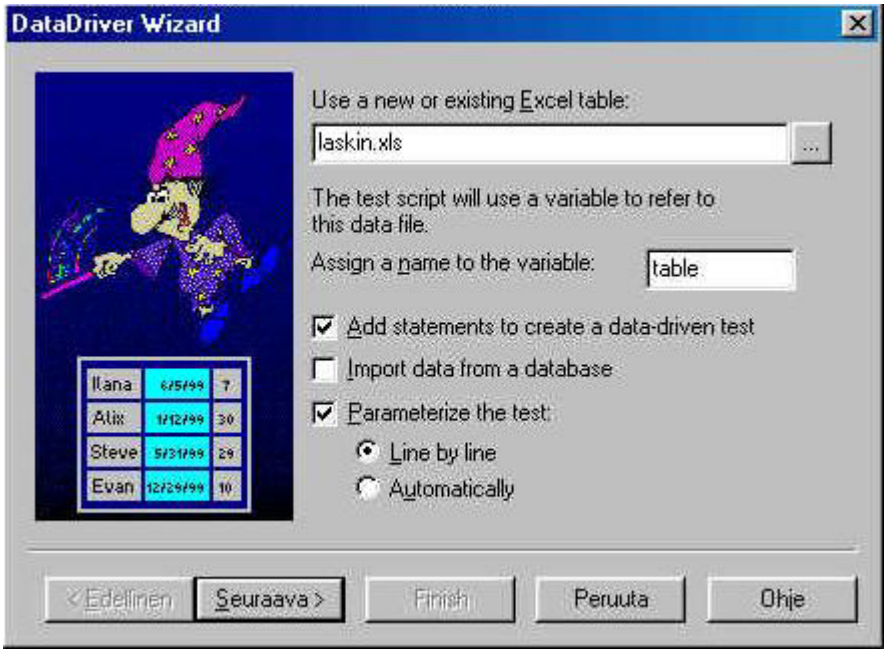
Laskettaessa kattavuutta tulosarvojen ekvivalenssiluokille muodostuu kattavuus täydeksi valitsemalla mukaan syötteet, joiden muodostama tulo ylittää ja alittaa kelvollisten tulosteiden arvoalueen *ekvivalenssiluokkien* rajat:

$C_{ep} = P_c / P_t * 100\% = 100\%$, missä

P_c testin kattamien tulosteavaruuden luokkien määrä: 4,

P_t kaikkien tulosteavaruuden luokkien kokonaismäärä: 4.

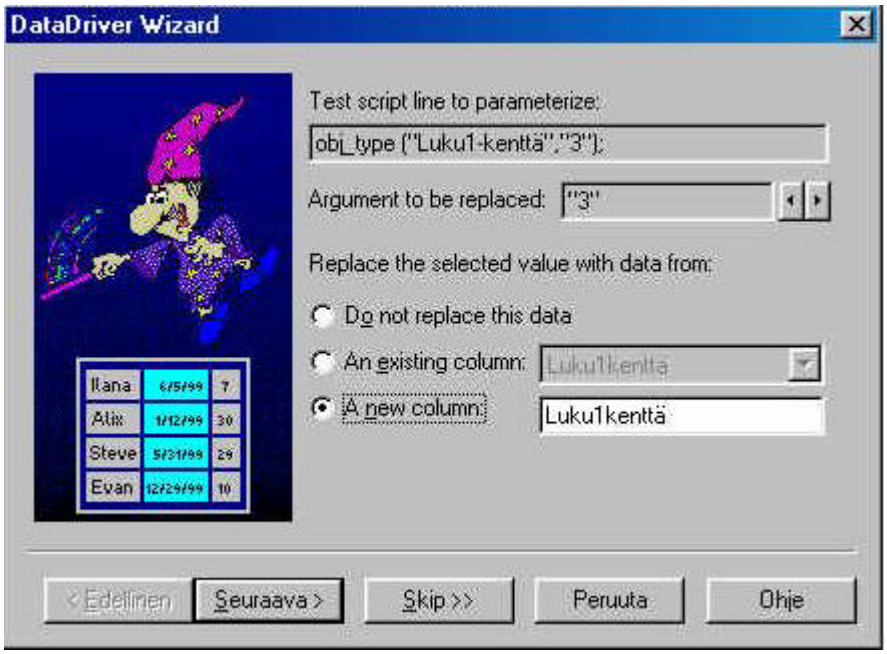
Seuraavana vaiheena skriptin toteutuksessa on luoda taulu, joka sisältää edellä kuvatun kaltaiset testattavat syötteet ja niiden halutut kombinaatiot WinRunnerissa olevalla *Data-Driver Wizard* –työkalulla (kts. Kuva 29), joka myös lisää suurimman osan taulun käsittelyyn tarvittavasta koodista skriptiin. Esimerkin testiskripitissä työkalulla luodaan Excel-taulu laskin.xls ja muuttuja nimeltään *table*, jonka kautta kyseistä taulua käsitellään testiskriptissä.



Kuva 29. WinRunnerin *DataDriver Wizard* –työkalu, jolla saadaan määrättyä syötettävät tiedot sisältävä taulu.

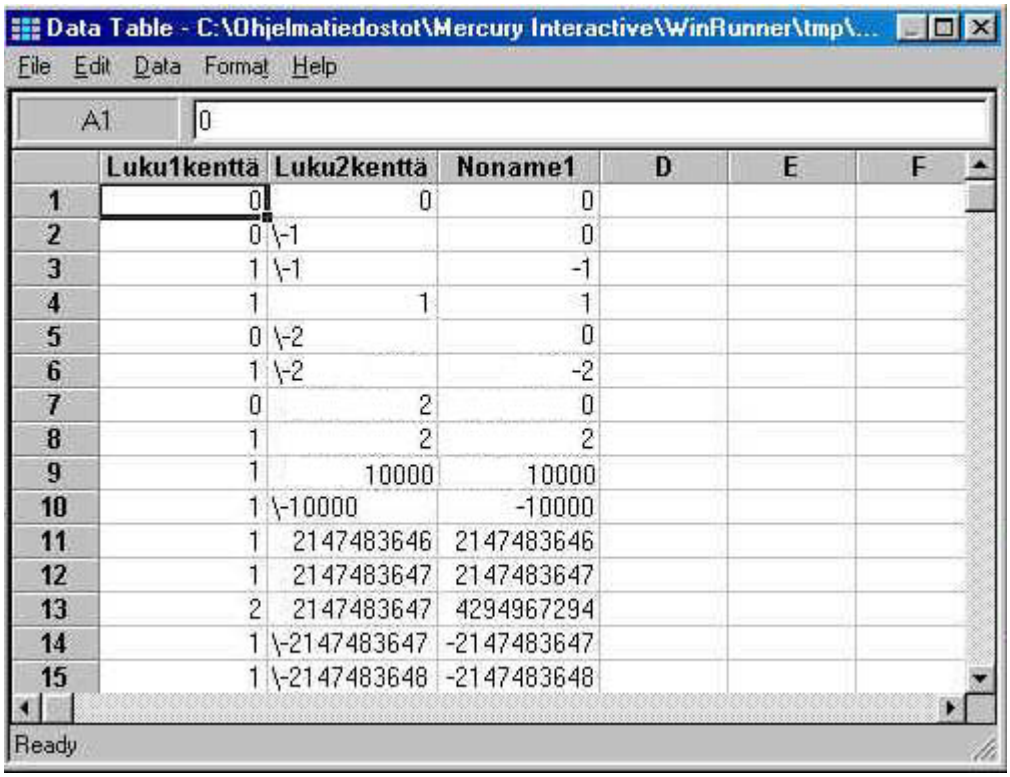
Seuraavaksi määritellään parametrisoitavat tiedot. Editoitavassa testiskriptissä korvataan Luku1-kenttään ja Luku2-kenttään sijoitettavat kiinteät lukuarvot muuttujilla. Muuttujiin tullaan sijoittamaan testin suorituksen aikana sovelluksella testattavat arvot. Muuttujiin

sijoitettavat arvot puolestaan tallennetaan testikantaan, joka on edellä luotu Excel-taulu. Tauluun tehdään syötteille omat sarakkeensa (Luku1Kenttä/Luku2Kenttä (kts. Kuva 30).



Kuva 30. *DataDriver Wizard* –työkalulla määritellään parametrisoitavat syötearvot.

Lopuksi lisätään testissä syötettävät arvot tauluun (kts. Kuva 31) ja myös testisyötteillä saatavien tuloksten vertailuarvot sijoitetaan tauluun omaan sarakkeeseensa (*Noname 1*) ja laitetaan koko syöttötoiminto silmukkaan, joka suoritetaan niin useasti kuin taulussa riittää arvoja syötettäväksi.



	Luku1kenttä	Luku2kenttä	Noname1	D	E	F
1	0	0	0			
2	0	\-1	0			
3	1	\-1	-1			
4	1	1	1			
5	0	\-2	0			
6	1	\-2	-2			
7	0	2	0			
8	1	2	2			
9	1	10000	10000			
10	1	\-10000	-10000			
11	1	2147483646	2147483646			
12	1	2147483647	2147483647			
13	2	2147483647	4294967294			
14	1	\-2147483647	-2147483647			
15	1	\-2147483648	-2147483648			

Kuva 31. Excel-taulu, jossa parametrisoidut syötearvot ja saaduille vasteille vertailuarvot ts. oikeat vastaukset.

editoidussa skriptissä (kts Kuva 28):

luodun taulun avaus:

```
table = "laskin.xls";
rc = ddt_open(table, DDT_MODE_READ);
if (rc!= E_OK && rc != E_FILE_OPEN)
       pause("Cannot open table.");
```

haettujen arvojen syöttö kenttiin:

```
obj_type ("Luku1-kenttä",ddt_val(table,"Luku1kenttä"));
…
obj_type ("Luku2-kenttä",ddt_val(table,"Luku2kenttä"));
```

tietokannan sulkeminen:

```
ddt_close(table);
```

taulusta syötteitä ja vertailutuloksia lukeva silmukka:

```
ddt_get_row_count(table,table_RowCount);
for(table_Row = 1; table_Row <= table_RowCount; table_Row ++)
{
…
}
```

Tämän jälkeen lisätään sovelluksen laskeman tuloksen kaappaus Tulos-kentästä. editoidussa skriptissä (kts Kuva 28):

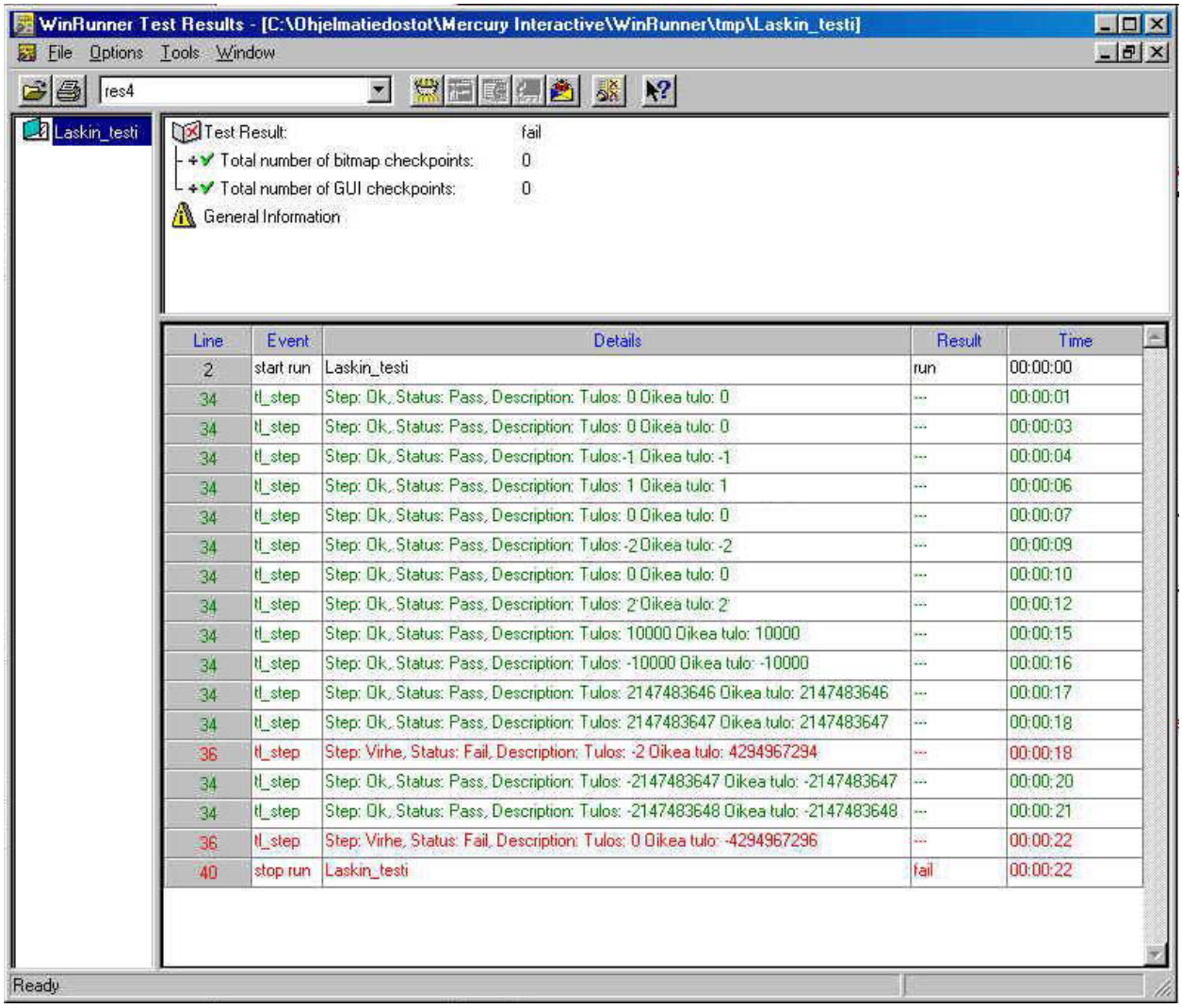
```
obj_get_text("Tulos-kenttä", text);
```

Lopuksi suoritetaan vertailu Tulos kentästä saadulle arvolle (text) ja tietokannasta haetulle oikealle vastaukselle (ddt_val(table, "Noname1")) ja raportoidaan menikö laskutoimitus testattavalla sovelluksella oikein.
editoidussa skriptissä (kts Kuva 28):

```
if (text == ddt_val(table, "Noname1")){
       tl_step("Ok",0,"Tulos: " & text & " Oikea tulo: " &
       ddt_val(table, "Noname1"));}
else{
       tl_step("Virhe",1,"Tulos: " & text " Oikea tulo: " &
       ddt_val(table, "Noname1"));}
```

9.4.5 Editoidun skriptin ajo ja tulosten tarkastelu

Editoinnin jälkeen skripti on ajettavissa eli on vuorossa testausprosessin neljäs vaihe *testin ajo*. Testin ajaminen onnistuu, jos skriptiin lisätyt koodit ovat skriptikielen syntaksin mukaisia, muussa tapauksessa skriptin ajo pysähtyy virheelliselle skriptiriville ja skripti ilmoittaa virhetilanteesta virheilmoituksella. Tässä vaiheessa voidaan joutua palaamaan testausprosessissa takaisin debuggaus vaiheeseen. Onnistuneen testiskriptin ajon jälkeen on vuorossa testausprosesin seuraava vaihe: *tulosten tarkastelu.* WinRunner avaa käyttäjän tarkasteltavaksi raportointi-ikkunan (Test Results), joka sisältää testaajan (skriptissä tl_step –lauseet) ja myös työkalun itsensä tuottamat (virhe)ilmoitukset testin etenemisestä (kts. Kuva 32), jonka jälkeen päädytään prosessin viimeiseen vaiheeseen virheiden raportointiin. Jää käyttäjän päätettäväksi kuinka suhtautua saatuihin tuloksiin ja mitä jatkotoimenpiteitä ne vaativat. Testin ajon tuottamat virheraportit voidaan lähettää eteenpäin esimerkiksi yrityksen laatujärjestelmään tai laatutietokantaan.



Kuva 32. WinRunnerin tuottamat testitulokset. Tuloksissa huomataan että sovellus ei laske oikein laskuja 2 * 2147483647= 4294967294, vaan sovellus saa tulokseksi -2 ja 2 * -2147483648= -4294967296, josta sovellus saa tulokseksi 0 (näytöllä rivin väri punainen). Virhe johtuu edellä kuvatusta 32-bittisten lukujen arvoalueen ylityksestä. Raportin lopussa WinRunner toteaa sovelluksen toimineen väärin testissä, joten Laskin-sovellus ei läpäise testiä (näytöllä rivin väri punainen ja *Result* arvo: *fail*). Muissa tapauksissa eli eri osatesteissä sovelluksen kertolaskutoiminto on toiminut oikein (näytöllä rivin väri vihreä).

9.5 Testien ylläpito

Edellä kuvatuilla toimenpiteillä on saatu toteutettua testityökalulle ajettava testi, joka testaa Kertolaskin-sovelluksen kertolasku toimintoa. Testi on itsessään skriptikomponentti, jonka toteutuksessa uudelleenkäytettävyys ja ylläpidettävyys on otettu huomioon. Tällä periaatteella toteutetulla komponentilla kyetään testaamaan saman toiminnallisuuden omaavia sovelluksia pienillä muutoksilla. Sovelluksen käyttöliittymän muuttuessa tai vaihtuessa

kokonaan toiseksi, mutta toiminnallisuuden säilyessä samana tai edes samankaltaisena kyetään samaa komponenttia käyttämällä toteuttamaan sama testi uudelle versiolle sovelluksesta tai kokonaan uudelle sovellukselle, ainoastaan vaihtamalla käytettävää ohjelman graafisten komponenttien kuvaustiedostoa. Eli yksinkertaisesti ladataan suorituksen alussa eri GUI Map –tiedosto, johon on talletettu testin suorittamiseen tarvittavien komponenttien kuvaukset oikeassa muodossa, jotta skripti kykenee käyttämään niitä.

Sama skripti on myös uudelleenkäytettävissä testattavan sovelluksen toiminnallisuuden muuttuessa, esimerkiksi edellä havaitun kertolaskufunktion toiminnan virheellisyyden voi korjata siirtymällä käyttämään syötearvojen ja tulostearvojen kohdalla 32-bittisten lukujen sijasta 64-bittisiä lukuja, jolloin virhe raja-alueelta siirtyy 64-bittisten kokonaislukujen lukuavaruuden rajoille. Mustalaatikkomenetelmän mukaisessa testaamisessa testaajalla toisaalta ei tulisi olla edes tietoa ohjelman toteutuksesta, eikä siten myöskään lukuavaruuden muutoksesta, mikä vaikuttaa muutosten testaukseen. Edellä kuvatun kaltaisessa tapauksessa toimittaessa testauksen lasilaatikkoperiaatteen mukaan on testattava funktion toimintaa sen muutoksia vastaavilla arvoilla, joten suoritettaessa Data Driven – tyyppistä testiä on testissä käytettävät syötteet korvattava paremmin uutta tilannetta vastaavilla arvoilla. Uudet käytettävät lukuarvot saadaan testattua samalla skriptillä muuttamalla ainoastaan käytettävää syötetietokantaa, jolloin vastaavan laajuinen toiminnallisuuden testaaminen kyetään suorittamaan korjatulle sovellukselle käyttöliittymän pysyessä samana.

Luodusta testiskriptistä saadaan perinteisten ohjelmointimenettelyiden tapaan luotua oma itsenäinen ja uudelleenkäytettävä komponentti yksinkertaisesti tekemällä skriptikoodista oma toiminnallinen kokonaisuus eli esimerkiksi funktio, jota kyetään kutsumaan muualta skriptikoodista tavallisella funktiokutsulla. Skriptikomponentin muunneltavuus edellä kuvatun kaltaisiin tilanteisiin, eli sovelluksen käyttöliittymän muutoksiin ja toiminnallisuuden muutoksiin, tapahtuu parametrisoimalla muuttuva data skriptissä eli koodin ulkopuoleiset tietolähteet: käyttöliittymän kuvaustiedosto ja syötetietokantataulu. Tarvittavat skriptin kannalta muuttuvat tiedot välitetään skriptikomponetille sen aliohjelmakutsussa parametreina, jolloin samaa skriptikomponenttia kyetään käyttämään kaikkien samantyyppisten sovellusten testauksessa.

9.6 Testattavuuden kehittäminen testityökaluille

Testattavuuden parantaminen testityökaluille tarkoittaa käytännössä testattavan sovelluksen kehittämistä suuntaan, jossa testattavan sovelluksen tuottamat tulosteet testisyötteille ovat testiyökalun havaittavissa ja mahdollisimman helposti verifioitavissa. Testattavuus määritellään testattavan sovelluksen kykynä paljastaa virheet itsessään. Lähdettäessä parantamaan ohjelmiston testattavuutta testityökalujen kannalta on otettava käytettävä testaustyöväline huomioon jo ohjelmiston suunnitteluvaiheessa.

Riippuen eri testityökaluista ja niiden ominaisuuksista on valittava, millä tavalla halutut toiminnot kyetään testaamaan, jos ollenkaan. Joten jo ohjelmiston suunnitteluvaiheessa tulisi valita kannattaako toteutettavan ohjelmiston testauksessa käyttää jotakin testityökalua. Päätöksen ollessa myönteinen automatisoidun testauksen toteuttamisessa ja käytöstä tietyssä vaiheessa ohjelmistotuotantoprosessia on valittava, mikä testausväline sopii parhaiten käyttöön ja miten halutut testattavat ominaisuudet saadaan parhaiten testityökalun tarkasteltavaksi. Ohjelmistoa ei kannata kuitenkaan ryhtyä toteuttamaan ainoastaan testauksen ehdoilla. Jos joudutaan tekemään suuria muutoksia aiottuun toteutustapaan testityökalun takia, kannattaa kyseenalaistaa testityökalun soveltuvuus aiottuun tehtäväänsä.
Yleisinä suuntaviivoina testattavuuden kehittämisestä graafisen käyttöliittymän kautta sovellusta testaaville työkaluille voidaan esittää muutamia perusperiaatteita sovelluksen kehitysvaiheisiin.

1) Sovelluksen käyttöliittymä tulisi erottaa sovelluksen varsinaisesta ohjelmakoodista. Tällöin muutokset ohjelman lähdekoodiin eivät vaikuta sovelluksen graafisen käyttöliittymän kuvaukseen, vaan ainoastaan sovelluksen toiminnallisuuteen.
2) Käyttäessään valmiita sovelluskehitysympäristöjä ohjelmistotalojen ei kannata kehittää omatekoisia komponentteja, jos kehitysympäristö tarjoaa vastaavia valmiina. Standardikäyttöliittymäkomponenttien käyttö takaa useimmissa tapauksissa testityökalun oikean toiminnan komponentin kanssa, koska niille löytyy useimmiten valmiina työkalun valmistajan tuki tai se on suurella varmuudella saatavissa tai tulossa työkalulle.

3) Kaikki käytettävät samanlaisen toiminnallisuuden omaavat komponentit tulisi toteuttaa samalla koodilla tai ainakin samalla toteutusperiaatteella, jolloin niiden testattavuus helpottuu testiskriptien uudelleenkäytön myötä.
4) Sovellusten tuottamien tulosteiden ulos saaminen ja niiden välittäminen käytettävälle testityökalulle tulee tehdä mahdollisimman helpoksi, esimerkiksi valitsemalla käyttöön sellaiset käyttöliittymäkomponentit, joiden kanssa työkalu toimii oikein ja luotettavasti.

10. YHTEENVETO

Ohjelmiston testaus on peruslähtökohdiltaan suhteellisen pessimistinen osa ohjelmistotuotantoa: kaikkia virheitä ei voida löytää kuin erittäin triviaaleissa tapauksissa ja testauksessa on tyydyttävä ohjelman toiminnan varmentamiseen tietylle tasolle. Ongelman syynä on ehkä eniten se, että testaus on usein erillinen prosessi ohjelmistotuotannossa. Ohjelma tai sen osa testataan vasta kun se on toteutettu. Eräs ratkaisuna ongelmaan on testauksen liittäminen oleelliseksi osaksi ohjelman toteuttamista, kuten joissakin uudemmissa ohjelmistontuotantomalleissa jo tehdään.

Ohjelmistojen testauksen ja sen menetelmien kehittämisessä on jatkuvasti tarkoitus parantaa onnistumisprosenttia. Etenevällä tutkimuksella saadaan lisätietoa ja yhä uusia esimerkkitapauksia tyypillisimmistä virheistä ja niiden sijainnista ohjelmistossa, jolloin vikojen etsiminen ja ennustettavuus helpottuu. Esteinä tälle kehitykselle on ohjelmistojen ja niiden tuotantovälineiden ja ympäristöjen jatkuva ja kiihtyvä kasvaminen ja edellä mainituista tekijöistä muodostuvat lukemattomat kombinaatiot. Virhemahdollisuuksien kasvaessa ovat ohjelmistotalot pakotettuja parantamaan laadunvalvontaansa, jonka perustana on ohjelmistojen perusteellinen testaus. Sovelluskehitysprosessiin on otettu mukaan ohjelmistojen nopean ja tehokkaan kehittämisen lisäksi myös laatunäkökulmat ja panostettu sovellusten testattavuuteen. Mitä aikaisemmassa vaiheessa virhe paljastuu, sitä edullisemmaksi se ohjelmistonkehitysprosessin käytettävien resurssien kannalta tulee. Virheiltä ei voida välttyä, mutta niiden tekemistä voidaan vaikeuttaa ja paljastumista helpottaa.

LÄHTEET

[BSI98a] British Standard: Software testing – Part 1: Vocabulary, BS 7925-1:1998. BSI Publications, 1998.

[BSI98b] British Standard: Software testing – Part 2: Software component testing, BS 7925-2:1998. BSI Publications, 1998.

[Faq01] Internet FAQ Archives. Internet FAQ Consortium 2001 - 2002. Viitattu 1.10.2002, saatavilla www-muodossa URL: http://www.faqs.org/faqs/software-eng/testing-faq/.

[Few99] Fewster, Mark - Graham, Dorothy: Software Test Automation: Effective use of test execution tools. Addison-Wesley, ACM press, New York, 1999.

[Fri95] Friedman, Michael A. - Voas, Jeffrey M.: Software Assessment: Reliability, Safety, Testability. John Wiley & Sons, Inc, 1995.

[Hai96] Haikala, Ilkka - Märijärvi, Jukka: Ohjelmistotuotanto. Gummerus Kirjapaino Oy, 1996.

[IEE00] IEEE Software: Strengthening the Case of Pair Programming. IEEE, 2000. Viitattu 6.10.2002, saatavilla www-muodossa URL: http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF.

[Kan01] Kaner, Cem – Bach, James – Pettichord, Bret: Lessons learned in software testing. John Wiley & Sons, Inc., 2001.

[Mel96] Meltzer, Ingo: Testing of a Computer Program on the Example of a Medical Application with Diversification and other Methods. University of Ulm Faculty of Computer Science Department of Applied Information Processing, 1996. Viitattu 1.10.2002, saatavilla www-muodossa URL: http://www.mathematik.uni-ulm.de/~melzer/thesis/node21.html.

[Mye79] Myers, Glenford J.: The Art of Software Testing. John Wiley & Sons, Inc., 1979.

[STR91] STRÍ TS2: Modelling a Software Quality Handbook (MSQH). Icelandic Council for Standardisation (STRÍ), 1991.

[Vit00] Vitharna, Padmal – Jain, Hemant: Research issues in testing business componets. Information & Management 37. Elsevier Science B.V., 2000.

[Win99] WinRunner User's Guide, version 6.0. Mercury Interactive Corporation, 1999.