REGRESSIOTESTAUS JA

TESTIEN VALINTATEKNIIKAT

Juha Holopainen
Pro gradu -tutkielma
Tietojenkäsittelytiede
Kuopion yliopiston
tietojenkäsittelytieteen laitos
Huhtikuu 2005

KUOPION YLIOPISTO, informaatioteknologian ja kauppatieteiden tiedekunta
Tietojenkäsittelytieteen koulutusohjelma

HOLOPAINEN JUHA, M.: Regressiotestaus ja testien valintatekniikat
Pro gradu -tutkielma, 82 s., 3 liitettä (8 s.)
Pro gradu -tutkielman ohjaajat: FT Anne Eerola ja FM Tanja Toroi
Huhtikuu 2005

Avainsanat: testaus, regressiotestaus, regressiotestauksen tehostaminen, valintatekniikat, priorisointitekniikat

Regressiotestauksella tarkoitetaan ohjelman uudelleentestaamista muutoksen jälkeen. Regressiotestauksen tavoitteena on paljastaa muutoksen mahdollisesti aiheuttamat virheet ohjelmassa ennen muutosta olleeseen toiminnallisuuteen. Tämä tehdään testaamalla ohjelmaa regressiotestitapauksilla, joista osa on peräisin ohjelman vanhasta testijoukosta ja osa on uusia, jo aiemmin testatun toiminnallisuuden testaamiseen tarkoitettuja regressiotestitapauksia. Uuden toiminnallisuuden testaamista kutsutaan progressiotestaukseksi. Progressiotestausta ei tarvita, jos ohjelmaan ei ole tehty uutta toiminnallisuutta.

Tässä tutkielmassa määritellään regressiotestaus sekä kerrotaan regressiotestausprosessin eri vaiheissa suoritettavista toimenpiteistä. Lisäksi tutkielmassa keskitytään kertomaan regressiotestauksen tehostamisesta.

Tällä hetkellä yleisin tapa suorittaa regressiotestaus on käyttää uudelleen kaikki vanhat testitapaukset. Tehokkain ja eniten tutkittu tapa tehostaa regressiotestausta on vähentää regressiotestauksessa ajettavien vanhojen testitapausten määrää käyttämällä regressiotestien valintatekniikoita. Muita kirjallisuudessa tutkittuja tapoja regressiotestauksen tehostamiseksi ovat priorisointi- ja harventamistekniikoiden käyttö. Regressiotestauksen tehokkaassa suorittamisessa on tärkeää myös automatisointi, mutta tässä tutkielmassa automatisointi ohitetaan maininnalla. Tärkein huomio kohdistetaan valintatekniikoihin.

Esipuhe

Tämä tutkielma on tehty Kuopion yliopiston tietojenkäsittelytieteen laitokselle keväällä 2005.

Haluan kiittää ohjaajiani Anne Eerolaa ja Tanja Toroita asiantuntevasta ohjauksesta ja erinomaisista neuvoista.

Kuopiossa 29.4.2005

_____

Juha Holopainen

SISÄLLYS

1 JOHDANTO

Regressiotestaus on tärkeä osa ohjelmistontuotantoprosessia, sillä ohjelmoijat korjaavat virheen usein väärin. Ohjelmaan tehtyjen muutosten ja korjausten on huomattu olevan alttiimpia virheiden löytymiselle kuin ohjelman alkuperäinen koodi (Myers, 1979). Cem Kanerin mukaan joissakin isommissa järjestelmissä on havaittu virheen korjauksen epäonnistuvan jopa 80 prosentissa tapauksista. Hänen mukaansa tilannetta voidaan pitää erittäin hyvänä, jos korjaus epäonnistuu vain joka kymmenes kerta (Kaner ym., 1999, s. 93–94). Jonesin tekemien tutkimusten mukaan 1990-luvulla virheellisiä korjauksia esiintyi keskimäärin noin seitsemässä prosentissa ohjelmistoon tehdyistä korjauksista (Jones, 1997).

Regressiotestaus on terminä monille epäselvä ja sen käytännön suorituksesta on olemassa erilaisia tulkintoja. Tämän tutkielman ensimmäisessä kolmanneksessa selvennetään regressiotestaus-termiä ja esitetään tulkinta regressiotestausprosessin eri vaiheista ja niissä tehtävistä toimenpiteistä. Loppuosa tutkielmasta käsittelee regressiotestauksen tehostamista.

Luvussa 2 kerrotaan regressiotestaukseen liittyvistä käsitteistä ja pyritään selventämään, miten regressiotestaus eroaa progressiotestauksesta. Luvussa 3 kerrotaan testausprosessin eri vaiheista ja asiaan liittyvästä muutosvaikutusanalyysistä.

Luvussa 4 kerrotaan regressiotestausta tehostavista metodologioista. Tässä tutkielmassa suurin huomio kiinnitetään regressiotestien valintatekniikoihin, mutta myös muita metodologioita käsitellään lyhyesti. Koska metodologioiden tehokas käyttö edellyttää niihin vaikuttavien tekijöiden tuntemista, kerrotaan aliluvussa 4.3 eri metodologioihin vaikuttavista asioista. UML-määrityksiin pohjautuvista valintatekniikoista kerrotaan luvussa 5.

Luvussa 6 kerrotaan, kuinka komponenttipohjaisten ohjelmien regressiotestausta voidaan parantaa komponentteihin liitettävän ylimääräisen tiedon avulla. Tutkielmassa esitetään kaksi eri tapaa tällaisen tiedon jäsentämiseksi.

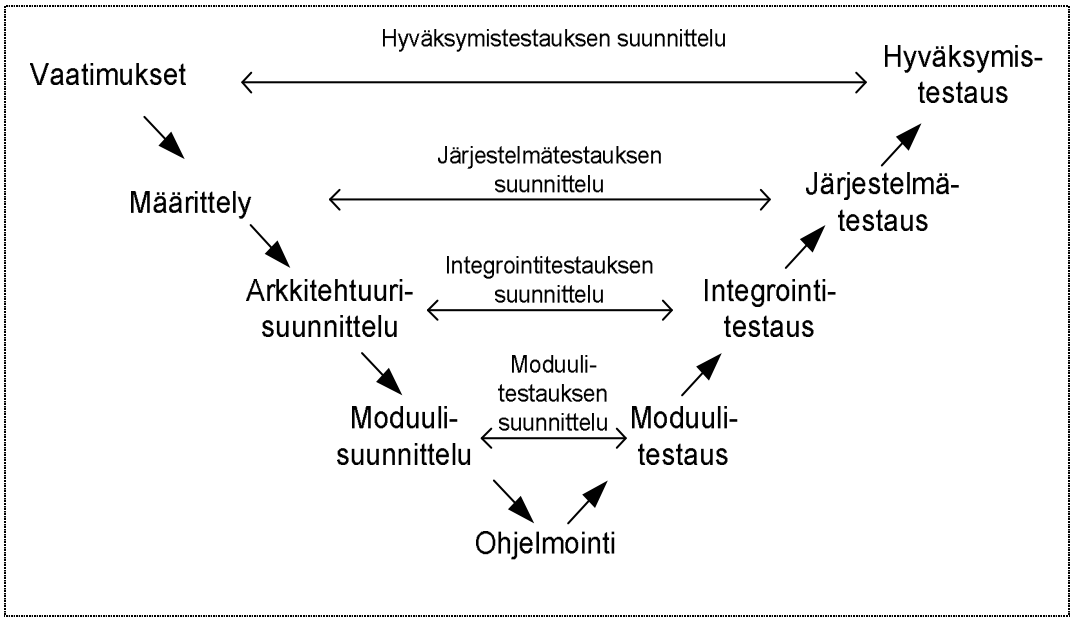
Tässä tutkielmassa ei käsitellä tarkemmin olio-ohjelmien regressiotestausta, mutta asiasta voi lukea Holopaisen selvityksestä (Holopainen, 2004).

2 REGRESSIOTESTAUKSEN KÄSITTEET

Muutetun ohjelman testausprosessi koostuu kahdesta erityyppisestä testauksesta: *regressiotestauksesta* ja *progressiotestauksesta*. Luvussa 2.1 kerrotaan testauksesta yleensä. Luvussa 2.2 kerrotaan tarkemmin regressiotestauksesta ja luvussa 2.3 progressiotestauksesta. Luvussa 2.4 esitellään, missä tilanteissa regressiotestausta voi ja kannattaa käyttää.

2.1 Testauksen määritelmä

Ohjelman *testauksella* tarkoitetaan ohjelman suorittamista tietyillä syötteillä pyrkimyksenä paljastaa ohjelman sisältämiä virheitä. Perimmäinen idea on vahvistaa luottamusta koodin oikeellisuuteen löytämällä mahdollisimman monta ohjelmassa olevaa virhettä ja poistamalla ne. Kaikkia vähänkään suuremman ohjelman sisältämiä virheitä ei voida milloinkaan löytää. Usein testaus suoritetaan käyttämällä V-mallia (kuva 1), jossa testaus on integroitu osaksi kehitysprosessia.



Kuva 1. Testauksen V-malli (Paakki, 2000).

Ohjelman testaamiseksi sille luodaan *testitapauksia* (test case), joiden tiedot kirjataan testitapausten määrittelydokumentteihin. Testitapaus sisältää sarjan ohjelmalle annettavia *syötteitä*, ohjelman *odotetun tuloksen* annetuilla syötteillä, *esiehdot* testin suorittamiseksi sekä *jälkiehdot* ohjelman tilan oikeellisuuden varmentamiseksi testin jälkeen. Esiehdoissa kerrotaan, missä tilassa ohjelman ja ympäristön pitää olla testitapauksen suorittamiseksi. Jälkiehdot kertovat, missä tilassa ohjelman pitää olla suorituksen jälkeen. Näiden lisäksi testitapauksen määrittelydokumentissa selitetään lyhyesti testitapauksen testaamat ohjelman toiminnot ja osat sekä kerrotaan viitteet muihin asiaan liittyviin dokumentteihin. Jos testitapaus on riippuvainen toisista testitapauksista, ilmoitetaan mitkä testitapaukset pitää ajaa ennen kyseistä testitapausta tai muulla tavoin selvennetään suoritusjärjestys. (Paakki, 2000)

Määritellyistä testitapauksista kootaan *testijoukkoja* (test suite) ohjelman tai sen osien testaamiseksi. Testijoukko on joukko testitapauksia, jotka on kerätty yhteen tiettyä tarkoitusta varten. Ohjelmalle luotu testijoukko sisältää kaikki ne testitapaukset, joilla ohjelmaa tullaan testaamaan. Ohjelmalle luotu testijoukko sisältää osatestijoukkoja, jotka on tarkoitettu ohjelman eri osien, kuten esimerkiksi komponenttien, testaamiseen. Tässä tutkielmassa testijoukolla tarkoitetaan koko ohjelman testijoukkoa, ellei toisin mainita.

Testausprosessissa ohjelmaa testataan jokaisella ohjelmalle luodun testijoukon sisältämällä testitapauksella. Ensin ohjelman tila ja suoritusympäristö alustetaan testitapauksessa kerrottujen esiehtojen mukaiseksi. Seuraavaksi testattavalle ohjelmalle annetaan testitapauksessa määritellyt syötteet. Lopuksi ohjelman antamaa tulosta verrataan testitapauksessa määriteltyyn odotettuun tulokseen ja tarkastetaan, että jälkiehdot ovat voimassa. Mikäli saatu tulos eroaa odotetusta, on ohjelmassa virhe, olettaen että testitapaus on laadittu huolellisesti. (Harrold ym., 2001)

Kattavuudella voidaan tarkoittaa montaa eri asiaa. *Testitapauksen kattavuus* ohjelmassa voidaan määrittää etsimällä ohjelmasta ensin määrätynlaiset entiteetit ja tutkimalla, kuinka suuren osan ohjelman entiteeteistä testitapaus käy läpi, kun testitapaus ajetaan. *Entiteetillä* tarkoitetaan jotakin ohjelman osaa, kuten esimerkiksi lauseita (lausekattavuus) tai ohjelman sisältämiä suorituspolkuja (polkukattavuus). *Testijoukon kattavuudella* tarkoitetaan testijoukon sisältämien testitapausten yhteenlaskettua kattavuutta

ohjelmassa. Kattavuuden perusteella voidaan myös päättää ohjelman riittävästä testauksesta, eli millaisten kattavuusehtojen täyttyessä regressiotestaus lopetetaan.

Tässä selvityksessä mainitaan usein termi *vaarallinen entiteetti*. Termillä tarkoitetaan entiteettiä, johon ohjelmaan tehty muutos on voinut vaikuttaa, ja joka näin ollen voi käyttäytyä muutetussa ohjelmassa eri tavalla kuin alkuperäisessä ohjelmaversiossa. (Harrold ym., 2001)

2.2 Regressiotestauksen määritelmä

Ohjelman r*egressiotestauksella* tarkoitetaan aiemmin testatun ohjelman uudelleentestaamista pyrkimyksenä varmistaa, että ohjelmaan tehty muutos on oikeellinen eikä ole aiheuttanut virheitä ohjelmassa ennen muutosta olleeseen toiminnallisuuteen. Ohjelmaan lisätyn uuden toiminnallisuuden testaamisen katsotaan kuuluvan progressiotestauksen piiriin (kts. kappale 2.3). Ohjelmaan tehty muutos voi olla koodin muuttamista, lisäämistä tai poistamista. Toimintaympäristöllä tarkoitetaan ohjelmaan yhteydessä olevia ulkopuolisia ohjelmia tai järjestelmiä, kuten myös laitteistoa ja käyttöjärjestelmää, joilla ohjelmaa suoritetaan.

Ohjelmaan tehdyn muutoksen laadusta riippuen voidaan ohjelman regressiotestauksessa käyttää uudelleen vaihteleva määrä vanhoja, edellisellä testauskierroksella ohjelman onnistuneesti läpäisseitä testitapauksia. Näiden lisäksi joudutaan joskus tekemään uusia regressiotestitapauksia vanhan toiminnallisuuden oikeellisuuden tarkistamiseksi. Jos ohjelmaan tehty muutos on toteutettu pyrkimyksenä korjata ohjelmasta löydetty virhe, regressiotestataan muutettua ohjelmaa virheen löytäneillä testitapauksilla. Jos ohjelman vanhasta, aiemmin testatusta toiminnallisuudesta löytyy ohjelmaan tehdyn muutoksen jälkeisessä regressiotestauksessa virhe, sanotaan että ohjelma on *regressoitunut* eli taantunut.

Perustason versioksi (baseline version) kutsutaan komponenttia tai systeemiä, joka on läpäissyt testijoukon. Jos komponentti tai systeemi ei ole läpäissyt testijoukkoa, sitä kutsutaan *deltaversioksi* (delta version). *Deltakooste* (delta build) on kaikki perustason ja deltatason komponentit sisältävä ajettava kokonaisuus. *Regressiotestitapaukseksi* (regression test case) kutsutaan ohjelman vanhaa toiminnallisuutta testaavaa testitapaus-

ta. Regressiotestitapaus voi olla sellainen testitapaus, jonka perustason ohjelma on läpäissyt, ja jonka deltakoosteen odotetaan läpäisevän. (Binder, 1999, s.756). Edellä mainittujen uudelleenkäyttöön perustuvien regressiotestitapausten lisäksi on olemassa myös uusia regressiotestitapauksia, jotka on kehitetty muutetussa ohjelmassa ennen muutosta olleen toiminnallisuuden oikeellisuuden testaamiseen. Regressiotestauksessa paljastuneita virheitä kutsutaan *regressiovirheiksi* (regression fault).

Ohjelmaan tehtyjen muutosten seurauksena osa vanhoista testitapauksista voi olla käyttökelvottomia muutetussa ohjelmassa. Näin voi käydä varsinkin silloin, kun ohjelman määrityksiin tehdään muutoksia, eli esimerkiksi poistetaan jokin testitapauksen testaama toiminto. *Testitapauksen uudelleenkelpoistamisprosessin* (test case revalidation) tarkoituksena on löytää käyttökelvottomat testitapaukset testijoukosta ja kelpoistaa ne käytettäväksi regressiotestauksessa tai vaihtoehtoisesti poistaa ne, mikäli kelpoistaminen ei onnistu. Uudelleenkelpoistamisprosessissa tarkastellaan testitapauksen syötettä ja odotettua tulosta. Mikäli syöte on kelvoton, testitapaus voidaan poistaa testijoukosta tai käyttää jatkossa negatiivisena testitapauksena. Mikäli syöte on kelvollinen, mutta tulos on kelvoton, testitapauksen odotettu tulos pitää muuttaa vastaamaan nykyisiä olosuhteita. (Onoma ym., 1998)

Uudelleenkelpoistaminen täytyy tehdä manuaalisesti, joten se vie paljon aikaa. Uudelleenkelpoistaminen voidaan tehdä ennen regressiotestausta tai regressiotestauksen jälkeen testituloksia tarkasteltaessa. Edellä mainitussa tapauksessa uudelleenkelpoistaminen suoritetaan koko regressiotestijoukolle. Jälkimmäisessä tapauksessa tarkastellaan yleensä vain niitä testitapauksia, jotka tuottivat odotetuista tuloksista poikkeavia tuloksia, joten tällöin uudelleenkelpoistaminen on huomattavasti halvempaa kuin edellä mainitussa tilanteessa. Toisaalta tämä on myös vaarallisempaa, koska tällöin jotkin virheet voivat jäädä huomaamatta, koska testitapaukset eivät ole ajan tasalla. (Onoma ym., 1998)

2.3 Progressiotestauksen määritelmä

Ohjelmaan lisätyn uuden toiminnallisuuden testaamista kutsutaan *progressiotestaukseksi*. Progressiotestauksessa ohjelmalle luodaan *progressiotestijoukko*, joka koostuu *prog-*

ressiotestitapauksista. Progressiotestausta tarvitaan vain silloin, kun ohjelmaan on lisätty uutta toiminnallisuutta. Uuden ohjelman ensimmäinen testauskierros on kokonaisuudessaan progressiotestausta, koska kaikki ohjelman toiminnallisuus on tällöin uutta.

Progressiotestaus ja regressiotestaus on helpointa erottaa toisistaan käytettävän testijoukon perusteella. Progressiotestauksessa käytettävä testijoukko sisältää testitapauksia, jotka testaavat ohjelman uutta toiminnallisuutta, eli toiminnallisuutta, jota ei ole vielä kertaakaan testattu. Regressiotestauksessa käytettävä testijoukko sisältää testitapauksia, jotka testaavat ohjelman vanhaa toiminnallisuutta, eli toiminnallisuutta, jota on jo aiemmin testattu.

2.4 Regressiotestauksen käyttötilanteet

Sen jälkeen, kun testattava kohde on progressiotestattu, eli testattu alkuperäisellä testijoukolla, voidaan regressiotestaus suorittaa missä tahansa kehitysvaiheessa (kuva 1) sekä ohjelman ylläpitovaiheen aikana. Komponenttitasolla perustason komponenttiin tehty muutos johtaa deltaversion syntymiseen. Järjestelmätasolla regressiotestattava deltakooste syntyy, kun uusia komponentteja integroidaan osaksi perustason järjestelmää (Binder, 1999, s. 760). Regressiotestaus suoritetaan usein seuraavien prosessien aikana (Binder, 1999, s. 756):

- *Uudelleenkäyttöön pohjautuvassa ohjelmistontuotannossa*. Uudelleenkäytettävät objektit voivat olla monenlaisia, kuten esimerkiksi koodinpätkiä, luokkia tai komponentteja.
- *Nopean iteratiivisen ohjelmistontuotannon aikana* regressiotestaus voidaan suorittaa useita kertoja päivässä.
- *Integroitaessa komponentteja*. Regressiotestaus on hyvä ensiaskel integrointitestauksessa. Kertyneiden testijoukkojen ajaminen uudelleen, kun komponentteja lisätään perättäisiin testauskonfiguraatioihin, kasvattaa regressiotestijoukkoa inkrementaalisesti ja paljastaa regressiovirheitä.
- *Ohjelmiston ylläpitoprosessin aikana*. Ohjelmiston ylläpitoprosessi voi olla sisällöltään joko korjaavaa, täydentävää, sopeuttavaa tai ennakoivaa (Gao ym., 2003):

- *Korjaavan (corrective) ylläpitoprosessin* tarkoituksena on korjata löydetyt virheet. Komponenttipohjaisissa järjestelmissä korjauksia tehdään yleensä yksittäisiin komponentteihin. Rajapinnat, komponentin yleinen rakenne ja määritykset pysyvät yleensä muuttumattomina.
- *Täydentävän (perfective) ylläpitoprosessin* tarkoituksena on parantaa tuotteen suorituskykyä ja muita laatutekijöitä. Ylläpitoprosessin aikana tuotteen toimintoja voidaan lisätä, poistaa tai muuttaa. Ensimmäisessä tapauksessa kaikki vanhat testitapaukset voidaan käyttää uudelleen. Kahdessa muussa tapauksessa jotkin testitapaukset täytyy poistaa testijoukosta ja tehdä uusia testitapauksia testaamaan muuttuneita toimintoja. Selkeästi suurin osa ohjelmistoille suoritettavista ylläpitotoimenpiteistä on luonteeltaan täydentävää ylläpitoa.
- *Sopeuttavassa (adaptive) ylläpitoprosessissa* muutetaan tuotetta vastaamaan muuttuneen toimintaympäristön vaatimuksiin. Esimerkiksi Windows-ympäristössä toimiva ohjelma muutetaan toimimaan Linux-ympäristössä.
- *Ennakoivan (preventive) ylläpitoprosessin* tarkoituksena on ennakoivasti muuttaa tuotetta, ennen kuin virheitä pääsee tapahtumaan. Esimerkiksi ennen vuosituhannen vaihdetta tehdyt korjaukset ohjelmiin, joissa vuosiluku ilmoitettiin kahdella numerolla.

Regressiotestaus voidaan jakaa kahteen ryhmään sen perusteella, mihin ylläpitoprosessiin se liittyy. *Korjaava regressiotestaus* (corrective regression testing) liittyy korjaavaan tai ennakoivaan ylläpitoprosessiin ja sille on ominaista mahdollisuus käyttää uudelleen suuri osa vanhoista testitapauksista. Tämä johtuu siitä, että ohjelman määrityksiin ei yleensä ole tullut muutoksia ja ohjelmakoodiin on tehty vain pieniä muutoksia. Tämän tyyppinen regressiotestaus suoritetaan epäsäännöllisin väliajoin. *Progressiivinen regressiotestaus* (progressive regression testing) liittyy sopeuttavaan tai täydentävään ylläpitoprosessiin. Koska tällöin ohjelmaan on tehty suuria muutoksia ja ohjelman määritykset ovat todennäköisesti muuttuneet, ei voida käyttää uudelleen yhtä suurta

osaa vanhoista testitapauksista kuin korjaavassa regressiotestauksessa. Progressiivinen regressiotestaus suoritetaan usein säännöllisin väliajoin. (Paakki, 2000)

Seuraavaksi on esitetty joitakin regressiotestausta vaativia tilanteita ja ehdotuksia, mitä kyseisissä tilanteissa kannattaisi testata (Binder, 1999, s. 760):

- Kun on kehitetty uusi aliluokka, yliluokan testitapaukset ajetaan aliluokalle. Tämän lisäksi täytyy aliluokka testata sille luoduilla uusilla testitapauksilla.
- Kun yliluokkaan tulee muutos, ajetaan yliluokan testitapaukset uudelleen yliluokalle sekä kaikille aliluokille. Tämän lisäksi ajetaan uudelleen aliluokan testitapaukset.
- Kun palvelinluokkaan tulee muutos, ajetaan palvelinluokan sekä kaikkien asiakasluokkien testitapaukset uudelleen.
- Kun on korjattu virhe, ajetaan virheen paljastanut testitapaus uudelleen. Mikäli testi menee läpi, ajetaan uudelleen kaikki testit, jotka testaavat sellaisia testattavan järjestelmän osia, joihin korjaus on voinut vaikuttaa.
- Kun järjestelmään liitetään uutta toiminnallisuutta, kannattaa järjestelmä testata uudelleen kaikilla aiempien toiminnallisuuksien testaamiseen kehitetyillä testijoukoilla, ennen kuin uudelle toiminnallisuudelle kehitetyt testitapaukset ajetaan. Näin toimimalla paljastetaan edellisissä toiminnallisuuksissa mahdollisesti piilevät virheet, yhteensopivuusongelmat sekä uuden koodin mahdollisesti aiheuttamat sivuvaikutukset.
- Kun järjestelmä on vakautunut ja lopullinen (final) ohjelmakooste on valmis, ajetaan järjestelmän koko testijoukko uudelleen.

Onoma tutkimusryhmineen (Onoma ym., 1998) ehdottaa, että regressiotestaus on syytä suorittaa aina, kun järjestelmään, eli ohjelmaan tai sen toimintaympäristöön tehdään muutoksia. Regressiotestaus kannattaa sulauttaa osaksi ohjelmistontuotanto- ja ylläpitoprosesseja sen sijaan, että se olisi erillinen vaihe edellä mainituissa prosesseissa. Esimerkiksi, kun moduuliin tehdään muutos, se kannattaa ensin uudelleentestata yksikkötestaustasolla (unit testing), ennen kuin se testataan yhdessä muiden moduulien kanssa integrointitasolla. Tällaista menetelmää kutsutaan *Monitasoiseksi Regressiotestaukseksi* (Multi-Level Regression Testing (MLRT)). MLRT-menetelmässä täysin samat

testitapaukset voivat tulla läpikäydyksi useaan kertaan, esimerkiksi kerran yksikkötasolla ja toistamiseen integrointitasolla. Tämä ei ole ajan tuhlausta, koska samat testit voivat paljastaa uusia virheitä, kun ne ajetaan eri tasoilla.

Tällaisella menetelmällä on useita etuja. Ensinnäkin testijoukot voidaan liittää eri tasoilla oleviin komponentteihin. Toiseksi komponentteja voidaan testata samanaikaisesti rinnakkain. Kolmas menetelmän tarjoama etu on virheiden havaitsemiseen kuluvan viiveen minimointi ja virheen paikantamisen helpottuminen. (Onoma ym., 1998, s. 85-86)

3 REGRESSIOTESTAUS KÄYTÄNNÖSSÄ

3.1 Regressiotestauksen vaiheet

Tässä luvussa esitetään erilaisia tulkintoja regressiotestauksessa suoritettavista vaiheista. Ensin kerrotaan Binderin tulkinta, jossa regressiotestauksella tarkoitetaan ainoastaan vanhojen testitapausten uudelleenkäyttöä. Seuraavaksi kerrotaan Onoman tutkimuksesta, jossa havaittiin useiden ohjelmistotalojen noudattavan tiettyjä askeleita regressiotestauksessaan. Lopuksi esitetään tässä tutkimuksessa kehitetty kuva regressiotestauksen eri vaiheista ja selitetään jokaisen vaiheen aikana tehtävät toimenpiteet.

Binder on kirjassaan (Binder, 1999, s.761) listannut seuraavat regressiotestauksen päävaiheet, jotka ovat samoja, suoritettiinpa regressiotestaus missä vaiheessa ohjelmiston kehitystä tahansa. Aluksi poistetaan alkuperäisestä testijoukosta käyttökelvottomat testitapaukset. Käyttökelvottomalla testitapauksella tarkoitetaan sellaista testitapausta, jota ei voida ajaa deltakoosteella esimerkiksi siksi, että testattava toiminto on poistettu. Seuraavaksi valitaan alkuperäisestä testijoukosta regressiotestijoukkoon mukaan otettavat testitapaukset, käyttämällä esimerkiksi jotakin regressiotestien valintatekniikkaa (luku 4.2). Toinen vaihtoehto on valita regressiotestijoukkoon kaikki alkuperäisen testijoukon testitapaukset. Tämän jälkeen luodaan testauskokoonpano ja ajetaan regressiotestijoukko. Lopuksi huomioidaan läpäisemättömät testit ja ryhdytään tarvittaviin toimenpiteisiin.

Onoma esitti vuonna 1998 julkaistussa artikkelissaan tutkimuksensa ohjelmistotalojen käytännön regressiotestauksesta. Tutkimuksessa huomattiin suuriman osan yrityksistä noudattavan seuraavia askelia regressiotestauksessaan (Onoma ym., 1998):

1. *Muutospyyntö*. Ohjelmasta löytynyt virhe, kuten myös ohjelman toiminnalliseen- tai tekniseen määrittelyyn tehtävä muutos johtaa muutospyynnön syntymiseen.
2. *Muutoksen tekeminen*. Usein muutokset tehdään lähdekoodiin, mutta myös määrittely- ja suunnitteludokumentteja voidaan joutua muuttamaan.

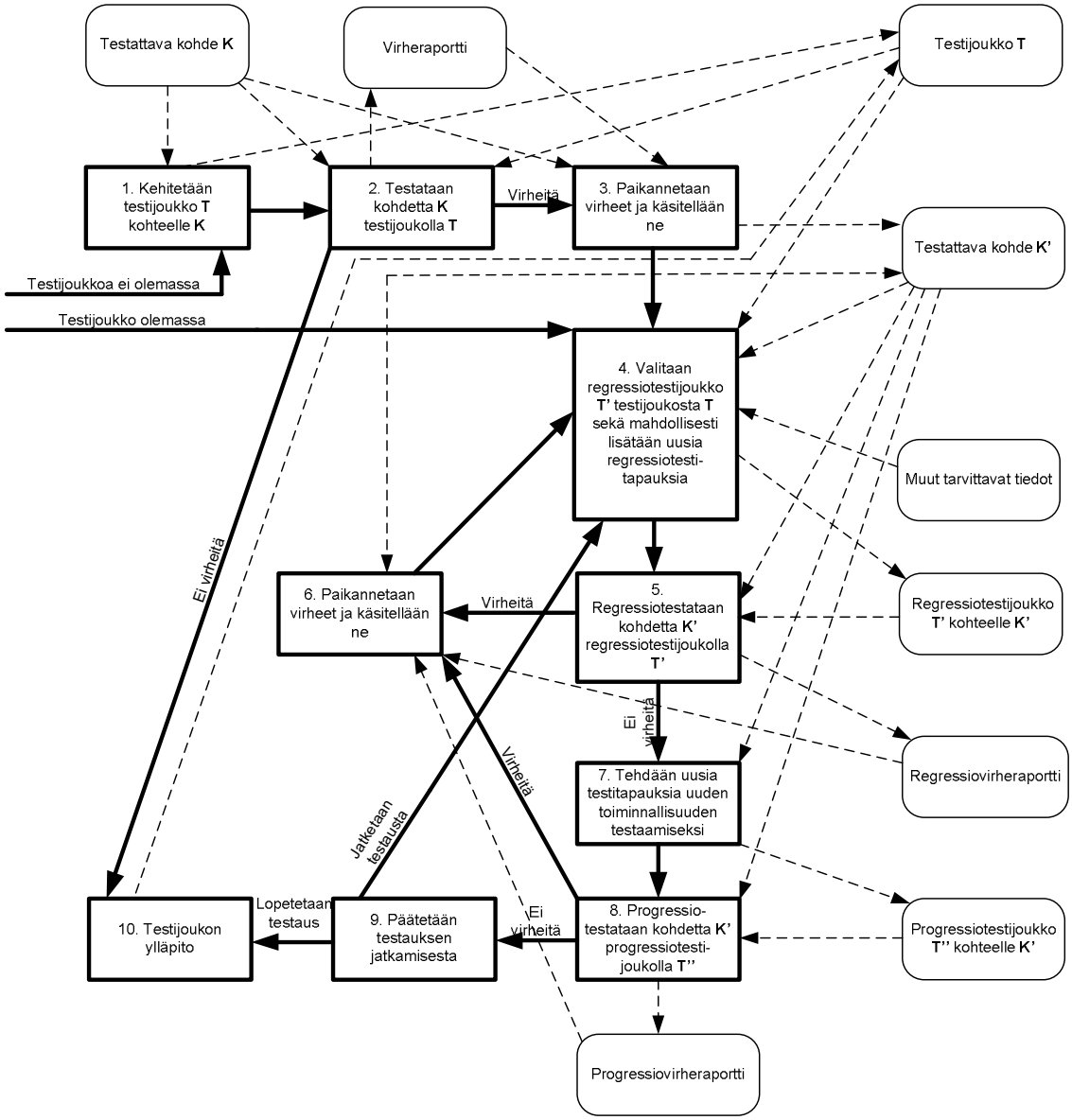
3. *Testitapausten valinta*. Valitaan regressiotestaukseen mukaan otettavat testitapaukset. Joskus testitapaukset uudelleenkelpoistetaan tässä vaiheessa. Tavoitteena on testitapausten lukumäärän minimoimisen sijaan valita oikeat testitapaukset. Joskus testaajat valitsevat uudelleen kaikki olemassa olevat testit (testaa kaikki uudelleen -tekniikka).
4. *Testijoukon ajaminen*. Valitut testit ajetaan. Tämä vaihe on yleensä automatisoitu, koska ajettavien testien määrä on usein suuri. Joskus tässä vaiheessa tallennetaan testien suoritushistoria (läpikäydyt polut ja aliohjelmakutsut) myöhempää käyttöä varten.
5. *Virheen löytäminen testituloksia tutkimalla*. Virheet löydetään yleensä vertaamalla saatuja testituloksia odotettuihin tuloksiin. Mikäli tulokset eivät ole yhdenmukaisia, pitää selvittää onko virhe koodissa vai testitapauksessa. Mikäli testitapauksia ei ole uudelleenkelpoistettu aiemmin, se tehdään tässä vaiheessa.
6. *Virheen tunnistus*. Jos lähdekoodin epäillään olevan virheellinen, täytyy ohjelmoijan paikantaa virhe. Virheen paikannus voi olla vaikeaa, mikäli ohjelmaan on tehty useita muutoksia ja ohjelma on muutenkin laaja. Virheen alkusyyn löytäminen on kuitenkin välttämätöntä.
7. *Virheen käsittely*. Kun virheen alkusyy on paikannettu, täytyy se käsitellä. Virhe voidaan käsitellä tekemällä ohjelmalle uusi muutospyyntö virheen korjaamiseksi tai poistamalla regressiovirheen syntymiseen johtanut muutos. Onoma huomasi tutkimuksissaan, että joissakin harvoissa tapauksissa virhe yksinkertaisesti jätettiin korjaamatta. Tämä tapahtui yleensä silloin, kun löydetty virhe ei ollut vakava ja sen korjaamiseen ei riittänyt aikaa.

Kuvassa 2 esitetään tämän tutkimuksen tuloksena regressiotestauksen eri vaiheet ja eri vaiheisiin liittyvät syötteet ja tulosteet. Tummennetuilla suorakaiteilla on merkitty eri tiloja. Tilojen syötteitä ja tuloksia on merkitty suorakaiteilla, joiden kulmat on pyöristetty. Mikäli tilan suorituksen vaatimuksena on tietty syöte, on tarvittavasta syötteestä vedetty nuoli kyseiseen tilaan. Jos tilassa suoritettujen toimenpiteiden tuloksena syntyy uusi tulos tai vanhaan tulokseen tulee muutoksia, on tilasta piirretty nuoli luotuun/muutettuun tulokseen. Tummennetut nuolet merkitsevät siirtymiä eri tilojen välillä.

Testattava kohde **K** voi olla mikä tahansa testattava kokonaisuus, kuten esimerkiksi ohjelma tai komponentti. Selvyyden vuoksi seuraavissa vaiheissa testauksen kohteeksi on valittu ohjelma. Vaiheista on kerrottu tarkemmin aliluvuissa 3.1.1-3.1.10:

1. Kun kohdetta **K** testataan ensimmäisen kerran, sille ei ole olemassa testijoukkoa. Tällöin kehitetään testijoukko **T** kohteelle **K**.
2. Kohdetta **K** testataan vaiheessa 1 luodulla testijoukolla **T**. Jos testauksessa ei löydetä virheitä, voidaan testaus päättää ja siirtyä vaiheeseen 10. Mikäli virheitä löytyy, siirrytään paikantamaan virheitä ja korjaamaan niitä vaiheeseen 3.
3. Löytyneet virheet paikannetaan ja käsitellään, jonka seurauksena testattavasta kohteesta **K** syntyy muutettu versio **K'**.
4. Valitaan alkuperäisen testijoukon **T** sisältämistä testitapauksista regressiotestijoukko **T'** kohteen **K'** vanhan toiminnallisuuden regressiotestaamiseksi. "Muut tarvittavat tiedot" -syötelaatikko ilmentää regressiotestien valintatekniikoiden tarvitsemia tietoja, joiden avulla testit voidaan valita. Esimerkiksi kattavuuteen perustuva valintatekniikka tarvitsee kattavuustiedot ohjelmasta ja testitapauksista. Tässä vaiheessa voidaan luoda uusia regressiotestitapauksia testaamaan vanhaa toiminnallisuutta.
5. Regressiotestataan komponenttia **K'** regressiotestijoukolla **T'**. Jos virheitä ei löydy, siirrytään vaiheeseen 7.
6. Testauksessa löytyneet virheet paikannetaan ja käsitellään. Jos kohdetta joudutaan muuttamaan virheiden korjaamiseksi, saadaan kohteesta uusi muutettu versio **K'**. Tämän jälkeen palataan vaiheeseen 4 tarkistamaan, pitääkö regressiotestijoukkoon tehdä muutoksia testattavaan kohteeseen tehtyjen muutosten seurauksena.
7. Jos kohteeseen ei ole lisätty uutta toiminnallisuutta voidaan vaiheet 7 ja 8 ohittaa ja siirtyä vaiheeseen 9. Mikäli kohteeseen **K'** on lisätty uutta toiminnallisuutta, tehdään progressiotestijoukko **T''** uuden toiminnallisuuden testaamiseksi. Tämän vaiheen (sekä seuraavan) ei katsota varsinaisesti kuuluvan regressiotestauksen piiriin.

8. Progressiotestataan kohdetta **K'** vaiheessa 7 luodulla progressiotestijoukolla **T''**. Mikäli virheitä löytyy, käsitellään ne vaiheessa 6. Muussa tapauksessa siirrytään vaiheeseen 9.
9. Tässä vaiheessa päätetään jatketaanko testausta ja siirrytään vaiheeseen 4, vai lopetetaanko ja siirrytään vaiheeseen 10. Päätöstä tehdessä pitää huomioida testauksessa saavutettu kattavuus sekä aika- ja kustannustekijät. Jos regressiotestaus päätetään lopettaa, yhdistetään testausprosessin aikana luodut testitapaukset testijoukkoon **T** ja siirrytään testijoukon ylläpitovaiheeseen (vaihe 10).
10. Testijoukon ylläpito. Regressiotestauskierrosten välissä testijoukko on säilössä. Ylläpitovaiheen aikana testijoukosta voidaan poistaa turhia testitapauksia.



Kuva 2. Testausprosessin vaiheet.

3.1.1 Vaihe 1: Kehitetään testijoukko T kohteelle K

Jos testattavalle ohjelmalle ei ole olemassa vanhaa testijoukkoa, aloitetaan tästä vaiheesta. Jos ohjelmalle on olemassa testijoukko, jolla ohjelmaa on testattu jo aiemmin, aloitetaan vaiheesta 4.

Ennen testauksen aloittamista tehdään *testaussuunnitelma* (esim. Paakki, 2000; Kaner ym., 1999), jossa suunnitellaan testausprosessin läpivienti. Testaussuunnitelmassa kerrotaan esimerkiksi, mitä osia ja asioita testausprosessissa tullaan testaamaan, aikatau-

lu testaukselle sekä testaukseen osallistuvat ihmiset ja muut resurssit (Paakki, 2000). Testaussuunnitelmaa päivitetään testausprosessin aikana.

Tässä vaiheessa kehitetään testitapauksia ohjelman toiminnallisuuden oikeellisuuden varmistamiseksi. Testitapausten tiedot kirjataan *testitapausten määrittelydokumentteihin*. Tärkeimpiä asioita testitapauksen luomisessa on ohjelmalle annettavien syötteiden, odotetun tuloksen sekä testitapauksen esi- ja jälkiehtojen määrittelyt. Näiden lisäksi testitapauksen määrittelydokumentissa selitetään lyhyesti testitapauksen testaamat ohjelman toiminnot ja osat sekä kerrotaan viitteet muihin asiaan liittyviin dokumentteihin. Jos testitapaus on riippuvainen toisista testitapauksista, ilmoitetaan riippuvuudet testitapauksen määrittelydokumentissa. (Paakki, 2000) Testitapauksen odotettu tulos voi olla mikä tahansa tulos, joka voidaan ohjelman toiminnan seurauksena todeta. Testitapauksen jälkiehdot kertovat, missä tilassa ohjelman ja suoritusympäristön pitää olla testin suorituksen jälkeen. Odotetut tulokset saadaan yleensä ohjelman määrittelyistä, joissa kerrotaan mitä ohjelman pitäisi tehdä. Luodut testitapaukset lisätään ohjelman testijoukkoon **T**. Automaattista testitapausten luontia käsitellään kirjassa Software Test Automation (Fewster ym., 1999).

Jos regressiotestaus halutaan automatisoida, kannattaa odotetut tulokset tallentaa erilliseen tietokantaan (oraakkeli), josta automatisoinnissa käytettävä tuloksen vertailijatyökalu ne löytää. Oraakkelilla voidaan Binderin (Binder, 1999) mukaan tarkoittaa mitä tahansa luotettavaa tietoa odotetusta tuloksesta, kun ohjelmaa ajetaan tietyillä syötteillä. Yleensä oraakkelilla tarkoitetaan tapaa saada selville odotettu tulos sekä myös tapaa verrata odotettua tulosta ohjelmasta testauksen aikana saatuun tulokseen.

Fewster kertoo kirjassaan (Fewster ym., 1999, s.13-17) testausprosessin viisi askelta, joista kolme ensimmäistä askelta liittyy testijoukon kehittämiseen. Ensin tunnistetaan, mitä ohjelmasta pitää testata. Seuraavaksi päätetään, miten edellisessä vaiheessa tunnistetut testattavat asiat voidaan käytännössä testata. Tässä vaiheessa suunnitellaan testitapausten syötteet, odotetut tulokset sekä testitapausten suorituksen esi- ja jälkiehdot. Kolmannessa vaiheessa rakennetaan testitapaukset. Automaattisessa testauksessa luodaan testiskriptit ja tallennetaan odotetut testitapaukset tietokantaan, josta odotettua ja saatua tulosta vertaileva työkalu ne löytää. Automaattisen testauksen valmistelu vie paljon enemmän aikaa kuin manuaalisen testauksen valmistelu, koska automaattisessa

testauksessa varsinkin odotettujen tulosten syöttäminen työkaluille on monimutkaisempaa kuin manuaalisessa testauksessa. (Fewster ym., 1999)

Onoma kertoo käytännön regressiotestausta tutkivassa artikkelissaan (Onoma ym., 1998), että testitapausten määritykset ovat usein puutteellisia. Tähän on syynä suurten ohjelmien monimutkaisuus, mistä johtuen ohjelmien esi- ja jälkiehtojen tarkka määrittäminen on vaivalloista ja vaikeaa. Puutteellisista esiehtojen määrityksistä johtuen testitapaus saattaa toimia eri tavalla eri suorituskerroilla. Puutteellisten jälkiehtojen takia testitapaus saattaa mennä läpi, vaikka se on saattanut aiheuttaa ohjelman suoritusympäristöön epätoivottuja muutoksia. (Onoma ym., 1998)

3.1.2 Vaihe 2: Testataan kohdetta K testijoukolla T

Tässä vaiheessa ajetaan ensimmäisessä vaiheessa kehitetyt testitapaukset ohjelmalle **K**. Testitapaus ajetaan asettamalla ohjelma ensin testitapauksen esiehtojen määrittämään tilaan ja antamalla ohjelmalle testitapauksessa kerrotut syötteet. Tämän jälkeen tarkastetaan ohjelmasta saatu tulos ja jälkiehdot. Näiden selvittäminen ei ole aina yksinkertaista vaan saattaa vaatia työkalujen käyttöä. Saatua tulosta verrataan testitapauksessa määritettyyn odotettuun tulokseen. Mikäli saatu tulos on sama kuin odotettu tulos ja jälkiehdot täyttyvät, ohjelma läpäisee testin. Jos saatu tulos poikkeaa odotetusta, on ohjelmassa virhe. Tämä olettaen, että testitapaus on luotu huolellisesti ja on näin ollen virheetön.

Jokainen ohjelmasta löytynyt virhe dokumentoidaan huolellisesti niin, että virhe pystytään toistamaan myöhemmin ja paikantamaan se vaiheessa 3 (luku 3.1.3). Löydetyt virheet kirjataan *virheraporttiin* (Kaner ym., 1999), jossa selitetään, millainen virhe on kyseessä, millä ohjelmaversiolla virhe havaittiin sekä mahdolliset viitteet muihin dokumentteihin. Virheraportissa kerrotaan, miten virhetilanteeseen päädyttiin ja kuinka virhe voidaan toistaa (jos voidaan). Mahdollisesti virheraportissa annetaan myös ehdotus, kuinka virhe tulee korjata. Virheraporttiin merkitään myös virheen kriittisyysaste, eli missä järjestyksessä virheet pitäisi korjata. Paakki luettelee monisteessaan neljä prioriteettiastetta (urgent, high, medium, low). (Paakki, 2000)

Tässä vaiheessa on mahdollista hankkia myös testitapausten kattavuustiedot, mikäli halutaan käyttää koodiin perustuvia regressiotestausta tehostavia metodologioita. Kattavuustietojen keräämiseen voidaan käyttää erilaisia tekniikoita (coverage monitor, cove-

rage analyzer, profilointitekniikat). Useimmat kattavuuden mittaamiseen käytettävät työkalut toimivat staattisesti, eli ne käsittelevät koodia ennen testijoukon suoritusta niin, että ajettaessa testitapauksia nähdään niiden läpikäymät entiteetit. Koodin käsittely voi kuitenkin kasvattaa ohjelman kokoa ja hidastaa ohjelman suoritusta merkittävästi, jolloin esimerkiksi reaaliaikaisen ohjelman testaus vaikeutuu. Koko ohjelman käsittely aiheuttaa myös turhia kustannuksia, mikäli testijoukko kattaa vain pienen osan ohjelmasta. Tikir ym. (Tikir ym., 2002) esittelevät dynaamisen tavan käsitellä koodia. Tikirin tekniikka lisää ohjelmaan kattavuuden mittaamiseksi tarkoitettua käsittelykoodia ajon aikana ja poistaa lisätyn koodin, kun sitä ei enää tarvita. (Tikir ym., 2002)

Jos ohjelmasta löytyi virheitä, siirrytään vaiheeseen 3 (luku 3.1.3), muuten voidaan siirtyä vaiheeseen 9 (luku 3.1.9). Koska ohjelmalle luotiin vaiheessa 1 uusi testijoukko, on hyvin harvinaista, ettei testijoukko paljastaisi yhtään virhettä. Käytännössä hyvin tehty testijoukko löytää aina virheitä.

3.1.3 Vaihe 3: Paikannetaan virheet ja käsitellään ne

Tässä vaiheessa paikannetaan löydetyt virheet käyttämällä apuna edellisessä vaiheessa luotua virheraporttia ja päätetään, mitä virheille tehdään. Virheiden paikantaminen saattaa olla hyvinkin vaikeaa varsinkin monimutkaisissa ja paljon riippuvuussuhteita sisältävissä ohjelmissa. Virheiden paikantamista on käsitelty esimerkiksi Virkasen Pro gradu -tutkielmassa (Virkanen, 2002). Virhe voidaan paikantaa käyttämällä staattisia tai dynaamisia virheenjäljitystyökaluja. Staattisia virheenjäljitystyökaluja, kuten kääntäjiä, voidaan käyttää ilman ohjelman ajamista. Dynaamisten virheenjäljitystyökalujen, kuten debugger, käyttämiseksi pitää ohjelmaa samalla suorittaa. (Virkanen, 2002) Koodivirheen paikantamisen tekevät yleensä ohjelmoijat, koska he tuntevat parhaiten ohjelman toiminnan.

Virheen paikantamisen jälkeen päätetään virheen käsittelystä. Yleensä lähetetään korjauspyyntö virheen tehneelle taholle, joka päättää miten virhe korjataan. Muutosten suunnittelussa voidaan käyttää apuna muutosvaikutusanalyysiä (impact analysis) (luku 3.2), jonka avulla voidaan ennakoida muutosten vaikutusta ohjelmaan ja sen regressiotestaukseen. Tässä vaiheessa korjataan myös mahdolliset ohjelman määrityksissä olevat virheet muuttamalla ohjelman määritysdokumenttia.

3.1.4 Vaihe 4: Valitaan regressiotestijoukko T' testijoukosta T sekä mahdollisesti lisätään uusia regressiotestitapauksia

Tästä vaiheesta lähtien voidaan katsoa regressiotestaus alkaneeksi. Vaiheisiin 1-3 (luvut 3.1.1-3.1.3) ei enää mennä ohjelman elinkaaren aikana. Kun ohjelmaa joudutaan sen elinkaaren aikana testaamaan, aloitetaan suoraan tästä vaiheesta. Jos tähän vaiheeseen tullaan ohjelman ylläpitovaiheessa, käytetään vanhaa ohjelmalle tehtyä testaussuunnitelmaa, jota päivitetään regressiotestausprosessin aikana.

Tässä vaiheessa kootaan regressiotestijoukko, jolla varmistetaan, että ohjelmaan tehdyt muutokset ovat oikeellisia, eivätkä ne ole aiheuttaneet virheitä muualla ohjelmassa. Tämä vaihe jakaantuu kolmeen askeleeseen: 1. Ensin lisätään regressiotestijoukkoon testitapaukset, joilla varmistetaan ohjelmaan mahdollisesti tehdyn korjauksen toimivuus. 2. Seuraavaksi regressiotestijoukkoon lisätään vanhoja testitapauksia, joilla varmistetaan, että korjaus ei ole aiheuttanut virheitä muualla ohjelmassa. 3. Lopuksi kehitetään uusia regressiotestitapauksia vanhan toiminnallisuuden testaamiseksi ja lisätään ne regressiotestijoukkoon. Seuraavaksi kerrotaan tarkemmin, mitä eri askelten kohdalla tehdään.

Askel 1. Jos ohjelmaan tehtiin muutoksia virheiden korjaamiseksi, valitaan regressiotestijoukkoon virheet alun perin paljastaneet testitapaukset. Näin varmistetaan, että virhe on todellakin korjattu. Tässä askeleessa voidaan myös luoda uusia regressiotestitapauksia korjauksen oikeellisuuden varmistamiseksi.

Askel 2. Seuraavaksi regressiotestijoukkoon valitaan vanhoja, aiemmin läpimenneitä testitapauksia. Helpoin tapa on käyttää uudelleen kaikki alkuperäisessä testijoukossa olevat vanhat testitapaukset, mutta ajettavien testitapausten määrää voidaan rajata käyttämällä apuna regressiotestitapausten valintatekniikoita. Valintatekniikoita on esitelty luvussa 4.2. Valintatekniikoiden käyttämisestä tulevat hyödyt riippuvat monista eri asioista. Valintatekniikoiden tehokkuuteen vaikuttavia asioita käsitellään luvussa 4.3. Mitä enemmän ohjelman alkuperäisessä testijoukossa on testitapauksia, sitä suurempia säästöjä valintatekniikoiden käytöllä voidaan saavuttaa. Ensimmäisillä regressiotestauskierroksilla alkuperäinen testijoukko on todennäköisesti pieni, joten koko testijoukon uudelleenajaminen ei aiheuta suuria kustannuksia, mutta ohjelman elinkaaren aikana testijoukkoon lisätään testejä ja sen ajamiskustannukset kasvavat.

Koodiin pohjautuvat valintatekniikat tarvitsevat tiedot testitapausten kattavuudesta ohjelmassa. Nämä tiedot ovat peräisin yleensä testitapausten edelliseltä ajokerralta. Tuoreiden kattavuustietojen saaminen edellyttäisi ohjelman ajamista testijoukolla. Näin tehtäessä ei valintatekniikoista olisi mitään hyötyä, koska valintatekniikoiden hyödyt tulevat ajettavien testitapausten määrän pienenemisestä. Tämän vuoksi täytyy käyttää hyväksi viimeisimmän ajokerran kattavuustietoja.

Kattavuustietojen on huomattu joissain tutkimuksissa (esim. Rosenblum ym., 1997) olevan suhteellisen pysyviä, kun taas toisissa tutkimuksissa (Elbaum ym., 2001a) kattavuustietojen on huomattu muuttuvan suuresti ohjelmaan tehtyjen muutosten seurauksena.

Koska ohjelmaan on tullut muutoksia, eivät kaikki vanhat testitapaukset välttämättä ole käyttökelpoisia uuden ohjelmaversion testaamiseen. Tämän takia testitapaukset täytyy uudelleenkelpoistaa (revalidate), eli tarkistaa, että testitapaus toimii oikein ja on hyödyllinen muutetun ohjelman testaamiseen. Käyttökelvottomat testitapaukset pitää korjata tai poistaa testijoukosta. Artikkelissa (Onoma ym., 1998) kerrotaan testitapausten uudelleenkelpoistamisen olevan suurimmaksi osaksi manuaalinen prosessi ja vievän runsaasti aikaa suurten testijoukkojen kohdalla. Onoman mukaan uudelleenkelpoistaminen voidaan tehdä ennen regressiotestien valintaa koko testijoukolle tai vasta testien suorituksen jälkeen läpäisemättömille testitapauksille. Testauksen jälkeen tehtävässä uudelleenkelpoistamisessa tutkitaan vain läpäisemättömiä testitapauksia, jolloin tutkittavien testitapausten määrä on pienempi verrattuna siihen, että uudelleenkelpoistaminen suoritettaisiin ennen regressiotestauksen suorittamista koko testijoukolle. Toisaalta pelkästään läpäisemättömille testitapauksille tehtävä uudelleenkelpoistaminen ei ole turvallista, koska osa epäkelvoista testitapauksista läpäisee ohjelman ja näin ollen osa virheistä saattaa jäädä huomaamatta.

Askel 3. Lopuksi regressiotestijoukkoon lisätään uusia regressiotestitapauksia varmistamaan ohjelmassa ennen muutosta olleiden toiminnallisuuksien oikeellisuus muutetussa versiossa. Jos ohjelmaan tehty muutos on ollut pieni, ei uusia regressiotestitapauksia välttämättä tarvita. Uusia regressiotestitapauksia joudutaan tekemään, mikäli muutosten seurauksena testijoukosta on jouduttu poistamaan korjauskelvottomia testitapauksia ja testijoukon kattavuus on tämän seurauksena laskenut.

Ideat uusien regressiotestitapausten luomiseen voivat tulla testausryhmän ulkopuolelta, kuten esimerkiksi asiakkailta. Kanerin mukaan jotkut testausryhmät tekevät regressiotestitapauksen jokaisesta ulkopuolelta ilmoitetusta virheraportista (Kaner ym., 1999).

3.1.5 Vaihe 5: Regressiotestataan kohdetta K' regressiotestijoukolla T'

Tässä vaiheessa testataan komponenttia **K'** valitulla regressiotestijoukolla **T'**. Vanhoja testitapauksia ajettaessa voidaan saatua tulosta verrata edellisellä ajokerralla saatuun tulokseen. Jos saatu tulos poikkeaa aiemmin saadusta odotetusta tuloksesta, on ohjelmaan muutoksen seurauksena syntynyt regressiovirhe. Löytyneet virheet kirjataan regressiovirheraporttiin ja käsitellään vaiheessa 6 (luku 3.1.6). Tässä vaiheessa on mahdollista hankkia myös ajettavien testitapausten uudet kattavuustiedot. Kattavuustietojen keräämistä käsiteltiin vaiheessa 2 (luku 3.1.2).

Mikäli regressiotestitapausten kattavuus tiedetään, testauksessa voidaan käyttää apuna regressiotestien priorisointitekniikoita, jotka järjestävät testitapaukset yleensä niin, että virheet löytyisivät testauksen alkuvaiheessa. Priorisointitekniikoista kerrotaan tarkemmin luvussa 4.1.1.

3.1.6 Vaihe 6: Paikannetaan virheet ja käsitellään ne

Tässä vaiheessa paikannetaan löydetyt virheet käyttämällä apuna testauksessa luotua virheraporttia ja päätetään miten virheet käsitellään. Virheiden käsittelystä kerrotaan vaiheessa 3 (luku 3.1.3). Jos virheitä käsiteltäessä tehdään ohjelmaan muutoksia, palataan vaiheeseen 4 (luku 3.1.4) tarkistamaan pitääkö ohjelmaan tehdyn muutoksen seurauksena tehdä muutoksia ohjelman regressiotestijoukkoon.

Mikäli ohjelmaan tehty muutos aiheutti ohjelmassa runsaasti uusia virheitä, voidaan muutoksen korjaamisen sijasta perua muutos kokonaan ja palata muutosta edeltävään ohjelmaversioon. Koska vanha ohjelmaversio on jo testattu, ei siitä todennäköisesti löydetä enää uusia virheitä vanhoilla testitapauksilla.

Muutoksen peruminen on vaikeaa, mikäli muut muutokset ovat riippuvaisia peruttavasta muutoksesta. Tällöin peruttavan muutoksen lisäksi pitää luopua myös muista

riippuvista muutoksista. Jos testaukselle suunniteltu aika on vähissä ja virheen katsotaan olevan vähäinen, voidaan virhe jättää kokonaan huomiotta. Muutoksen huomiotta jättämiselle täytyy aina hankkia projektipäällikön hyväksyntä. (Onoma ym., 1998)

3.1.7 Vaihe 7: Tehdään progressiotestitapauksia uuden toiminnallisuuden testaamiseksi

Tässä vaiheessa tehdään uusia testitapauksia ohjelmaan **K'** lisätylle uudelle toiminnallisuudelle ja lisätään ne progressiotestijoukkoon **T''**. Nämä testitapaukset eivät ole regressiotestitapauksia, koska ne eivät testaa ohjelman vanhaa toiminnallisuutta. Uusien testitapausten luomisesta kerrotaan tarkemmin vaiheessa 1 (luku 3.1.1).

Jos testattavaan kohteeseen ei ole lisätty uutta toiminnallisuutta, voidaan vaiheet 7 ja 8 ohittaa.

3.1.8 Vaihe 8: Testataan kohdetta K' progressiotestijoukolla T''

Tässä vaiheessa ajetaan vaiheessa 7 (luku 3.1.7) tehdyt progressiotestitapaukset ohjelmalle **K'**. Löytyneet virheet kirjataan progressiovirheraporttiin ja käsitellään vaiheessa 6 (luku 3.1.6). Jos virheitä ei löydy, siirrytään vaiheeseen 9 (luku 3.1.9).

3.1.9 Vaihe 9: Päätetään testauksen jatkamisesta

Tässä vaiheessa päätetään testauksen jatkamisesta. Testausta voidaan jatkaa, mikäli testauksen kattavuustavoitteet eivät ole täyttyneet. Yleensä kattavuuskriteerit on pyritty täyttämään kehitettäessä testijoukkoa vaiheissa 4 (luku 3.1.4) ja 7 (luku 3.1.7), mutta tavoitteista on voitu jäädä jälkeen testausprosessissa kohdattujen ongelmien vuoksi. Esimerkiksi joistakin ohjelman osista on voitu testauksen aikana löytää odottamattoman paljon virheitä joten voidaan päättää, että tällaiset osat tulisi testata ennakoitua suuremmalla määrällä testitapauksia. Jatkamispäätöstä tehtäessä pitää huomioida käytettävissä oleva aika ja testauksesta aiheutuvat kustannukset. Mikäli testausta päätetään jatkaa, siirrytään vaiheeseen 4 (luku 3.1.4).

Kun testaus lopulta päätetään, tehdään testausprosessista *yhteenvetoraportti*. Yhteenvetoraportissa arvioidaan testausprosessin tuloksia, kerrotaan testatut osat ja asiat, testausympäristö ja viitteet testausprosessin aikana tehtyihin dokumentteihin. Yhteenvetoraportissa arvioidaan, miten hyvin testausprosessissa onnistuttiin täyttämään testaus-

suunnitelmassa kerrotut kriteerit ja mitä ei ehditty testaamaan tarpeeksi perusteellisesti. Yhteenvetoraportissa kerrotaan myös tärkeimmät testaustoimenpiteet ja tapahtumat ja kerrotaan, miten paljon resursseja (esim. henkilötyötunnit, tietokonetunnit, tarvittu henkilökunta) kului testauksen suoritukseen. (Paakki, 2000)

Lopuksi yhdistetään vaiheessa 4 (luku 3.1.4) luodut uudet regressiotestitapaukset ja vaiheessa 7 (luku 3.1.7) luodut progressiotestitapaukset vanhojen testitapausten joukkoon **T**. Jos vanhoihin testitapauksiin on tehty regressiotestauksen aikana muutoksia, korvataan vanhat testitapaukset päivitetyillä versioilla.

Lopuksi päätetään testausprosessi ja siirrytään ylläpitämään testijoukkoa vaiheeseen 10 (luku 3.1.10).

3.1.10 Vaihe 10: Testijoukon ylläpito

Ohjelman regressiotestauskierrosten välissä testijoukko on säilössä. Ohjelman elinkaaren aikana testijoukkoon lisätään uusia testitapauksia ja sen koko kasvaa. Samalla kasvavat myös testijoukon uudelleenkelpoistamis- sekä ajamiskustannukset ja regressiotestausta tehostavien metodologioiden sisältämien tekniikoiden käyttäminen hidastuu.

Testijoukon kokoa voidaan pienentää poistamalla siitä testitapauksia pysyvästi käyttämällä testijoukkojen harventamistekniikoita (luku 4.1.2). Suurin osa harventamistekniikoista etsii testijoukosta sellaiset testitapaukset, joiden poistaminen ei vähennä testijoukon kattavuutta. Vaikka testijoukon kattavuus pysyykin samana, saattaa testijoukon kyky löytää virheitä alentua poistojen seurauksena.

Binder kertoo kirjassaan Testing object-oriented systems (Binder, 1999) testijoukkojen ylläpidosta ja erottaa testijoukosta neljänlaisia testitapauksia, joita kannattaa (tai pitää) poistaa. *Rikkinäiset testitapaukset* käyttävät jotakin muutettua tai poistettua ohjelman osaa. Testitapauksista voi tulla tällaisia, mikäli ohjelman määrityksiä muutetaan. *Vanhentuneet testitapaukset* saattavat antaa vääränlaisia tuloksia, koska testitapauksen määritykset ovat vanhentuneita suhteessa ohjelman määrityksiin. Testitapauksista voi tulla vanhentuneita ohjelman määrityksiin tehtyjen muutosten seurauksena. Esimerkiksi jos testitapauksen testaaman muuttujan yläraja on nostettu sadasta kymmeneentuhanteen, vanhentunut testitapaus tulkitsee virheellisesti syötetyn arvon 101 epäkelvoksi muutetussa ohjelmassa. *Kontrolloimattomat testitapaukset* ovat herkkiä ohjelman kont-

rolloimattomille syötteille tai tiloille. Kontrolloimaton testitapaus voi antaa samalla ohjelmaversiolla ajettuna erilaisia tuloksia. *Tarpeettomilla testitapauksilla* on täysin samanlaiset syötteet tietylle rajapinnalle kuin toisellakin testitapauksella. Aina samanlaiset testitapaukset eivät ole tarpeettomia, koska joskus testejä toistetaan tietyn tilan tai aikaviiveen aikaansaamiseksi. (Binder, 1999, s. 768).

3.2 Muutosvaikutusanalyysi

Ohjelman muutosvaikutusanalyysillä tarkoitetaan ohjelmaan suunnitellun tai jo tehdyn muutoksen vaikutuksen analysointia ohjelman testaamisen kannalta. Luvussa 3.2.1 kerrotaan yleistä asiaa muutosvaikutusanalyysistä ja luvussa 3.2.2 kerrotaan eräästä tutkimuksesta muutosvaikutusanalyysitekniikoista.

3.2.1 Yleistä

Ohjelman muutosvaikutusanalyysin (software change impact analysis) tarkoituksena on ennen muutoksen tekemistä arvioida, mihin ohjelman osiin suunniteltu muutos mahdollisesti vaikuttaa tai mihin ohjelmaan jo tehty muutos on todennäköisesti vaikuttanut. Ennen muutosta tehtävä analyysi auttaa muutosten hinnan arvioimisessa ja vaihtoehtoisten toteutustapojen valinnassa. Muutoksen tekemisen jälkeen muutosvaikutusanalyysistä saatuja tietoja voidaan käyttää lisätestausta vaativien ohjelman osien löytämiseen. Muutosvaikutusanalyysin tekeminen suuriin ohjelmistoihin on kallista, koska yleensä se tehdään ilman automatisointia. Tämän vuoksi muutosvaikutusanalyysi tehdään vain harvoin, eli silloin, kun muutoksen tekemiseen liittyy suuria kustannuksia.

Suosittu tapa arvioida osat, joihin muutos mahdollisesti vaikuttaa, on käyttää hyväksi ohjelman osien riippuvuussuhteita. Orso mainitsee artikkelissaan (Orso ym., 2003), että riippuvuussuhteet voidaan selvittää käyttämällä ohjelmaan eteenpäin laskevia viipalointitekniikoita (forward slicing approach) tai laskemalla ohjelmalle transitiivinen sulkeuma alkaen muutoksen tekokohdasta (esimerkiksi Bohner ym., 1996). Jos esimerkiksi laskettaisiin transitiivinen sulkeuma muutetulle komponentille, tulosjoukossa olisivat kaikki ne komponentit, joihin muutettu komponentti on suorasti tai epäsuorasti yhteydessä. Olio-ohjelmoinnissa monimutkaiset periytymissuhteet vaikeuttavat

transitiivisen sulkeuman laskemista. Viipalointitekniikat ja transitiivisen sulkeuman laskeminen eivät kuitenkaan ole tarkkoja ja saattavat yliarvioida muutoksen vaikutusta ohjelmaan.

Kaksi uudempaa tekniikkaa PathImpact (Law ym., 2003) ja CoverageImpact (Orso ym., 2003) käyttävät hyväkseen ohjelmasta kerättävää dynaamista, eli ajonaikaista tietoa ja ovat tarkempia kuin edellä mainitut tekniikat. Toisaalta näin kerätty tieto ei ole aivan yhtä turvallista, kuin staattisilla tekniikoilla kerätty.

3.2.2 Orson tutkimus muutosvaikutusanalyysitekniikoista

CoverageImpact (Orso ym., 2003) ja PathImpact (Law ym., 2003) toimivat perusperiaatteiltaan samalla tavalla, mutta käytännössä niiden toiminta eroaa toisistaan. Tutkimuksessa (Orso ym., 2004) vertailtiin kahden edellä mainitun tekniikan toimintaa niiden tarkkuuden ja tehokkuuden, eli tila- ja aikavaatimusten, perusteella.

Tutkimuksessa huomattiin CoverageImpactin olevan aika- ja tilavaatimuksiltaan huomattavasti PathImpactia halvempi, mutta samalla myös epätarkempi. CoverageImpact-tekniikan ei tarvitse käsitellä dynaamista tietoa, joten sen aikavaatimukset olivat mitättömät. CoverageImpact-tekniikka ei myöskään tarvitse tietoja ohjelman poluista ja sen tilavaatimukset pysyvät hyvin pieninä. PathImpact-tekniikan tila- ja aikavaatimukset kasvavat sen mukaan, miten pitkään ohjelma on päällä, eli miten suuria suorituspolkuja ohjelman aikana syntyy. Suorituspolkujen koot saattavat PathImpact-tekniikan tapauksessa olla useita gigatavuja.

CoverageImpact soveltuu siis PathImpact-tekniikkaa paremmin käytettäväksi suuria suorituspolkuja luovissa ohjelmissa. Toisaalta CoverageImpact on joskus paljon PathImpact-tekniikkaa epätarkempi.

Orson tutkimuksessa huomattiin näiden tekniikoiden olevan paljon staattisia analyysimenetelmiä taloudellisimpia tilanteissa, joissa turvallisuus ei ole pääasia.
(Orso ym., 2003; Orso ym., 2004)

4 REGRESSIOTESTAUSTA TEHOSTAVAT METODOLOGIAT

4.1 Yleistä

Onoma kertoo artikkelissaan (Onoma ym., 1998) Yhdysvalloissa ja Japanissa tehdyistä tutkimuksista, joiden mukaan regressiotestaus on yleisesti käytössä yritysmaailmassa ja usein regressiotestaukseen valitaan mukaan kaikki olemassa olevat testitapaukset. Ohjelman elinkaaren aikana testijoukkoon lisätään uusia testitapauksia ja testijoukon ajaminen vie yhä enemmän aikaa. Kun testijoukko kasvaa liikaa, ei koko testijoukon uudelleen käyttäminen regressiotestauksessa ole enää käytännöllistä. Useissa tutkimuksissa on tutkittu, kuinka regressiotestien valintaa ja regressiotestausta yleensä voitaisiin tehostaa.

Tässä luvussa esitellään metodologiat, joiden alle regressiotestausta tehostavat tekniikat kuuluvat. Regressiotestauksen kustannustehokkuuden parantamiseksi on olemassa useita erilaisia metodologioita. Näiden joukosta voidaan löytää kolme jo olemassa olevia testitapauksia hyödyntävää metodologiaa:

1. *Regressiotestien valintatekniikat* (regression test selection) valitsevat osan ohjelmalle olemassa olevista testitapauksista käytettäväksi regressiotestauksessa.
2. *Testitapausten priorisointi* (test case prioritization) järjestää testitapaukset niin, että testauksen kannalta hyödyllisimmät testitapaukset suoritetaan ensin.
3. *Testijoukon harventaminen* (test-suite reduction, test-suite minimization) pyrkii alentamaan tulevien regressiotestausvaiheiden kustannuksia poistamalla testitapauksia testijoukosta pysyvästi.

Testitapausten priorisointitekniikoita käsitellään luvussa 4.1.1 ja testijoukon harventamistekniikoita käsitellään luvussa 4.1.2. Regressiotestien valintatekniikat ovat tehokkain ja eniten tutkituin regressiotestauksen tehostamiseen pyrkivä metodologia, joten tässä tutkielmassa suurin huomio kiinnitetään valintatekniikoihin. Regressiotestien va-

lintatekniikoista kerrotaan yleisesti luvussa 4.2, minkä lisäksi luvussa 5 kerrotaan tarkemmin UML-määrityksiin pohjautuvasta regressiotestien valinnasta. Valinta-, priorisointi- ja harventamis-metodologioiden tehokkuuteen vaikuttavia asioita tarkastellaan luvussa 4.3. Regressiotestausta voidaan tehostaa huomattavasti testauksen automatisoinnilla. Asiaa käsitellään lyhyesti luvussa 4.4.

4.1.1 Testitapausten priorisointitekniikat

Priorisointitekniikoiden tarkoituksena on järjestää testitapaukset niin, että ajettaessa ne saavuttavat nopeammin tietyn päämäärän. Päämäärä voi olla esimerkiksi saavuttaa nopeasti mahdollisimman suuri koodikattavuus tai testata ensin ohjelman käytetyimmät ominaisuudet. (Rothermel ym., 2001)

Yleinen tapa verrata priorisointekniikoita on tutkia niiden kykyä järjestää testitapaukset sellaiseen järjestykseen, että ohjelman virheet löytyvät mahdollisimman varhaisessa vaiheessa. (Elbaum ym., 2004) Tässä tutkielmassa priorisointitekniikoiden tehokkuudella tarkoitetaan tekniikoiden kykyä järjestää testitapaukset edellä mainitun päämäärän mukaisesti.

Virheiden aikainen löytyminen säästää aikaa ja resursseja, koska virheiden korjaaminen voidaan tällöin aloittaa aiemmin. Virheiden aikaisen löytymisen tärkeys korostuu isojen testijoukkojen kohdalla. Virheiden nopeammasta löytymisestä on hyötyä myös siinä tapauksessa, että testausprosessi joudutaan syystä tai toisesta keskeyttämään, jolloin loppupään testitapauksia ei ehditä käymään läpi. Priorisointitekniikoiden tuomat hyödyt kasvavat ajettavan testijoukon koon mukaan. Jos testijoukon ajaminen kestää vain vähän aikaa, ei priorisointitekniikoista ole mainittavaa hyötyä.

Priorisointitekniikat eivät poista testijoukosta yhtään testiä, vaan ne ainoastaan uudelleenjärjestävät testit (Srivastava ym., 2002). Priorisointitekniikoita voidaan käyttää yhdessä muiden regressiotestausta tehostavien metodologioiden kanssa.

Tutkittujen priorisointitekniikoiden tehokkuus riippuu muun muassa järjestettävän testijoukon ja testattavan ohjelman ominaisuuksista. Tähän päivään mennessä ei ole saatu kehitettyä sellaista menetelmää, mikä kertoisi tietynlaiseen tapaukseen parhaiten soveltuvan priorisointitekniikan. Priorisointitekniikoiden käytännön soveltamisesta on tehty tutkimuksia, joiden tulokset eivät ole täysin yhdenmukaisia. Joissain tutkimuksissa

(Elbaum ym., 2002; Rothermel ym., 2001; Srivastava ym., 2002) tutkitut tekniikat ovat lisänneet virheiden löytymisnopeutta merkittävästi, mutta toisissa tutkimuksissa (Elbaum ym., 2001b; Elbaum ym., 2003) virheiden löytymisnopeus on vaihdellut suuresti ohjelman ja testijoukon ominaisuuksien mukaan. (Elbaum ym., 2004)

Tämän tutkielman liitteenä on selostus Elbaumin priorisointitekniikoista tekemästä tutkimuksesta (liite A). Tutkimuksessa saatiin tulokseksi todennäköisyyksiä, joiden avulla voidaan päätellä eri tilanteisiin parhaiten soveltuvia priorisointitekniikoita.

Testitapausten välillä voi olla riippuvuussuhteita niin, että testitapaukset pitää ajaa tietyssä järjestyksessä oikeellisen tuloksen saamiseksi (Onoma ym., 1998). Tällaista useista toisistaan riippuvasta testitapauksesta koostuvaa testijonoa pitää käsitellä kokonaisuutena, eli ajaa testitapaukset määrätyssä järjestyksessä.

Erilaisia testitapausten priorisointitekniikoita on olemassa monenlaisia. Artikkelissa (Elbaum ym., 2002) on listattu 16 erilaista priorisointitekniikkaa sekä kaksi tekniikoiden vertailua helpottamaan tehtyä tekniikkaa. Suurin osa testitapausten priorisointitekniikoista käyttää priorisoinnissa apuna tietoja testitapausten ajohistoriasta. On kuitenkin olemassa myös sellaisia priorisointitekniikoita, joita voidaan käyttää uusien testitapausten järjestämiseen. (Rothermel ym., 2001)

4.1.2 Testijoukon harventamistekniikat

Kun ohjelmaa kehitetään, täytyy sitä testaavaan testijoukkoon lisätä uusia testitapauksia. Testijoukon kasvaessa myös testijoukon ajaminen kestää kauemmin ja sen ylläpito on kalliimpaa. Testijoukosta voidaan usein löytää testitapauksia, jotka testaavat samoja ohjelman osia. Tällaisia testitapauksia voidaan poistaa testijoukosta ilman, että testijoukon kattavuus laskee. Tällaisten testitapausten etsimistä ja poistamista kutsutaan testijoukon harventamiseksi (test-suite reduction/minimization)

Testijoukon harventamisella voidaan vähentää testijoukon ajamis-, kelpoistamis- ja ylläpitokustannuksia. Vaikka testijoukon kattavuus ei laske, saattaa testijoukon kyky havaita virheitä vähentyä, koska samoja toimintoja testataan vähemmillä testitapauksilla. (Rothermel ym., 1998b)

Testitapausten välillä voi olla riippuvuussuhteita niin, että testitapaukset pitää ajaa tietyssä järjestyksessä oikeellisen tuloksen saamiseksi (Onoma ym., 1998). Tällaisesta

riippuvuussuhteita sisältävästä testijonosta ei saa poistaa testitapauksia, koska muuten lopputulos voi olla virheellinen.

Testijoukon harventamisen tuomista hyödyistä on tehty joitakin tutkimuksia. Wongin tutkimuksessa (Wong ym., 1995) tarkasteltiin kymmentä C-kielistä ohjelmaa, joista tutkimukseen tehtiin yhteensä 183 virheitä sisältävää versiota. Testijoukkojen harventamiseen käytettiin ATACMIN-työkalua (Horgan ym., 1992). Tutkimuksesta saadut tulokset antavat ymmärtää, ettei testijoukon harventaminen alenna testijoukon kykyä havaita virheitä merkittävästi.

Toisessa tutkimuksessa (Rothermel ym., 1998b) tutkittiin joukkoa yhden virheen sisältämiä ohjelmaversioita Harrold-Gupta-Soffa -harventamistekniikkaa apuna käyttäen. Tässä tutkimuksessa saatiin edellisestä tutkimuksesta poikkeavia tuloksia koskien harventamisen vaikutusta testijoukon kykyyn havaita virheitä. Tutkimustulosten mukaan testijoukon kyky havaita virheitä aleni huomattavasti testijoukon harventamisen seurauksena.

Molemmissa tutkimuksissa havaittiin, että testijoukon kasvaminen huonontaa virheiden havaitsemisen tehokkuutta ja että testijoukon harvennuksella saatiin pienennettyä testijoukon kokoa. Yksi mahdollinen syy tutkimustulosten eroavaisuuksiin voi piillä tutkittujen testijoukkojen eroavaisuuksissa. Koska testijoukkojen virheiden havaitsemiskykyyn vaikuttavat muutkin asiat (kappale 4.3) kuin vain testijoukon koko, testijoukkoja ei kannata harventaa pelkästään koodikattavuuden perusteella. (Harrold, 1999)

Testijoukon harventamiseen liittyen on pohdittu jakostrategioiden (dividing strategies) käyttöä. Jonesin työryhmän tekemässä tutkimuksessa (Jones ym., 2003) tutkittiin erityisen vaativan modified condition/decision coverage (MC/DC)-kattavuuskriteerin täyttävän testijoukon harventamista. Blackin työryhmän tekemässä tutkimuksessa (Black ym., 2004) tutkittiin, kuinka testijoukko voitaisiin minimoida niin, että testijoukon kyky löytää virheitä ei dramaattisesti laskisi.

4.2 Regressiotestien valintatekniikat

4.2.1 Yleistä asiaa valintatekniikoista

Ohjelman mahdollisen taantumisen huomaamiseksi voidaan ohjelma uudelleentestata kaikilla muutosta edeltävillä testitapauksilla, mutta tämä on aikaa vievää ja tehotonta. Käyttämällä regressiotestien valintatekniikoita ohjelmaan tehtyä muutosta edeltäneestä testijoukosta poimitaan joukko testitapauksia käytettäväksi muutetun ohjelman testaamiseen, ja näin rajoitetaan testitapausten määrää.

Tähän mennessä ei ole keksitty sellaista regressiotestien valintatekniikkaa, joka olisi muita parempi kaikissa mahdollisissa tilanteissa. Mitä kattavampi regressiotestijoukosta pyritään tekemään, sitä enemmän aikaa kuluu testitapausten valikoimiseen (Gao ym., 2003, s. 214). Valintatekniikka on epäkäytännöllinen, jos regressiotestijoukon valintaan, ajamiseen ja tulosten kelpoistamiseen kuluu yhteensä enemmän aikaa kuin alkuperäisen testijoukon uudelleenajamiseen kokonaisuudessaan (Graves ym., 2001, s. 188).

Tässä tutkielmassa kuvataan tarkemmin valintatekniikoita, jotka valitsevat testitapaukset analysoimalla itse ohjelmaa. Muunlaisista valintatekniikoista kerrotaan luvussa 4.2.6. Ohjelmaa analysoivat tekniikat pyrkivät yleensä tunnistamaan ohjelman *vaaralliset entiteetit*, eli sellaiset ohjelman osat, joihin ohjelmaan tehty muutos on voinut vaikuttaa, ja jotka näin ollen voivat toimia poikkeavasti muutetussa ohjelmaversiossa.

Vaaralliset entiteetit voidaan tunnistaa joko *staattisesti* tai *dynaamisesti*. Staattisessa lähestymistavassa vaaralliset entiteetit löydetään analysoimalla itse ohjelmaa. Dynaamisessa lähestymistavassa analysoidaan ohjelman sijaan kunkin testitapauksen testihistoriaa. Staattiset lähestymistavat ovat todennäköisesti tehokkaampia, koska ne käyvät ohjelman läpi vain kerran. Dynaamiset lähestymistavat joutuvat käymään läpi jokaisen testin testihistorian ja ovat näin todennäköisesti staattisia lähestymistapoja hitaampia, mutta samalla myös tarkempia (Gao ym., 2003).

Ohjelmaa analysoivat tekniikat poikkeavat toisistaan siinä, miten ne tunnistavat vaaralliset entiteetit, ja miten perinpohjaisesti vaarallisia entiteettejä testaavan regressiotestijoukon ne valitsevat.

Ohjelman vaaralliset entiteetit voidaan löytää tutkimalla joko ohjelman koodia tai määrityksiä. Regressiotestien valintatekniikoiden tutkimus on aiemmin keskittynyt pääosin *koodiin pohjautuviin valintatekniikoihin* (luku 4.2.4), mutta viime vuosikymmenen lopulla on alettu yhä enemmän tutkia *määrityksiin pohjautuvia valintatekniikoita* (luku 4.2.5).

Vaarallisten entiteettien tunnistamisen lisäksi valintatekniikoiden tehtävä on päättää, miten perinpohjaisesti löydetyt ohjelman osat pitäisi testata. Regressiotestien valintatekniikat voidaan jakaa kolmeen ryhmään sen perusteella, miten ne päättävät riittävästä testauksesta (Gao ym., 2003):

Minimointitekniikat (minimization techniques). Minimointiin perustuvien regressiotestien valintatekniikat luovat kaikki ohjelman vaaralliset entiteetit kattavan testijoukon mahdollisimman vähillä testitapauksilla. Minimointitekniikoita ei ole suunniteltu turvallisiksi, eli ne saattavat jättää regressiotestijoukosta pois sellaisia alkuperäisen testijoukon testitapauksia, jotka paljastaisivat virheen muutetussa ohjelmassa.

Kattavuuteen perustuvat tekniikat (coverage-based techniques) (mm. Bates ym., 1993; Binkley, 1995). Tämäntyyppiset tekniikat valitsevat regressiotestitapaukset jonkin kattavuuskriteerin perusteella. Kattavuuteen perustuvat tekniikat valitsevat regressiotestijoukkoon enemmän testitapauksia kuin minimointitekniikat, mikä nostaa regressiotestauksessa tarvittavaa aikaa. Toisaalta kattavuuteen perustuvat tekniikat ovat yleensä myös minimointitekniikoita inklusiivisempiä, eli ne valitsevat regressiotestijoukkoon enemmän virheitä mahdollisesti paljastavia testitapauksia. Kattavuuteen perustuvilla tekniikoilla valitun regressiotestijoukon ominaisuudet vaihtelevat hyvin paljon sen mukaan, millaisia kattavuuskriteerejä valintatekniikassa käytetään. Kattavuuteen perustuvia tekniikoita ei ole suunniteltu turvallisiksi, eli ne saattavat jättää regressiotestijoukosta pois sellaisia alkuperäisen testijoukon testitapauksia, jotka paljastaisivat virheen muutetussa ohjelmassa.

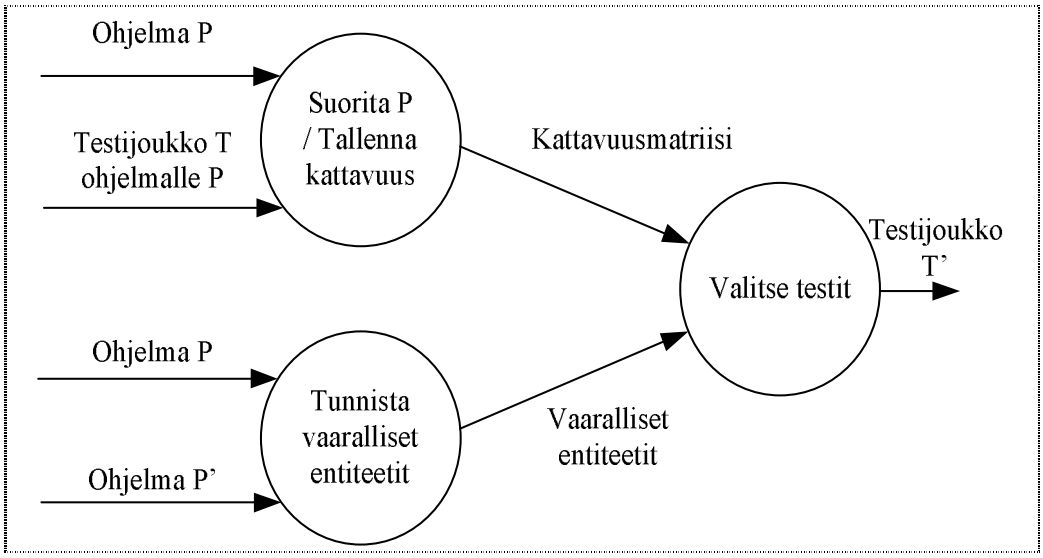
Turvalliset tekniikat (safe techniques) (mm. Chen ym., 1994; Rothermel ym., 1997). Turvalliset valintatekniikat on suunniteltu niin, että ne valitsevat alkuperäisestä testijoukosta kaikki sellaiset testit, jotka voivat mahdollisesti paljastaa virheen muutetussa ohjelmassa. (Graves ym., 2001)

Testitapausten välillä voi olla riippuvuussuhteita niin, että testitapaukset pitää ajaa tietyssä järjestyksessä oikeellisen tuloksen saamiseksi. Mikäli jokin tällaisen riippuvuuksia sisältävän testijonon testitapauksista valitaan regressiotestijoukkoon, täytyy regressiotestijoukkoon lisätä testijono kokonaisuudessaan. (Onoma ym., 1998)

Yleinen analysointiin perustuva malli regressiotestien valintaan esitetään luvussa 4.2.2. Luvussa 4.2.3 esitellään kriteerit, joilla regressiotestien valintatekniikoita voidaan vertailla. Koodiin pohjautuvia regressiotestien valintatekniikoita käsitellään luvussa 4.2.4. Määrityksiin pohjautuvia regressiotestien valintatekniikoita käsitellään luvussa 4.2.5. Muita valintatekniikoita, jota eivät perustu ohjelman vaarallisten entiteettien tunnistamiseen käsitellään luvussa 4.2.6. Gravesin tekemä tutkimus valintatekniikoiden eroista on tutkielman liitteenä (liite B).

4.2.2 Ohjelman analysointiin perustuva malli regressiotestien valintaan

Olkoon ohjelma **P'** muutettu versio ohjelmasta **P** ja olkoon **T** ohjelman **P** testaamiseksi kehitetty testijoukko. **P'**:n regressiotestauksen aikana on käytössä testijoukko **T** ja tietoa ohjelman **P** testauksesta testijoukolla **T**. Kun tutkitaan testijoukon **T** uudelleenkäyttöä ohjelman **P'** testaamiseen, esiin nousee kaksi ongelmaa: Ensinnäkin, mitä testijoukon **T** sisältämiä testitapauksia kannattaa käyttää ohjelman **P'** testaamiseen. Toiseksi, mitä uusia testitapauksia pitää kehittää **P'** sisältämän uuden toiminnallisuuden testaamiseen. Regressiotestien valintatekniikat keskittyvät ensimmäiseen ongelmaan, eli ohjelman **P'** testaamiseen tarkoitetun testijoukon **T'** kokoamiseen testijoukon **T** sisältämistä testitapauksista.



Kuva 3. Ohjelman analysointiin perustuva malli regressiotestien valitsemiseksi (Harrold ym., 2001)

Kuvassa 3 on esitetty ohjelman analysointiin perustuva malli regressiotestien valitsemiseksi. Ensin ohjelmaa **P** testataan testijoukolla **T**, jolloin tekniikka kerää tavanomaisten tietojen (testi läpi tai ei läpi) lisäksi tiedot jokaisen testitapauksen kattavuudesta ohjelmassa. Tämän lisäksi tekniikka vertailee ohjelmia **P** ja **P'** ja etsii **P**:stä vaaralliset entiteetit, eli sellaiset ohjelman osat, joihin tehty muutos on voinut vaikuttaa. Tekniikka jättää pois regressiotestijoukosta sellaiset testitapaukset, jotka eivät kata vaarallisia entiteettejä, eivätkä näin ollen pysty paljastamaan uusia virheitä. (Harrold ym., 2001)

Koodipohjaisissa valintatekniikoissa kattavuusmatriisi ja vaaralliset entiteetit etsitään ohjelman koodia analysoimalla. Määrityspohjaisissa valintatekniikoissa tarkastellaan koodin sijaan ohjelman määrittelyjä.

4.2.3 Kriteerit valintatekniikoiden vertailemiseksi

Usein käytetty tapa tarkastella valintatekniikoiden eroja on käyttää Rothermelin artikkelissa (Rothermel ym., 1996) esitettyjä kriteerejä:

- *Inklusiivisuus* (inclusiveness) kertoo, kuinka hyvin valintatekniikka osaa valita alkuperäisestä testijoukosta virheitä mahdollisesti paljastavia testejä mukaan regressiotestijoukkoon. Turvallinen valintatekniikka ottaa mukaan kaikki sellaiset alkuperäisen testijoukon testit, jotka voivat paljastaa virheen.

- *Tarkkuus* (precision) kertoo kuinka hyvin valintatekniikka osaa seuloa pois sellaiset testitapaukset, jotka eivät voi paljastaa virheitä. 100-prosenttisen tarkkuuden saavuttava valintatekniikka ei ota testijoukkoon mukaan yhtään turhaa testiä.
- *Tehokkuus* (efficiency) mittaa valintatekniikan tila- ja aikavaatimuksia. Tehokkuutta mitattaessa regressiotestauksen elinkaaresta voidaan erottaa kaksi erilaista vaihetta: *alustava vaihe* (preliminary phase) ja *kriittinen vaihe* (critical phase). Alustavassa vaiheessa ohjelmoijat työskentelevät koodin parissa. Testaajat voivat tässä vaiheessa testata joitakin ohjelman osia, lähinnä sellaisia joihin ei todennäköisesti enää tehdä muutoksia. Kriittisessä vaiheessa ohjelmakoodi on jäädytetty niin, että sitä ei voi muuttaa. Tässä vaiheessa ohjelmaa pyritään testaamaan perusteellisesti käytettävän ajan puitteissa. Valintatekniikoiden tehokkuutta vertaillessa kiinnitetään enemmän huomiota tekniikan tehokkuuteen kriittisen vaiheen aikana.
- *Yleiskäyttöisyys* (generality) kertoo, kuinka useassa tilanteessa valintatekniikan käyttö on mahdollista.

Valintatekniikan yleiskäyttöisyyden arvioiminen on vaikeaa, joten yleensä valintatekniikoita tarkastellaan kolmen ensimmäisen kriteerin pohjalta.

4.2.4 Koodiin pohjautuvat valintatekniikat

Koodiin pohjautuvat valintatekniikat pyrkivät tunnistamaan ohjelman vaaralliset entiteetit analysoimalla ohjelman koodia.

Vaarallisten entiteettien tunnistamiseksi on kehitetty useita lähestymistapoja. Esimerkiksi *palomuuritekniikat* (firewall approach) tunnistavat ohjelman vaaralliset lauseet ja funktiot, kun taas *tietovirtatekniikat* (data-flow approach) ja *ohjelman viipalointitekniikat* (program-slicing approach) tunnistavat ohjelmasta vaaralliset määrittely-käyttö -parit (definition-use pairs). Määrittely-käyttö -parilla tarkoitetaan, että tietty muuttuja on ensin määritelty ohjelmassa ja myöhemmin kyseistä muuttujaa on käytetty ohjelmassa. Muuttujan määrittely-käyttö -parin välissä saa olla (ja yleensä onkin) muita lauseita,

mutta välissä olevat lauseet eivät saa käsitellä kyseistä muuttujaa millään tavalla. (Graves ym., 2001)

4.2.5 Määrityksiin pohjautuvat valintatekniikat

Koodiin pohjautuvat regressiotestien valintatekniikat soveltuvat hyvin yksikkötestaukseen, mutta testattavan komponentin koon ja monimutkaisuuden kasvaessa sekä ohjelmasta kerättävien tietojen määrän kasvaessa koodiin pohjautuvien valintatekniikoiden käyttö vaikeutuu. Toinen ongelma on koodin saatavuus ja sen ymmärtäminen. Koodiin pohjautuvat valintatekniikat ovat myös kieliriippuvaisia, minkä johdosta saatetaan joutua käyttämään useita erilaisia koodiin pohjautuvia valintatekniikoita. Jos testattavan komponentin koodi ei syystä tai toisesta ole saatavissa, on koodipohjaisten valintatekniikoiden käyttäminen mahdotonta. (Chen ym., 2002)

Määrityksiin pohjautuvat valintatekniikat eivät ole aina yhtä tarkkoja kuin koodiin pohjautuvat valintatekniikat. Ne valitsevat turhia testitapauksia, jotka eivät voi paljastaa regressiovirheitä muutetussa ohjelmassa. Lisäksi määrityksiin pohjautuvat regressiotestien valintatekniikat edellyttävät, että ohjelman määritykset ovat ajantasaiset ja huolellisesti tehdyt. Tämä ei ole aina helppoa toteuttaa käytännössä, koska eri komponenttien väliset suhteet ovat hyvin monimutkaisia ja kaikkia ohjelmaan tehtyjä muutoksia ei kirjata asianmukaisesti. (Briand ym., 2002; Chen ym., 2002) Luvussa 5 kerrotaan, kuinka regressiotestit voidaan valita UML-määrityksiin perustuen.

4.2.6 Muut regressiotestien valintatekniikat

Tässä luvussa käsitellään sellaisia yleisesti käytössä olevia regressiotestien valintatekniikoita, jotka eivät analysoi testattavan ohjelman koodia tai määrityksiä.

Eräs tapa valita testit on ottaa alkuperäisestä testijoukosta tietty määrä satunnaisesti valittuja testitapauksia regressiotestaukseen. Tällaista tekniikkaa kutsutaan *satunnaistekniikaksi* (random) ja sen yhteyteen merkitään yleensä, kuinka paljon alkuperäisestä testijoukosta valitaan regressiotestijoukkoon. Esimerkiksi 50% alkuperäisen testijoukon testitapauksista valitsevaa tekniikkaa kutsutaan random50-tekniikaksi. Gravesin valintatekniikoista tekemissä tutkimuksessa (Graves ym., 2001) huomattiin satunnaistekniikoiden olevan keskimäärin hyvin inklusiivisiä (Random25-tekniikalla inklusiivisuusmediaani oli 88%), eli ne valitsivat regressiotestijoukkoon paljon virheitä paljastavia testejä.

Kun testitapausten valintaprosenttia kasvatettiin (random50 ja random75), myös inklusiivisuus kasvoi, mutta hitaammin. Random-tekniikat eivät ole turvallisia tekniikoita ja niiden inklusiivisuus voi vaihdella suuresti eri ohjelmien välillä.

Regressiotestien valintaan voidaan käyttää myös *riskipohjaisia tekniikoita*, jotka pyrkivät valitsemaan testitapaukset niin, että riskit minimoituvat. Riskillä tarkoitetaan tekijää, joka toteutuessaan aiheuttaa projektin kannalta ei-toivottuja seuraamuksia. Seuraavaksi esitetään eräs riskeihin pohjautuva malli, jonka avulla voidaan valita regressiotestauksessa käytettävät testitapaukset.

Chenin artikkelissaan (Chen ym., 2002) esittämä malli sisältää neljä vaihetta. Ensimmäisessä vaiheessa arvioidaan *testitapauksen hinta*. Hinnalla tarkoitetaan tuotteen käyttäjän ja toimittajan löytämistä virheistä aiheutuvia kustannuksia, kuten esimerkiksi markkinaosuuden laskua tai suurempia ylläpitokustannuksia. Arviointi tehdään kyselylomakkeilla, joissa kysytään esimerkiksi, kuinka usein käyttäjä käyttää testitapauksen testaamia ominaisuuksia ja kuinka monimutkaisesti testattavat osat on toteutettu. Testit jaetaan saadun pistemäärän mukaan viiteen eri hintaluokkaan 1-5, suurempi numero tarkoittaa suurempaa hinta-arviota. Jokaiseen hintaluokkaan sisällytetään yhtä paljon testejä (20%).

Toisessa vaiheessa lasketaan *testitapauksen vakavuustodennäköisyys* (severity probability). Myersin (Myers, 1979) mukaan, mitä enemmän virheitä on löydetty, sitä todennäköisemmin niitä löytyy jatkossakin. Näin ollen paljon virheitä löytäneet testitapaukset todennäköisesti löytävät paljon virheitä jatkossakin. Testitapausten testihistoriasta voidaan löytää tiedot, kuinka monta virhettä testitapaus löysi edellisellä kierroksella. Tämän perusteella saadaan testitapauksille virheen löytämistodennäköisyys, eli montako virhettä testitapaus on löytänyt. Löydetyille virheille on löytöhetkellä annettu vakavuusaste, eli kuinka kriittinen virhe on ohjelman toiminnan kannalta. Testitapauksen vakavuustodennäköisyys voidaan laskea kertomalla testitapauksen löytämien virheiden lukumäärä virheiden vakavuudella ja jakamalla saadun luvun perusteella viiteen luokkaan 1-5 niin, että jokaiseen luokkaan tulee yhtä paljon testitapauksia. Tarkastelun ulkopuolelle jätetään ne testitapaukset, jotka eivät ole löytäneet yhtään virhettä. Esimerkiksi jos testitapaus on löytänyt edellisellä kierroksella 4 virhettä, joiden vakavuusasteet ovat 2, 3, 3 ja 4, kerrotaan virheiden lukumäärä 4 virheiden keskimääräisellä vakavuu-

della 3 ((2+3+3+4)/4), jolloin saadaan lopputulokseksi 12 (4*3). Mihin viidestä vakavuustodennäköisyysluokasta kyseinen testitapaus sijoitetaan, tiedetään vasta, kun on laskettu muiden testitapausten vakavuustodennäköisyydet.

Kolmannessa vaiheessa lasketaan *testitapauksen riskialttius* (risk exposure) kertomalla jokaisen testitapauksen kohdalla kahdesta edellisestä vaiheesta saadut arvot (testitapauksen hinta ja -vakavuustodennäköisyys) keskenään. Tässä vaiheessa voidaan lukuarvoja painottaa, jos esimerkiksi tietynlaiset ohjelman ominaisuudet ovat tärkeämpiä kuin toiset.

Neljännessä vaiheessa valitaan regressiotestit käyttämällä apuna edellisestä vaiheesta saatuja tietoja. Regressiotestit valitaan niin, että saadaan riittävä varmuus ohjelman oikeellisesta toiminnasta. Tämä voidaan saavuttaa esimerkiksi valitsemalla 30 prosenttia alkuperäisen testijoukon testitapauksista. Näin ollen regressiotestijoukkoon valitaan suurimman riskialttiuden omaavia testitapauksia kunnes haluttu määrä testitapauksia on kasassa.

4.3 Metodologioiden tehokkuuteen vaikuttavat asiat

Regressiotestausta tehostavista tekniikoista tehdyissä tutkimuksissa on tekniikoiden tehokkuuden huomattu vaihtelevan paljon testausympäristön mukaan. Eri tekniikoiden tehokkuuteen vaikuttavien asioiden mahdollisimman tarkka selvittäminen on tärkeää, jotta eri tilanteissa osataan valita tilanteeseen sopivat tekniikat. Tärkeimmät tehokkuuteen vaikuttavat asiat ovat alkuperäisen testijoukon koostumus, ohjelmaan tehtyjen muutosten ominaisuudet sekä regressiotestattavan ohjelman ominaisuudet.

Alkuperäisen testijoukon koostumuksen vaikutusta valinta-, priorisointi- ja harventamistekniikoiden toimintaan on tutkittu tähän mennessä laajimmin artikkelissa (Rothermel ym., 2003) (luku 4.3.1). Tehdyn muutoksen ominaisuuksien vaikutusta valintatekniikoihin on tutkittu artikkeleissa (Kim ym., 2000) ja (Elbaum ym., 2003) (luku 4.3.2). Viimeksi mainitussa artikkelissa tutkimukseen on sisällytetty myös priorisointitekniikat. Testattavan ohjelman ominaisuuksien vaikutuksesta metodologioihin ei ole tähän mennessä tehty kattavaa tutkimusta.

4.3.1 Alkuperäisen testijoukon ominaisuuksien vaikutus eri metodologioihin

Valinta- priorisointi- ja harventamis-metodologioihin kuuluvien tekniikoiden kustannustehokkuus vaihtelee riippuen alkuperäisen testijoukon koostumuksesta. Varsinkin se, miten testijoukon sisältämien testitapausten syötteet on laadittu, vaikuttaa merkittävästi testijoukon koostumukseen ja näin ollen myös erilaisten tekniikoiden tehokkuuteen. Esimerkiksi tekstinkäsittelyohjelman testaamiseen tarkoitettu testijoukko voi sisältää pienen määrän testitapauksia, joista jokainen testaa suurta osaa ohjelman toiminnasta. Tällainen testitapaus voisi esimerkiksi avata dokumentin, käydä läpi satoja ohjelman eri toimintoja ja lopuksi sulkea dokumentin. Vaihtoehtoisesti testijoukko voi sisältää suuren määrän testitapauksia, joista jokainen testaa vain muutamaa ohjelman toimintoa. Testitapausten valinnalla on suuri vaikutus testauksen tehokkuuteen, mutta kirjat ja artikkelit tarjoavat erilaisia ja toisinaan ristiriitaisia neuvoja valintaan liittyen.

Tässä luvussa esitetään Rothermelin tekemä tutkimus (Rothermel ym., 2003) testijoukon rakeisuuden ja testisyötteiden ryhmittelyn vaikutuksesta eri metodologioihin. *Testijoukon rakeisuus* (test suite granularity) kertoo, kuinka rakeisia testijoukon sisältämät testitapaukset ovat. Testitapauksen rakeisuus on suuri, jos siinä on paljon erilaisia syötteitä ja/tai syötteiden koko on suuri. Lähdeartikkelin kirjoittajat ovat etsineet pienimmät mahdolliset testitapausten syötteet, joita testitapauksille voidaan antaa. Näitä kutsutaan *alkeistason syötteiksi*. Seuraavaksi kirjoittajat ovat koostaneet alkeistason syötteistä testitapauksia ja testitapauksista testijoukkoja. Testijoukon rakeisuutta ilmaistaan merkinnöillä G1, G2, G4, G8, G16, G32 ja G64, joissa G-kirjaimen perässä oleva numeroarvo kertoo, kuinka monesta alkeistason syötteestä testijoukon testitapaukset koostuvat. Näin ollen suuremman lukuarvon testijoukot sisältävät karkearakeisempia testitapauksia, eli testitapauksia, jotka sisältävät useita syötteitä.

Testisyötteiden ryhmittely (test input grouping) kertoo, missä määrin testijoukon testitapaukset sisältävät keskenään samankaltaisia alkeistason syötteitä. Artikkelin kirjoittajat ovat jakaneet löytämänsä alkeistason syötteet eri lokeroihin sen mukaan, mitä ohjelman toimintoa ne testaavat. Lokerossa olevista alkeistason syötteistä on sen jälkeen koostettu testijoukon rakeisuuden määrittämän kokoisia testitapauksia. Kun lokeron alkeissyötteet on koostettu rakeisuuden määrittämän kokoisiksi testitapauksiksi, voi

lokeroon jäädä jäljelle alkeistason syötteitä, joista ei saada koottua kokonaista testitapausta. Tällaiset ylijääneet syötteet yhdistetään muista lokeroista ylijääneiden syötteiden kanssa jonka jälkeen niistä koostetaan määritetyn rakeisuustason kokoisia testitapauksia.

Testisyötteiden ryhmittelyn sanotaan olevan korkea, mikäli suuri osa testitapauksista on homogeenisia, eli ne on koottu yhtä toimintoa testaavista alkeistason syötteistä. Testisyötteiden ryhmittelyasteen sanotaan olevan matala, mikäli suuri osa testijoukon testitapauksista on heterogeenisia, eli ne on jouduttu koostamaan monia eri toimintoja testaavista alkeistason syötteistä.

Rothermelin tutkimuksessa ei tarkastella eri metodologioiden hyödyllisyyttä pelkästään ohjelmiston rakennusvaiheen aikana, vaan huomio kiinnitetään myös metodologioiden hyödyllisyyteen ohjelmiston ylläpitoprosessin aikana. Pelkästään rakennusvaiheen aikainen tarkastelu johtaisi vääränlaisiin johtopäätöksiin, koska ohjelmistot yleensä käyvät elinkaarensa aikana useita testaussyklejä.

Testauskohteina Rothermel käytti kahta C-kielistä avoimen lähdekoodin ohjelmaa, joista molemmista tutkittiin kymmentä peräkkäistä versiota. *Emp-server* -ohjelma toimii palvelinohjelmana Empire-nimisessä verkkopelissä. *Bash* -ohjelma on komentorivikehote, jonka välityksellä pystyy käyttämään useita Unix-palveluja. Eri versioiden välillä ohjelmien kehittäjät olivat lisänneet niihin uutta toiminnallisuutta ja korjanneet virheitä. Koodirivien ja funktioiden lukumäärällä mitattuna Emp-server -ohjelma on ensimmäisten versioiden kohdalla Bash-ohjelmaa suurempi. Bash-ohjelmaan oli kuitenkin lisätty kehityksen aikana enemmän uutta toiminnallisuutta, koska viimeisten versioiden kohdalla Bash-ohjelma menee koodirivien ja funktioiden lukumäärällä mitattuna ohi Emp-Server -ohjelmasta.

Testaa kaikki uudelleen-tekniikka

Testaa kaikki uudelleen (retest all) -tekniikka käyttää uudelleen kaikki sellaiset alkuperäisen testijoukon testitapaukset, jotka eivät ole vanhentuneita. Tämä tekniikka ilmentää yleistä nykykäytäntöä testien valinnassa (Onoma ym., 1998) ja toimi Rothermelin tutkimuksen kontrollitekniikkana, johon muita verrattiin.

Rothermelin tutkimuksessaan saamista tuloksista voidaan nähdä, että tekniikan tehokkuutta on mahdollista parantaa käyttämällä karkearakeisia testijoukkoja. Esimerkiksi emp-server -ohjelman kohdalla testijoukon rakeisuuden ollessa tasoa G1, testitapausten läpikäyminen vei 365 minuuttia kauemmin kuin rakeisuustason ollessa tasoa G4. Testitapausten ajamiseen kuluva aika laski siis melkein neljäsosaan alkuperäisestä. Bash-ohjelman tapauksessa samanlainen rakeisuustason nostaminen johti 415 minuutin ajonaikaisiin säästöihin, jolloin testijoukon suoritusaika lähes puolittui. Rakeisuuden nostamisella ei ole suurta vaikutusta virheiden löytymisen todennäköisyyteen, koska rakeisuustason nostaminen esti virheen havaitsemisen vain viidessä testitapauksessa (Rothermel ym., 2003). Aiemmassa tutkimuksessa (Rothermel ym., 2002, s. 230-240) huomattiin karkearakeisempien testijoukkojen havaitsevan virheitä hienorakeisempia testijoukkoja tehokkaammin.

Tuloksia tarkastellessa pitää huomioida joitakin asioita. Mitä suuremmaksi rakeisuusastetta kasvatetaan, sitä vähemmän hyötyä saavutetaan. Parhaat tulokset saadaan aloittamalla rakeisuusasteen kasvattaminen alimmalta rakeisuustasolta, eli kaikkein hienorakeisimmista testisyötteistä.

Testijoukkojen rakeisuusasteen nostamisella saavutettava hyöty tulee testitapausten alku- ja loppuvalmisteluihin kuluvan ajan vähenemisestä. Mikäli testitapausten alku- ja loppuvalmisteluihin kuluva aika on alkuperäisessä testijoukossa vähäinen, pienenevät rakeisuusasteen nostamisella saavutettavat hyödyt. Muut pienirakeisuudella saavutettavat hyödyt voivat tällöin olla suurempia kuin karkearakeisuudesta saatavat aikasäästöt.

Myös ohjelman suoritusaika suhteessa syötteen kokoon vaikuttaa rakeisuusasteen nostamisen hyödyllisyyteen. Tutkimuksessa käytettyjen ohjelmien suoritusaika kasvoi lineaarisesti suhteessa syötteen kokoon, joten testitapausten yhdistäminen tuotti hyviä tuloksia. Mikäli ohjelman suoritusaika kasvaa nopeasti suhteessa syötteen kokoon, voi karkearakeisemman testitapauksen ajaminen viedä kauemmin kuin usean hienorakeisemman testitapauksen. (Rothermel ym., 2003)

Regressiotestien valintatekniikat

Regressiotestien valintatekniikoista Rothermel otti tutkimukseensa mukaan kolme tekniikkaa:

- *Muutettu entiteetti* (modified entity) -tekniikka on turvallinen valintatekniikka, joka valitsee regressiotestijoukkoon mukaan sellaisia funktioita testaavat testitapaukset, jotka käyttävät muutettuja tai poistettuja muuttujia tai tietorakenteita. Lisäksi jos funktioon itseensä on tullut muutoksia tai se on poistettu muutetusta ohjelmaversiosta, funktiota testaavat testitapaukset valitaan mukaan regressiotestijoukkoon.
- *Muutettu ei-ydin entiteetti* (modified non-core entity) -tekniikka on muuten kuin muutettu entiteetti -tekniikka, mutta se jättää pois tarkastelusta sellaiset funktiot, jotka ovat usean testitapauksen testaamia. Voidaan esimerkiksi määrittää, että tarkastelusta jätetään pois sellaiset funktiot, joita yli 80-prosenttia testijoukon sisältämistä testitapauksista testaa.
- *Minimointitekniikka* (minimization technique) pyrkii valitsemaan vähimmäismäärän testitapauksia niin, että ohjelman jokainen mahdollisesti muuttunut entiteetti on ainakin yhden testitapauksen testaama.

Turvalliset regressiotestien valintatekniikat eivät odotetusti jättäneet testijoukosta pois sellaisia testejä, jotka voisivat paljastaa regressiovirheen. Vaikka toisissa tutkimuksissa on huomattu turvallisten valintatekniikoiden vähentävän testitapausten määrää, tässä tutkimuksessa turvalliset valintatekniikat eivät poistaneet yhtäkään testitapausta, eivätkä näin ollen tuoneet säästöjä. Tästä johtuen Rothermel keskittyi tutkimuksessaan ei-turvallisiin regressiotestien valintatekniikoihin, joista tutkimuksessa tarkasteltiin kahta: muutettu ei-ydin entiteetti -valintatekniikkaa ja minimointitekniikkaa, joka on edellä mainittua aggressiivisempi.

Testituloksissa huomattiin valintatekniikoiden tarjoavan paljon parempia tuloksia, kun niitä sovellettiin hienorakeisempiin testijoukkoihin karkearakeisten testijoukkojen sijasta. Esimerkiksi kun muutettu ei-ydin entiteetti -tekniikkaa sovellettiin emp-server ohjelmalle tehtyyn G1-rakeisuustason testijoukkoon, testijoukon suorittamisaika väheni 505 minuutista 181 minuuttiin, tarjoten täten 64-prosentin ajansäästön. Kun samaa tek-

niikkaa sovellettiin G64-rakeisuustason testijoukkoon, saavutettiin vain 9-prosentin ajansäästö. Tulos johtuu siitä, että hienorakeisemmat testijoukot ovat karkearakeisia testijoukkoja joustavampia, eli ne tarjoavat valintatekniikoille mahdollisuuden valita käytettävät testitapaukset tarkemmin.

Rothermel päättelee saamista tutkimustuloksistaan, että käyttämällä muutettu ei-ydin entiteetti -tekniikkaa hienorakeiseen testijoukkoon, voidaan saavuttaa aikasäästöjä verrattuna karkearakeisemmalle testijoukolle suoritettuun testaa kaikki uudelleen -tekniikkaan. Muutettu ei-ydin entiteetti -tekniikka ei ole turvallinen, joten jotkin virheet voivat jäädä huomaamatta.

Aggressiiviset regressiotestien valintatekniikat (esim. minimointitekniikka) voivat tarjota suuria aikasäästöjä, mutta toisaalta myös testijoukon kyky huomata virheitä vähenee huomattavasti. Ensimmäisessä tutkituista ohjelmista (emp-server), rakeisuuden lisääminen johti aikasäästöihin, mutta toisessa ohjelmassa (bash) rakeisuuden lisääminen lisäsi testijoukon ajoon kuluvaa aikaa. Testijoukon kyvyssä huomata virheitä ei huomattu laskua.

Testisyötteiden ryhmittelyllä Rothermel ei huomannut olevan merkittävää vaikutusta suoritusaikaan ja testijoukon kykyyn löytää virheitä. Testituloksista voidaan päätellä myös, että karkearakeisia testijoukkoja käyttämällä löydetään paremmin helposti löydettäviä virheitä, mutta vaikeasti löydettävien virheiden löytäminen on tällöin vaikeampaa.

Testitapausten priorisointitekniikat

Testitapausten priorisointi-metodologiasta Rothermel valitsi tutkimukseensa kolme tekniikkaa:

- *Täydentävä funktiokattavuus* (additional function coverage) -tekniikka valitsee iteratiivisesti funktiokattavuudeltaan suurimman testitapauksen ja laskee testitapauksen valittuaan jäljelle jääneiden testitapausten kattavuuden vielä kattamattomista funktioista. Tätä jatketaan, kunnes kaikki ohjelman funktiot ovat ainakin yhden testitapauksen kattamia.
- *Täydentävä muutettujen funktioiden kattavuus* (additional modified-function coverage) -tekniikka toimii kuten edellä mainittu tekniikka, mutta se valitsee aluksi testitapaukset niiden *muutettujen* funktioiden kattavuuden perusteella. Kun

kaikki muutettuja funktioita testaavat testitapaukset on järjestetty, järjestetään loput testitapaukset käyttämällä täydentävä funktiokattavuus -tekniikkaa.

- *Optimaalinen priorisointi* (optimal prioritization) -tekniikka käyttää hyväkseen tietoa eri testitapausten kyvystä paljastaa virheitä. Koska virheitä paljastavia testitapauksia ei voida ennen testausta tietää, tätä tekniikkaa ei voida soveltaa käytännössä. Tekniikka toimii kuitenkin hyvänä ylärajana priorisoinnin hyötyjä tarkasteltaessa

Testitapausten priorisointi -metodologiaa voidaan käyttää rinnakkain muiden metodologioiden kanssa. Rothermel huomasi tutkimuksessaan testitapausten priorisoinnin olevan tehokkaampaa hienorakeisiin testijoukkoihin sovellettuna. Tämä antaa syyn suosia hienorakeisia testijoukkoja karkearakeisten testijoukkojen sijaan. Käytettäessä testaa kaikki uudelleen -tekniikkaa, täytyy testaajien löytää sopiva tasapaino testijoukon rakeisuuteen liittyen.

Rakeisuuden kasvattamisesta johtuva priorisointitekniikan tehokkuuden laskun jyrkkyys on riippuvainen testijoukon virheitä paljastavien testitapausten määrästä. Paljon virheitä löytäviä testitapauksia sisältävissä testijoukoissa rakeisuuden kasvattamisesta johtuva tehokkuuden lasku on pienempi, kuin vähän virheitä paljastavia testitapauksia sisältävän testijoukon kohdalla

Testijoukon harventamistekniikat

Testijoukkojen harventaminen -metodologiasta Rothermel otti tutkimukseensa mukaan kaksi tekniikkaa:

- *Ei harventamista*. Ilmentää yleistä toimintamallia ja toimii tutkimuksen kontrollitekniikkana.
- *GHS -harventamistekniikka* (Harrold ym., 1993) pyrkii rakentamaan testijoukon, joka täyttää määritetyt kattavuuskriteerit. Tässä tutkimuksessa kattavuuskriteerinä käytettiin funktiokattavuutta.

GHS-testijoukon harventamistekniikka ja minimointiin pyrkivä regressiotestien valintatekniikka ovat testitapausten valikoinnissa periaatteiltaan samanlaisia, eli mo-

lemmat pyrkivät kattamaan ohjelman osat mahdollisimman vähillä testitapauksilla. Ero on siinä, että minimointitekniikka laskee kattavuuden ohjelman muutetuista entiteeteistä, kun taas testijoukon harventamistekniikka laskee kattavuuden kaikista ohjelman entiteeteistä, jotka tarkasteltava testijoukko kattaa.

GHS-harventamistekniikkaa käytettäessä testijoukon rakeisuuden vaikutukset olivat lähes samanlaiset kuin regressiotestien valintatekniikoista minimointitekniikan kohdalla. Rakeisuuden vaikutukset testijoukon suoritusaikaan olivat eri ohjelmien kohdalla jonkin verran poikkeavia: emp-server -ohjelman tapauksessa testijoukon suoritusaika laski, kun taas bash-ohjelman tapauksessa testijoukon läpiviemisaika nousi. Virheiden löytymisen todennäköisyys laski bash-ohjelman kohdalla enemmän kuin emp-server -ohjelman kohdalla.

Yksi GHS-harventamistekniikan eroista minimointitekniikan tuloksiin verrattuna, oli testisyötteiden ryhmittelyn merkittävä vaikutus testijoukon kykyyn havaita virheitä. Emp-server -ohjelman kohdalla samankaltaisten testisyötteiden sijoittaminen samaan testitapaukseen helpotti virheiden löytymistä merkittävästi verrattuna siihen, että testisyötteet olisi sijoitettu satunnaisesti eri testitapauksiin. Bash-ohjelman tapauksessa ei testisyötteiden ryhmittelyllä ollut vaikutusta suuntaan tai toiseen. Rothermel olettaa eron johtuvan siitä, että bash-ohjelman kohdalla tutkimuksessa tehdyt erilaiset harvennetut testijoukot erosivat suuresti toisistaan tarkasteltaessa niiden kykyä löytää virheitä. Emp-server -ohjelman tapauksessa vaihtelevuus harvennettujen testijoukkojen kyvyssä löytää virheitä oli pienempi. Rothermelin tutkimuksessaan saamista tuloksista voidaan täten päätellä testisyötteiden ryhmittelyllä olevan merkitystä joissain tapauksissa.

Yhteenveto tutkimuksesta

Kuten tieteellisissä tutkimuksissa aina, täytyy näitäkin tutkimustuloksia tarkasteltaessa ottaa huomioon joitakin tulosten yleistettävyyteen vaikuttavia asioita. Tutkimuskohteina Rothermelin tutkimuksessa käytettiin vain kahta erilaista ohjelmaa, mikä heikentää tulosten yleistettävyyttä. Toisaalta ohjelmista saadut tulokset olivat verrattain yhteneviä. Yleistettävyyttä parantaa myös se, että tutkimukseen valittiin suhteellisen suuri otos todellisia, perättäin julkaistuja versioita tutkituista ohjelmista. Kooltaan tutkitut ohjel-

mat olivat samantapaisia muiden yleisessä käytössä olevien ohjelmien kanssa. (Rothermel ym., 2003)

Rothermelin tutkimuksessa ohjelmiin tehdyistä virheistä pyrittiin tekemään mahdollisimman edustavia, mutta tutkijat myöntävät lisätutkimusten olevan tarpeen selvitettäessä erilaisten virhemallien vaikutusta tutkimustuloksiin. Testausprosessi simuloi ohjelmistoyrityksissä käytössä olevia testausmalleja. Tulosten yleistettävyyttä voitaisiin parantaa suorittamalla tutkimus ohjelmistoyrityksissä oikeasti käytössä oleville testijoukoille.

Tutkimuksessa ei oteta huomioon testijoukkojen suorittamisesta, tarkastamisesta ja ylläpidosta tulevia kuluja. Myöskään virheiden paikantamisesta tulevia kuluja ei huomioida. Tutkimuksessa ei myöskään huomioida testitapausten valintaan, priorisointiin ja harventamiseen kuluvaa aikaa. Toisissa tutkimuksissa (Rothermel ym., 1998a; Rothermel ym., 1998b) on huomattu analysointiin vaadittavan ajan olevan suhteellisen pieni verrattuna testien ajamiseen vaadittavaan aikaan, ja että analysointiprosessi on mahdollista automatisoida ja näin ollen suorittaa ei-kriittisenä aikana.

Rothermel päätteli saatujen tutkimustulosten perusteella testijoukon rakeisuudella olevan merkittävä vaikutus testaa kaikki uudelleen-, regressiotestien valinta- sekä regressiotestien harventaminen-metodologioiden aikatehokkuuteen. Tämä tulos saatiin muutettu entiteetti -tekniikkaa lukuun ottamatta kaikissa tapauksissa ja tulos oli yhdenmukainen testiohjelmien kesken. Rakeisuudella oli merkittävä vaikutus myös priorisointitekniikoiden virheiden löytymistehokkuuteen; tulos oli yhdenmukainen eri testiohjelmien kesken. Testijoukon rakeisuudella huomattiin olevan merkittävä vaikutus virheiden löytymiseen, mutta tämä tulos saatiin vain käyttämällä muutettu ei-ydin entiteetti -valintatekniikkaa ja GHS-harventamistekniikkaa emp-server -ohjelmaan.

Testisyötteiden ryhmittelyllä huomattiin olevan merkitystä testaa kaikki uudelleen-, regressiotestien valinta- ja regressiotestien harventamistekniikoiden kohdalla vain yhdessä tapauksessa, eli silloin kun GHS-harventamistekniikkaa sovellettiin emp-server -ohjelmaan. Testisyötteiden ryhmittelyllä oli vaikutusta myös toisen priorisointitekniikan (täydentävä muutettujen funktioiden kattavuus) tehokkuuteen.

4.3.2 Ohjelmaan tehdyn muutoksen ominaisuuksien vaikutus metodologioihin

Kim ja Elbaum tarkastelivat tutkimuksissaan (Kim ym., 2000; Elbaum ym., 2003) ohjelmaan tehdyn muutoksen vaikutusta eri lähtökohdista. Kim keskittyi tutkimuksessaan tarkastelemaan ohjelmaan tehtyjen muutosten lukumäärän vaikutusta valintatekniikoiden toimintaan. Elbaum keskittyi tutkimaan tarkemmin muutosten sijainnin ja koon vaikutusta valintatekniikoiden ja priorisointitekniikoiden toimintaan. Tässä luvussa tarkastellaan ensin Elbaumin tutkimustuloksia, jonka jälkeen siirrytään tarkastelemaan Kimin tutkimustuloksia.

Regressiotestien valinta- ja priorisointitekniikoiden tehokkuus riippuu monesta eri asiasta. Näiden asioiden selvittäminen ja ymmärtäminen on tärkeää, jotta regressiotestaus voitaisiin suorittaa mahdollisimman tehokkaasti. Artikkelissa (Elbaum ym., 2003) on tarkasteltu testattavaan ohjelmaan tehtyjen muutosten vaikutusta eri regressiotestaustekniikoiden toimintaan. Kyseisessä tutkimuksessa huomattiin ohjelmaan tehdyn muutoksen ominaisuuksien vaikuttavan merkittävästi regressiotestaustekniikoiden suorituskykyyn.

Muutosten levinneisyydellä ohjelman sisällä ja yksittäisten muutosten suuruus vaikutti muutetun ei-ydin funktio -tekniikan kokoaman testijoukon kokoon, mutta mainituilla asioilla ei näyttänyt olevan vaikutusta kummankaan minimointitekniikan valitseman testijoukon kokoon. Pienet, laajalle alueelle jakautuneet ja harvojen testitapausten kattamat muutokset nostivat minimointitekniikoiden virheidenlöytämistehokkuutta. Muutoksiin keskittyvä minimointitekniikka kokosi jatkuvasti huonommin virheitä löytäviä testijoukkoja, kuin kattavuuteen keskittyvä minimointitekniikka. Mikäli muutokset tehdään usean testitapauksen kattamiin funktioihin, minimointitekniikat tuottavat virheenlöytämiskyvyltään vaikeasti ennustettavia testijoukkoja.

Täydentävä funktiokattavuus -priorisointitekniikka oli tehokas joka tilanteessa, kun taas täysi funktiokattavuus -priorisointitekniikan tehokkuus oli usein vaikeasti ennustettavaa. Kahden ohjelman kohdalla täydentävä funktiokattavuus suoriutui lähes yhtä hyvin kuin optimaalinen priorisointitekniikka ja suoriutui muidenkin ohjelmien kohdalla muita tutkimukseen valittuja priorisointitekniikoita paremmin. Täysi funktiokattavuus -

tekniikan tehokkuutta oli usein vaikea ennustaa. Yhden ohjelman kohdalla tekniikka suoriutui välillä erinomaisesti ja välillä huonosti, mutta muiden kolmen ohjelman kohdalla tekniikka järjesteli testitapaukset jopa huonommin kuin satunnaistekniikka. Kun ohjelmaan tehdyt muutokset olivat hajanaisia, tehostui täydentävä funktiokattavuus -tekniikan toiminta, mutta täysi funktiokattavuus -tekniikan tehokkuus väheni entisestään.

Muutettu ei-ydin -tekniikka valitsi pieniä ja virheenhavaitsemiskyvyltään tehokkaita testijoukkoja, kun ohjelman muutokset oli tehty pienelle alalle ja ne olivat usean testitapauksen kattamia. Kun ohjelmaan tehdyt muutokset levitettiin ympäri ohjelmaa, kyseinen tekniikka valitsi usein lähes kaikki testitapaukset ja näin ollen tekniikan tuomat hyödyt vähenivät suuresti. (Elbaum ym., 2003)

Kimin työryhmä tutki tutkimuksessaan (Kim ym., 2000) ohjelmaan tehtyjen muutosten lukumäärän vaikutusta erilaisten regressiotestien valintatekniikoiden kustannustehokkuuteen. Valintatekniikoita arvioitiin niiden valitsemien testijoukkojen koon perusteella ja miten hyvin ne osasivat valita regressiotestijoukkoon virheitä mahdollisesti paljastavat testitapaukset. Säästöt syntyvät ajettavien testitapausten vähenemisestä verrattuna alkuperäiseen testijoukkoon ja kustannukset syntyvät virheitä mahdollisesti paljastavien testitapausten pois jättämisestä. Tässä tutkimuksessa yksittäisten testitapausten katsottiin olevan kehittämiskustannuksiltaan samanarvoisia. Tutkimuksessa ei huomioitu kustannuksia, jotka syntyvät testitapausten valintaprosessiin liittyvistä analysointikustannuksista.

Tutkimuksessa (Kim ym., 2000) valintatekniikoiden huomattiin valitsevan kooltaan ja virheidenlöytämistehokkuudeltaan erilaisia testijoukkoja riippuen siitä, miten paljon ohjelmaan on tehty muutoksia ennen regressiotestauksen suorittamista. Tutkimuksessa huomattiin myös, että osa virheistä jää huomaamatta testaa kaikki uudelleen -tekniikallakin, mikäli ohjelmaan on tehty paljon muutoksia.

Turvalliset valintatekniikat valitsivat lähes kaikki testitapaukset, kun ohjelmaan oli tehty paljon muutoksia. Tällöin testaa kaikki uudelleen -tekniikka voi olla taloudellisempi, koska turvallisen valintatekniikan analysointikustannukset saattavat ylittää pienemmästä testijoukosta saatavat hyödyt. Täytyy muistaa, että turvallisten valintatekniikoiden suorituskyky on riippuvainen monista asioista, kuten ohjelman rakenteesta, vir-

heiden sijainnista ja testijoukkojen rakenteesta. Kimin tutkimuksessa mukana olleet turvalliset valintatekniikat olivat kustannustehokkuudella mitattuna parhaimmillaan, kun ohjelmaan tehtyjen muutosten lukumäärä oli alhainen.

Satunnaistekniikat osoittautuivat yllättävän halvoiksi ja tehokkaiksi. Kun ohjelmaan oli tehty paljon muutoksia, satunnaistekniikoiden keskimääräinen tehokkuus läheni testaa kaikki uudelleen -tekniikan tehokkuutta ja erot eri suorituskertojen välillä tasaantuivat. Kun satunnaistekniikoita käytettiin vähän muutoksia sisältävään ohjelmaan, satunnaistekniikoiden tehokkuus vaihteli hyvin alhaisesta todella suureen. Ohjelmaan tehtyjen muutosten lukumäärän ollessa suuri random(25) osoittautui kustannustehokkaimmaksi. Tämä olettaen, että huomaamatta jääneet virheet eivät aiheuta suuria kustannuksia.

Minimointitekniikka oli kustannustehokkuudella mitattuna parhaimmillaan muutosten lukumäärän ollessa korkea. Näin ollen minimointitekniikkaa kannattaisi käyttää tilanteissa, joissa testitapausten ajaminen on kallista, eivätkä huomaamatta jääneet virheet aiheuta suuria menetyksiä.

4.4 Regressiotestauksen automatisointi

Fewster kertoo kirjassaan Software test automation (Fewster ym., 1999, s. 13-17) testausprosessin viisi askelta: (1) tunnistetaan testattavan ohjelman osat, (2) suunnitellaan testitapaukset (3) rakennetaan testitapaukset, (4) ajetaan testitapaukset ja (5) verrataan ohjelmasta saatuja tuloksia testitapauksissa määritettyihin odotettuihin tuloksiin. Jokaisessa askeleessa voidaan automatisoinnilla saavuttaa hyötyjä, mutta myöhäisemmissä askeleissa automatisoinnilla saavutetut hyödyt ovat suurimmat. Tämä johtuu siitä, että myöhemmät vaiheet ovat luonteeltaan mekaanisia ja suoritetaan monta kertaa, toisin kuin ensimmäiset askeleet, jotka suoritetaan vain kerran yhtä testitapausta kohti.

Regressiotestauksessa suuri osa ajettavista testitapauksista on vanhoja, ainakin kerran ajettuja testitapauksia. Koska regressiotestauksessa uudelleenkäytetään paljon vanhoja testitapauksia, voidaan regressiotestauksen automatisoinnilla saavuttaa suuria etuja verrattuna regressiotestien manuaaliseen ajamiseen.

Testitapauksen muuttaminen sellaiseen muotoon, että sen voi ajaa automaattisesti, vie monta kertaa kauemmin kuin saman testin ajaminen manuaalisesti. Testauksen automatisointi alkaa maksaa itseään takaisin sitten, kun automatisoitua testitapausta toistetaan tarpeeksi monta kertaa. Tämän vuoksi kerran tai pari kertaa toistettavia testejä ei kannata automatisoida (Binder, 1999, s. 802-803). Testin automatisointi vaatii yleensä 2-10 kertaa enemmän työtä, kuin testin ajaminen manuaalisesti. Testitapausten automatisoinnin viemään aikaan vaikuttaa moni asia, kuten automatisoijien kokemus, käytetyt työkalut, testattava ohjelma ja ennen kaikkea testausprosessin ominaisuudet. Mitä tarkemmin testitapaukset on alun perin määritelty, sitä helpompi ne on automatisoida. (Fewster ym., 1999)

Manuaalisella testauksella on tiettyjä etuja verrattuna automaattiseen testaukseen. Ihminen tekee testatessaan huomaamattaan paljon alitajuisia testejä, mutta automatisoidussa testissä tarkastetaan joka kerta vain ne asiat, jotka siinä on määritelty. Manuaalinen testaus on myös joustavampaa, koska ihminen osaa testatessaan miettiä mitä virheitä koodiin tehty muutos voisi aiheuttaa ja muokata testaustaan tämän pohjalta (Fewster ym., 1999, s. 104-105). Sekä manuaalisessa että automaattisessa testauksessa voi esiintyä inhimillisiä virheitä, koska testitapaukset ovat automaattisessa testauksessakin ihmisten suunnittelemia. Manuaalinen testaus on kuitenkin virhealttiimpaa, koska ihmistestaaja on testauksen aikana alttiina ympäristön häiriöille.

5 UML-MÄÄRITYKSIIN POHJAUTUVAT VALINTA-TEKNIIKAT

Tässä luvussa kerrotaan, kuinka regressiotestitapaukset voidaan valita hyödyntämällä ohjelmaa kuvaavia UML-kaavioita.

Aluksi selitetään, kuinka ohjelmasta tehdyistä UML-kaavioista voidaan löytää vaaralliset entiteetit. Lopuksi kerrotaan, miten testitapaukset voidaan valita löydettyjen vaarallisten entiteettien perusteella.

5.1 Aktiviteettikaavio

Chen kertoo artikkelissaan (Chen ym., 2002) aktiviteettikaavion (activity diagram) käytöstä ohjelmaan tehtyjen muutosten jäljittämisessä ja regressiotestitapausten valinnasta.

Löytääkseen määrittelyihin tehdyt muutokset Chenin tekniikka käyttää Rothermelin vertailualgoritmia (Rothermel ym., 1997), josta on kerrottu tarkemmin Holopaisen selvityksessä (Holopainen, 2004). Algoritmi vertailee alkuperäistä ja muutettua kaaviota ja etsii niiden välillä olevia muutoksia. Chenin tekniikka käy läpi aktiviteettikaaviota, toisin kuin Rothermelin tekniikka, joka käy läpi kontrollivirtakaaviota (control-flow graph). Aktiviteettikaavion solmuiksi luokitellaan aloitus- ja lopetusmerkit, toiminnat, päätöskohdat, synkronointiviivat (synkronoinnin haarautumisen ja yhdistymisen välinen alue tulkitaan yhdeksi solmuksi) sekä signaalin lähettäjät ja vastaanottajat. Aktiviteettikaavion kaaria ovat siirtymät ja viestit. Vertailualgoritmi käy samaan aikaan läpi sekä vanhaa että uutta aktiviteettikaaviota ja vertaa kaavioiden kaaria ja solmuja yksi kerrallaan. Mikäli algoritmi löytää poikkeaman kaavioiden väliltä, se merkitsee poikkeavan solmun tai kaaren vaaralliseksi, lopettaa kyseisen polun kulkemisen ja palaa takaisin etsimään vielä kulkemattomia osia. Solmu tai kaari on vaarallinen silloin, kun sen toiminta saattaa poiketa vanhan ja uuden version välillä. Kun kaaviosta on paikannettu muuttuneet osat, voidaan testitapaukset valita jäljitettävyysmatriisin avulla.

Kaikki ohjelmaan tehdyt muutokset eivät kuitenkaan muuta ohjelman määrittelyitä. Ohjelmaan tehdyt virheiden korjaukset ovat usein tällaisia muutoksia. Tällöin pitää tun-

nistaa ne aktiviteettikaavion solmut ja kaaret, joihin muutos vaikuttaa. Tehdessään koodimuutoksia ohjelman kehittäjän täytyy ylläpitää koodimuutos-dokumenttia, jossa koodiin tehdyt muutokset on yhdistetty aktiviteettikaavion eri osiin. Testaaja puolestaan ylläpitää testiprofiili-dokumenttia, jossa löydetyt virheet on yhdistetty aktiviteettikaavion osiin. (Chen ym., 2002)

5.2 Käyttötapauskaavio, sekvenssikaavio ja luokkakaavio

Käyttötapauskaavioita, sekvenssikaavioita ja luokkakaavioita käsitellään yhdessä, koska käyttötapauskaavion muuttumista ei voida todentaa tarkasti ilman sekvenssikaavioiden apua ja sekvenssikaavion muuttumisen huomaamiseksi täytyy tarkastella luokkakaavioita.

Uuden ja vanhan ohjelmaversion käyttötapauskaavioiden välillä voi olla seuraavanlaisia muutoksia, jotka vaikuttavat regressiotestien valintaan. Uuden ohjelmaversion käyttötapauskaaviossa voi olla uusia, poistettuja ja muuttuneita käyttötapauksia. Uudet ja poistetut käyttötapaukset löydetään vertailemalla keskenään vanhaa ja uutta käyttötapauskaaviota, mutta käyttötapauksiin tulleita muutoksia ei voida löytää pelkän käyttötapauskaavion avulla.

Muuttuneiden käyttötapausten löytämiseksi täytyy tutkia käyttötapauksista tehtyjä sekvenssikaavioita. Jokaiselle käyttötapaukselle pitää olla olemassa oma sekvenssikaavio, jossa kyseistä käyttötapausta kuvataan sarjana metodikutsujen aiheuttamia viestejä sekvenssikaavion läpi. Käyttötapaus luokitellaan muuttuneeksi, jos siitä tehdyssä sekvenssikaaviossa on uusia, poistettuja tai muuttuneita metodeja tai niiden mahdollisiin kutsujärjestyksiin on tullut muutoksia. Muuttuneiden metodien löytämiseksi täytyy tutkia luokkakaavioita. (Briand ym., 2002)

Luokkakaaviot tarjoavat komponentin sisältöä kuvaavan luokkahierarkian ja tietoa eri luokista. (Wu ym., 2003) Muutokset metodeissa voidaan löytää vertailemalla ohjelman alkuperäistä luokkakaaviota muutettuun luokkakaavioon. Uuden ja vanhan ohjelmaversion luokkakaavioiden välillä voi olla seuraavanlaisia muutoksia, jotka vaikuttavat regressiotestien valintaan. Uuden ohjelmaversion luokkakaaviossa voi olla uusia tai poistettuja muuttujia, metodeja, luokkia ja luokkien välisiä suhteita. Edellä mainittuihin

on voinut tulla myös muutoksia. Muuttuja luokitellaan muuttuneeksi silloin, kun sen vaikutusaluetta (staattinen tai ei-staattinen), tyyppiä tai näkyvyyttä on muutettu. Metodi luokitellaan muuttuneeksi silloin, kun sillä on uusi esi- tai jälkiehto, sen käyttämiin muuttujiin on tullut muutoksia tai sen kulkemiin (navigate) suhteisiin (relationship) on tullut muutoksia. Metodin kulkemat suhteet on lueteltu metodin jälkiehdoissa. Suhde luokitellaan muuttuneeksi, mikäli sen tyyppi (esim. yleistys, yhteys, kooste), yhteyksien lukumäärä (multiplicity) tai yhteyden suunta (navigability) on muuttunut. Yhteystyyppiä oleva suhde luokitellaan muuttuneeksi myös silloin, kun siihen liitettyyn yhteysluokkaan on tullut muutoksia. Luokka luokitellaan muuttuneeksi, jos sen attribuutteja, metodeja tai suhteita on lisätty, poistettu tai muutettu. (Briand ym., 2002)

Yksi testitapaus testaa yhtä käyttötapausta (use case), mutta samalle käyttötapaukselle saattaa olla useita testitapauksia. Testitapaus on sarja metodikutsuja, jotka testitapausta ajettaessa suoritetaan. Testitapausta ajettaessa kutsutaan suorasti vain rajapintametodeja. Jotta regressiotestitapaukset voitaisiin valikoida tarkasti, pitää testitapauksessa näiden suorien metodikutsujen lisäksi kertoa myös epäsuorat metodikutsut, jotka aiheutuvat toisiaan kutsuvista metodeista. Jokaiselle metodikutsulle määritellään kutsuttavan metodin nimi, kutsuva luokka ja kutsun kohdeluokka. Automatisoinnin helpottamiseksi metodikutsujen kuvaamiseen käytetään säännöllisiä lausekkeita. (Briand ym., 2002)

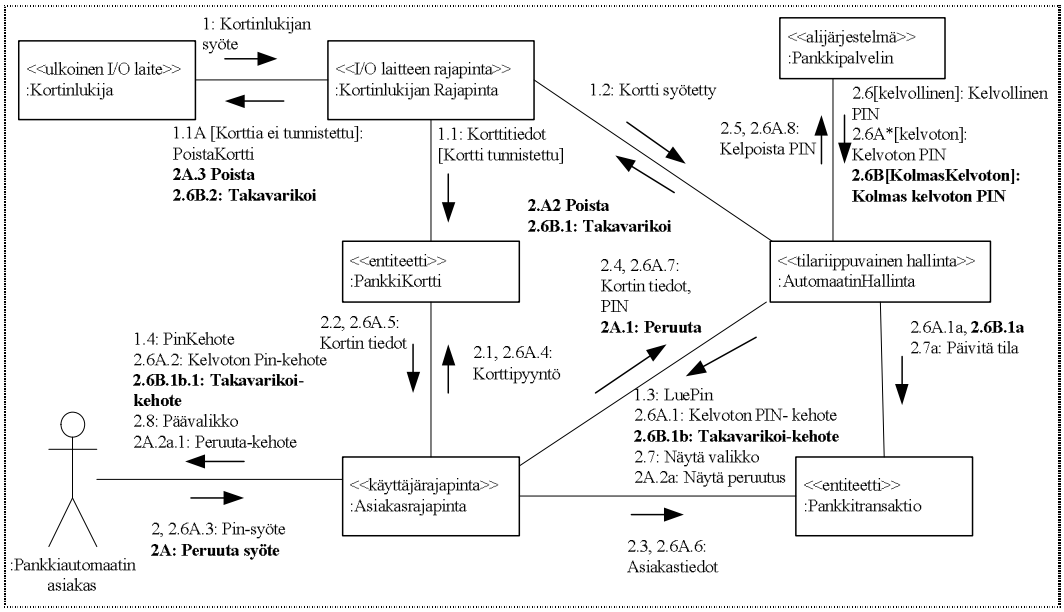
5.3 Yhteistyökaavio ja tilakaavio

Tässä luvussa esitetään Wun artikkelissaan (Wu ym., 2003) esittämiä tapoja hyödyntää ohjelmasta tehtyjä yhteistyö-, tila ja luokkakaavioita regressiotestauksessa. Luokkakaavioita on käsitelty luvussa 5.2, joten tässä ne ohitetaan maininnalla. Wu kertoo, kuinka vaaralliset entiteetit voidaan löytää ja antaa lisäksi neuvoja vaarallisten entiteettien regressiotestaukseen. Asian havainnollistamiseksi esimerkkinä käytetään Wun artikkelissaan esittämiä kaavioita pankkiautomaatin toiminnasta.

Ensin tarkastellaan regressiotestausta korjaavan ylläpitoprosessin aikana ja sen jälkeen täydentävän ja sopeuttavan ylläpitoprosessin aikana. Eri ylläpitoprosesseista on kerrottu luvussa 2.4.

5.3.1 Regressiotestaus korjaavan ylläpitoprosessin aikana

Luokkakaavioilla (class diagram) voidaan määritellä luokkien ominaisuuksia, operaatioita ja rajoituksia sekä luokkien välisiä periytymissuhteita. Yhteistyökaavioita (collaboration diagram) käytetään luokkien välisen vuorovaikutuksen kuvaamiseen, ja tilakaavioita (state chart) käytetään olioiden tai koko komponentin käyttäytymisen kuvaamiseen.



Kuva 4. Yhteistyökaavio Pankkikortin PIN-numeron kelpoistamiseen liittyen (Wu ym., 2003)

Yhteistyökaaviot havainnollistavat komponentin sisältämien olioiden välistä vuorovaikutusta. Kuva 4 näyttää olioiden vuorovaikutuksen lisäksi myös olioiden välisten siirtymien suoritusjärjestyksen. Esimerkiksi siirtymien suoritusjärjestys 1 - 1.1 - 1.2 - 1.3 - 1.4, 2 - 2.1 - 2.2 - 2.3 - 2.4 - 2.5 - 2.6 - 2.7 - 2.8 esittelee PIN-numeron kelpoistamisprosessin kokonaisuudessaan. Yhteistyökaavioissa merkitään isoilla kirjaimilla vaihtoehtoisia kaaria. Esimerkiksi riippuen kortin hyväksymisestä suoritetaan joko kaari 1.1 tai 1.1A. Pienet kirjaimet kertovat samaan aikaan suoritettavista siirtymistä. Esimerkiksi kaaret 2.7 ja 2.7a suoritetaan samanaikaisesti.

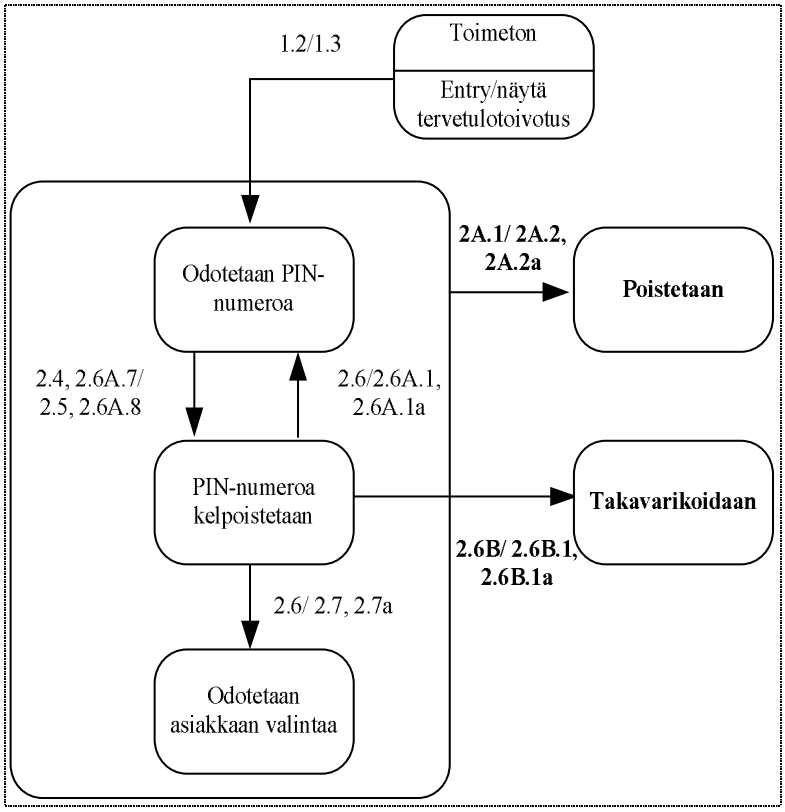
Erilaiset ohjelmaan tehdyt muutokset vaikuttavat yhteistyökaavioihin eri tavoilla:

- *Paikalliset muutokset johonkin luokan sisältämään funktioon*: Yhteistyökaavioissa funktiokutsuja merkitään olioiden välisillä viesteillä. Esimerkiksi kuvassa

4 viestit 2.1 ja 2.2 (hanki Pankkikortin tiedot) vastaa Pankkikortti-olion sisältämän luePankkikortinTiedot-metodin kutsua. Jos muutos tehdään toisten olioiden kanssa kommunikoivaan funktioon, myös siihen liittyvät viestit voivat muuttua, joten tällaiset viestit merkitään vaarallisiksi.

- *Vuorovaikutusjärjestykseen mahdollisesti vaikuttavat muutokset*: Funktion sisältämiä kutsuja voidaan lisätä ja poistaa, tai niihin voidaan tehdä muutoksia. Mikäli funktiokutsu lisätään, täytyy muutetun funktion jälkeen lisätä uusi sekvenssi. Esimerkiksi jos haluamme lisätä siirtymien 2.2 ja 2.3 välille uuden viestin, täytyy näiden siirtymien välille lisätä uusi siirtymä, jonka numero on 2.2.1. Kun funktiokutsu poistetaan, täytyy myös siihen liittyvät viestit poistaa kaaviosta. Tämän vuoksi poistettua viestiä seuraavien viestien numerointia pitäisi muuttaa. Tämä voidaan kuitenkin välttää tekemällä poistetun viestin tilalle valeviesti, joka on tosiasiassa pelkkä silmukka funktioon itseensä.

Tilakaavioita käytetään kuvaamaan olion tai komponentin tilan muutoksia. Tilakaavioiden ja yhteistyökaavioiden tulee olla keskenään yhtenäisiä. Kun yhteistyökaavioihin on tullut muutoksia, voidaan muutokset systemaattisesti siirtää suoraan tilakaavioon. Esimerkiksi kuvassa 4 on lihavoidulla tekstillä merkitty kahta tehtyä muutosta: Käyttäjälle on annettu mahdollisuus perua transaktio (2A: Peruuta syöte), ja järjestelmälle on lisätty mahdollisuus takavarikoida kortti, jos PIN-koodi kirjoitetaan väärin kolme kertaa peräkkäin. Muutosten seurauksena kuvassa 5 esitettyyn tilakaavioon on lisätty kaksi uutta tilaa *Poistetaan* ja *Takavarikoidaan*.



Kuva 5. Tilakaavio pankkiautomaattiohjelman kontrollikaaviosta (Wu ym., 2003).

Ohjelmaan tehdyt muutokset ovat voineet aiheuttaa virheitä epäsuorasti muiden komponentin osien toimintaan. Tämänlaiset virheet voivat olla kahdenlaisia (Wu ym., 2003):

- *Muutosten vaikutukset kontrollivirtaan*. Muutettuja ohjelman osia voidaan kutsua useissa eri tilanteissa. Esimerkiksi kuvaan 4 lisätty toimintosarja 2A, 2A.1, 2A.2 ja 2A.3 voidaan aloittaa kolmesta eri tilanteesta. Tilat on merkitty kuvassa 5 olevaan tilakaavioon: (1) odotetaan PIN-numeroa, (2) PIN-numeroa kelpoistetaan ja (3) odotetaan asiakkaan valintaa. Tilakaavion tummennetut tilat ja kaaret merkitään vaarallisiksi. Pyrittäessä korkeaan laatuun pitää ohjelmasta testata uudelleen myös kaikki polut, jotka sisältävät sellaisia tiloja tai siirtymiä, joihin muutos on voinut vaikuttaa.
- *Muutosten vaikutukset tietoriippuvuussuhteisiin* (data dependence). Komponentin rajapinnan kutsu on oikeasti komponentin implementoiman funktion kutsu. Tästä johtuen, kun rajapinnassa *a* määritellyllä funktiolla on tietoriippuvaisuus-

suhde toisessa rajapinnassa *b* määritettyyn funktioon, voi rajapintojen kutsujärjestyksellä olla vaikutusta lopputulokseen. Yhteistyökaaviossa entiteetti-olioon menevä viesti merkitsee yleensä olion sisältämien tietojen päivitystä, ellei oliosta heti sisään tulevan viestin jälkeen lähde viestiä ulospäin. Mikäli sisään tulevaa viestiä seuraa vastaus, tarkoittaa tämä usein sitä, että oliosta on juuri haettu tietoa tai olio on käsitellyt sisään menevää tietoa ja palauttanut tuloksen. Entiteetti-olion tietojen muuttaminen voi aiheuttaa virheitä myös muissa olion kanssa vuorovaikutuksessa olevissa olioissa. Tämä kannattaa ottaa huomioon testausta suunniteltaessa.

5.3.2 Regressiotestaus täydentävän ja sopeuttavan ylläpitoprosessin aikana

Täydentävä ja sopeuttava ylläpitoprosessi voi muuttaa ohjelmistokomponentin vaatimuksia. Wu keskittyi artikkelissaan kuitenkin vain sellaisiin tapauksiin, joissa vaatimukset eivät muutu.

Kontrollivirtojen yhtäläisyysarviointi

Yhteistyökaavioissa poluilla esitetään erilaisia kontrollijonoja. Esimerkiksi kuva 6 sisältää kolme erilaista polkua W3, W3A ja W3B. Jokaiselle polulle on määritetty *vartija* (guard), joka määrittää milloin komponentti suorittaa kyseisen polun. Vartija voi olla esimerkiksi Boolen lauseke, esimerkissä (kuva 6) tilillä olevat rahat voivat olla joko riittäviä tai riittämättömiä. *Kontekstirajoite* (context constraint) on yhdistelmä vartijoita, jotka liittyvät komponentin rajapinnan suoritukseen. Esimerkiksi kuvassa 6 on näytetty erään testitapauksen suoritusjärjestys vanhassa komponentissa: W1-W2-W3-W4-W5. Suorituksella on kaksi vartijaa: [Kelvollinen tili] ja [Riittävät rahat]. Uudessa komponentissa saman testitapauksen vartijoita on kolme: [Päivittäinen nostoraja ei ylity], [Kelvollinen tili] ja [Riittävät rahat].

Testattavien tapausten määrä riippuu siitä, miten komponenttia muutetaan (Wu ym., 2003):

- *Komponentin vartijat ja kontekstirajoitukset pysyvät samoina kuin vanhassa komponentissa*. Tällöin regressiotestauksessa on tarpeellista testata jokainen rajapinta uudelleen.
- *Muutetussa komponentissa on alkuperäistä komponenttia vähemmän vartijoita.* Tällöin kannattaa testata poistetun vartijan molemmat (tosi, epätosi) arvot.
- *Muutetussa komponentissa vartijoita on lisätty, poistettu tai yhdistelty*. Tässä tilanteessa voidaan testata kaikki uudet vartijat ja kaikki sellaiset kontekstirajoitukset, jotka eroavat alkuperäisen komponentin kontekstirajoituksista, kun niistä on poistettu uudet vartijat.
- *Muutettu komponentti sisältää uusia vartijoita, jotka osaltaan ohjaavat suoritusta*. Tällöin on tarpeellista testata uudet vartijat. Muuttumattomien vartijoiden yhdistelmiä ei ole tarpeellista testata uudelleen. Alla esitetään esimerkki tähän liittyen.

Esimerkiksi vanhalla komponentilla on kaksi vartijaa (tili ja rahat) ja kolme erilaista kontekstirajoitetta (kuva 6):

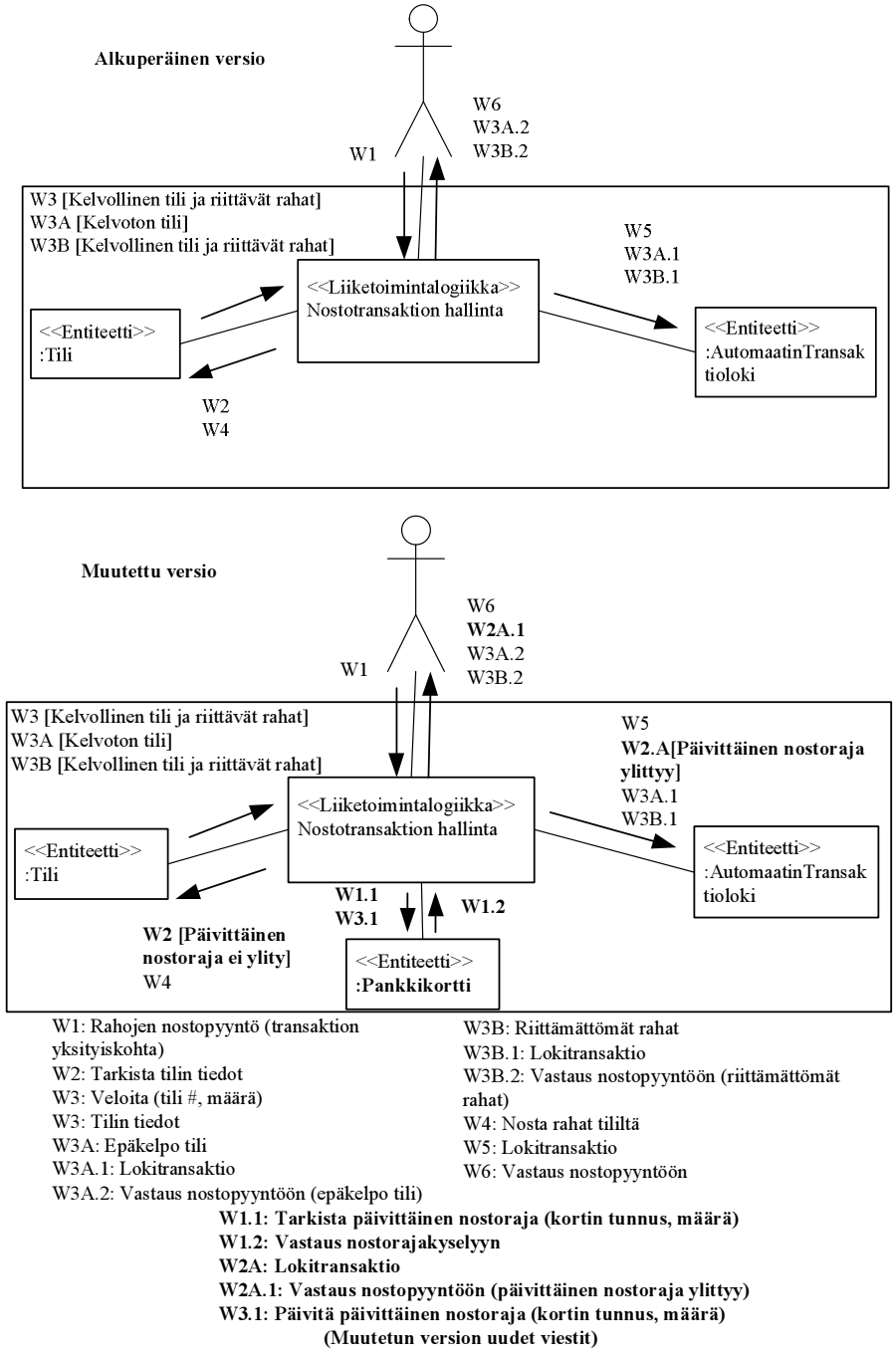
- [Kelvollinen tili] ja [Riittävät rahat]
- [Kelvollinen tili] ja [Riittämättömät rahat]
- [Kelvoton tili]

Muutetulla komponentilla on yksi uusi vartija [Päivittäinen nostoraja]. Tällöin muutetulla komponentilla on neljä kontekstirajoitetta:

- [Päivittäinen nostoraja ylittyy]
- [Päivittäinen nostoraja ei ylity], [Kelvollinen tili] ja [Riittävät rahat]
- [Päivittäinen nostoraja ei ylity], [Kelvollinen tili] ja [Riittämättömät rahat]
- [Päivittäinen nostoraja ei ylity], [Kelvoton tili]

Tässä tapauksessa on tarpeellista testata uudet vartijat, eli [Päivittäinen nostoraja ylittyy] ja [Päivittäinen nostoraja ei ylity]. Muuttumattomien vartijoiden yhdistelmiä ei ole tarpeellista testata uudelleen, joten tässä tapauksessa kannattaa testata kohta 1 ja yksi kohdista 2, 3 ja 4.

Edellä on oletettu, että vartijoiden järjestyksellä ei ole vaikutusta komponentin toimintaan. Käytännössä joissain tilanteissa vartijoiden järjestyksellä on vaikutusta komponentin toimintaan, mikä täytyy ottaa huomioon regressiotestausta suunniteltaessa.



Kuva 6. Yhteistyökaavio liittyen nostotapahtumaan pankkiautomaatista (muokattu lähteestä Wu ym., 2003).

Tietoriippuvuuksien yhtäläisyysarviointi

Komponentin muuttamisella voi olla vaikutuksia komponentin rajapintojen tietoriippuvuuksien suhteisiin. Kun tehdään uusi tietoriippuvuussuhde, pitää joko kehittää uusi testitapaus tai vaihtoehtoisesti käyttää vanhoja testitapauksia tietoriippuvuussuhteiden testaamiseksi. Jos tietoriippuvuussuhde poistetaan, pitää muutettu komponentti testata vanhalla testitapauksella, joka oli tehty kyseisen tietoriippuvuussuhteen testaamiseen.

5.4 Regressiotestitapausten valinta

Kaikissa edellä mainituissa tekniikoissa tunnistetaan ohjelman määrittelyistä vaaralliset entiteetit, eli ohjelman osat, jotka ovat voineet muuttua ohjelmaan tehdyn muutoksen seurauksena. Jotta regressiotestien valinta olisi mahdollista, täytyy ohjelman entiteettien ja testitapausten välille luoda *jäljitettävyysmatriisi*, jossa kerrotaan, mitä entiteettejä testitapaukset käyvät läpi ohjelman ajon aikana. Esimerkiksi Chenin tekniikassa luodaan jäljitettävyysmatriisi, johon merkitään, mitä aktiviteettikaavion kaaria ja solmuja eri testitapaukset testaavat.

Kun vaaralliset entiteetit on tunnistettu, voidaan regressiotestauksessa käytettävään regressiotestijoukkoon valita vaarallisia entiteettejä testaavia testitapauksia. Testitapaukset voidaan valita mukaan luvussa 4.2.1 esitettyjen testausstrategioiden perusteella. Mikäli halutaan säästää kustannuksissa, voidaan käyttää minimointistrategiaa, jolloin regressiotestijoukkoon valitaan minimimäärä testitapauksia, jotka yhdessä kattavat kaikki vaaralliset entiteetit. Turvallinen, mutta myös kalliimpi strategia on valita kaikki vaarallisia entiteettejä testaavat testitapaukset mukaan regressiotestaukseen. Näiden lisäksi on olemassa myös erilaisiin kattavuuskriteereihin perustuvia valintastrategioita.

Alkuperäisen testijoukon testitapaukset voidaan jakaa kolmeen eri ryhmään määrityksiin tulleiden muutosten perusteella (Briand ym., 2002):

Vanhentuneet (obsolete) testitapaukset ovat käyttökelvottomia eikä niitä voida ajaa muutetulla ohjelmalla, joten ne pitää joko jättää pois regressiotestijoukosta tai vaihtoehtoisesti korjata toimiviksi. Esimerkiksi Briandin tekniikassa testitapaukset on yhdistetty käyttötapauksiin, joille puolestaan on tehty omat sekvenssikaaviot. Testitapaus luokitellaan vanhentuneeksi, jos testitapauksen kutsumien rajapintaluokkien metodeja on lisät-

ty/poistettu tai niiden mahdollisiin suoritusjärjestyksiin on tullut muutos. Jos käyttötapaus poistetaan, vanhentuvat myös kyseiseen käyttötapaukseen liittyvät testitapaukset.

Uudelleentestattavat (retestable) testitapaukset käyvät läpi ohjelman vaarallisia entiteettejä ja ovat käyttökelpoisia regressiotestauksessa.

Uudelleenkäytettävät (reusable) testitapaukset ovat käyttökelpoisia, mutta ne eivät testaa ohjelman vaarallisia entiteettejä, joten niitä ei kannata ottaa mukaan regressiotestijoukkoon.

6 KOMPONENTTIPOHJAISTEN OHJELMISTOJEN REGRESSIOTESTAUS

Testaamalla ohjelman komponenttia riittävästi, voidaan luottaa siihen, että komponentti toimii virheettömästi. Tämä ei kuitenkaan anna aihetta luottaa järjestelmään, johon komponentti liitetään edes siinä tapauksessa, että järjestelmä on jo aiemmin läpäissyt riittävän testijoukon. Muutettu komponentti voi toimia virheellisesti yhteistyössä muiden komponenttien kanssa ja aiheuttaa virheitä muiden komponenttien toimintaan (Binder, 1999, s. 755-756).

Komponentteihin perustuvan ohjelmiston regressiotestaus eroaa merkittävästi perinteisen ohjelmiston regressiotestauksesta. Perinteisissä järjestelmissä ohjelmiston ylläpidosta huolehtii usein sama ryhmä, joka on rakentanut kyseisen ohjelmiston. Testaajilla on näin käytössään järjestelmän lähdekoodi ja vankka tuntemus järjestelmän toiminnasta. Komponenttiohjelmistoissa komponenttien ylläpidosta huolehtivat yleensä ulkopuoliset komponenttien valmistajat. Komponentin käyttäjillä ei ole tietoa komponentin toteutuksesta, vaan he ovat riippuvaisia komponenttien valmistajien tarjoamasta tiedosta. Tämä vaikeuttaa komponenttisysteemin testausta. (Gao ym., 2003)

Useat komponenttipohjaisuuden tuomat ongelmat johtuvat siitä, ettei komponenteista ole saatavilla riittävästi tietoa. Jos komponentin valmistaja tekee komponenttiin muutoksen, ei komponentin käyttäjä välttämättä tiedä muutoksen laajuutta ja joutuu näin ollen testaamaan uudelleen kaikki muutettuun komponenttiin yhteydessä olevat järjestelmän osat. Monikaan komponenttien valmistaja ei ymmärrettävistä syistä halua paljastaa komponentin lähdekoodia, ja toisaalta vaikka lähdekoodi olisikin saatavissa, voi tarvittavan tiedon löytäminen olla vaikeaa. Ongelma voidaan ratkaista tarjoamalla komponenteista riittävästi tietoa.

Komponentin valmistajat näkevät komponentin toteutuksen, joten he voivat käyttää regressiotestauksessaan apuna luvuissa 4 ja 5 esitettyjä tekniikoita. Tässä luvussa kerrotaan kaksi tapaa, kuinka komponentin valmistajat voivat tarjota komponentin käyttäjille valikoitua tietoa komponentista. Tämän tiedon avulla komponentin käyttäjät voivat tehostaa komponenttia käyttävän järjestelmän regressiotestausta.

Luvussa 6.1 kerrotaan Orson artikkelissaan (Orso ym., 2001) esittämä tapa liittää komponentteihin *metasisältöä*, jonka perusteella komponentteihin tehdyt muutokset voidaan havaita. Luvussa 6.2 kerrotaan Sajeevin artikkelissaan (Sajeev ym., 2003) esittämä tapa ilmoittaa komponentteihin tehdyistä muutoksista tätä tarkoitusta varten tehdyn luokkakaavion avulla.

6.1 Metasisällön hyödyntäminen komponenttiin tehtyjen muutosten selvittämisessä

Yksi tapa ilmoittaa komponentteihin tehdyistä muutoksista on liittää komponenttien oheen *metasisältöä*. Metasisältö sisältää tietoa (metadata) komponentista sekä metodeja (metamethods) tiedon keräämiseen.

Orson tutkimuksessa (Orso ym., 2001) esitetään kaksi lähestymistapaa metasisällön esittämiselle ja käyttämiselle komponentin regressiotestauksessa tarvittavien testitapausten valinnassa. Toinen tässä luvussa esiteltävistä lähestymistavoista käyttää ohjelmakoodia ja toinen ohjelman määrittelyjä. Havainnollisuuden vuoksi esitetään esimerkkiohjelma, joka käyttää yhtä komponenttia.

```
 1. public class VendingMachine {
 2.
 3.  final private int COIN = 25;
 4.  final private int VALUE = 50;
 5.  private int totValue;
 6.  private int currValue;
 7.  private Dispenser d;
 8.
 9.  public VendingMachine() {
10.   totValue = 0;
11.   currValue = 0;
12.   d = new Dispenser();
13.  }
14.
15.  public void insert() {
16.   currValue += COIN;
17.   System.out.println("Current value = " + currValue );
18.  }
19.
20.  public void return() {
21.   if ( currValue == 0 )
22.    System.err.println( "no coins to return" );
23.   else {
24.    System.out.println("Take your coins");
25.    currValue = 0;}
26.  }
27.
28.  public void vend( int selection ) {
29.   int expense;
30.   expense = d.dispense( currValue, selection );
31.   totValue += expense;
32.   currValue -= expense;
33.   System.out.println( "Current value = " + currValue );
34.  }
35. } // class VendingMachine
36.
37. public class Dispenser {
38.
39.  final private int MAXSEL = 20;
40.  final private int VAL = 50;
41.  private int[] availSelectionVals = {2,3,13};
42.
43.  public int dispense( int credit, int sel ) {
44.   int val=0;
45.   if ( credit == 0 )
46.    System.err.println("No coins inserted");
47.   else if ( sel > MAXSEL )
48.    System.err.println("Wrong selection "+sel);
49.   else if ( !available( sel ) )
50.    System.err.println("Selection "+sel+" unavailable");
51.   else {
52.    val = VAL;
53.    if ( credit < val )
54.     System.err.println("Enter "+(val-credit)+" coins");
55.    else
56.     System.err.println("Take selection"); }
57.   return val;
58.  }
59.
60.  private boolean available( int sel ) {
61.   for (int i = 0; i<availSelectionVals.length; i++)
62.    if (availSelectionVals[i] == sel) return true;
63.   return false;
64.  }
65. } // class Dispenser
```

Kuva 7. Esimerkkiohjelman ja -komponentin ohjelmakoodi (Orso ym., 2001).

Java-kielellä kirjoitettujen VendingMachine-pääohjelman sekä Dispenser-komponentin koodi esitetään ohjelman rakenteen havainnollistamiseksi kuvassa 7, mutta oletetaan että komponentin lähdekoodi ei ole näkyvissä pääohjelman kehittäjälle.

6.1.1 Artikkelissa esitetty esimerkkiohjelma

Orson artikkelissaan esittämä esimerkkiohjelma mallintaa myyntiautomaattia, johon käyttäjä voi laittaa rahaa, pyytää automaattia palauttamaan ylimääräiset rahat sekä pyytää automaattia antamaan tietyn tuotteen. Myyntiautomaatti antaa virheilmoituksen, mikäli pyydettyä tuotetta ei ole saatavissa, automaattiin on laitettu liian vähän rahaa tai valinta on virheellinen.

Taulukossa 1 on pääohjelmalle luotu testijoukko, jonka jokainen testitapaus koostuu sarjasta metodikutsuja. Testitapaukset on jaettu kolmeen jonoon (1-16, 17-20, 21-25) VendingMachine-ohjelman vend-metodille annetun syötteen *selection* mukaan. Ensimmäisessä testitapausten jonossa (1-16) syöte on 3, toisessa jonossa (17-20) syöte on 9 ja kolmannessa jonossa (21-15) syöte on 35. Taulukon tulos-sarake näyttää testitapauksen tuloksen. Kuten taulukosta nähdään, testitapaukset 4 ja 14 epäonnistuivat, johtuen Dispenser-komponentin dispense-metodissa olevasta virheestä: Jos haluttu tuote on saatavissa, mutta syötetty rahamäärä on enemmän kuin nolla mutta silti riittämätön, muuttujan *val* arvoa ei muuteta nollaksi, mikä johtaa ohjelman virheelliseen toimintaan.

Taulukko 1. VendingMachine-ohjelman testijoukko (Orso ym., 2001)

Testitapaus #	Testitapaus	Tulos
Vend-metodin parametri: 3 (valinta kelvollinen, tuote saatavissa)		
1	return	Läpi
2	vend	Läpi
3	insert, return	Läpi
4	insert, vend	Virhe
5	insert, insert, return	Läpi
6	insert, insert, vend	Läpi
7	insert, insert, insert, return	Läpi
8	insert, insert, insert, vend	Läpi
9	insert, insert, insert, insert, return	Läpi
10	insert, insert, insert, insert, vend	Läpi
11	insert, insert, return, vend	Läpi
12	insert, insert, vend, vend	Läpi
13	insert, insert, insert, return, vend	Läpi
14	insert, insert, insert, vend, vend	Virhe
15	insert, insert, insert, insert, return, vend	Läpi
16	insert, insert, insert, insert, vend, vend	Läpi
Vend-metodin parametri: 9 (valinta kelvollinen, tuote ei saatavissa)		
17	vend	Läpi
18	insert, vend	Läpi
19	insert, return, vend	Läpi
20	insert, vend, vend	Läpi
Vend-metodin parametri: 35 (valinta kelvoton)		
21	vend	Läpi
22	insert, vend	Läpi
23	insert, insert, vend	Läpi
24	insert, insert, insert, vend	Läpi
25	insert, insert, insert, insert, vend	Läpi

Taulukko 2. VendingMachine-ohjelman testitapausten kattavuudet (Orso ym., 2001).

Testi-tapaus #	Katetut merkitykselliset kaaret
1	(9,10),(20,21),(21,22).
2	(9,10),(28,29)
3	(9,10),(15,16),(20,21),(21,23)
4	(9,10),(15,16),(28,29)
5	(9,10),(15,16),(20,21),(21,23)
6	(9,10),(15,16),(28,29)
7	(9,10),(15,16),(20,21),(21,23)
8	(9,10),(15,16),(28,29)
9	(9,10),(15,16),(20,21),(21,23)
10	(9,10),(15,16),(28,29)
11	(9,10),(15,16),(20,21),(21,23),(28,29)
12	(9,10),(15,16),(28,29)
13	(9,10),(15,16),(20,21),(21,23),(28,29)
14	(9,10),(15,16),(28,29)
15	(9,10),(15,16),(20,21),(21,23),(28,29)
16	(9,10),(15,16),(28,29)
17	(9,10),(28,29)
18	(9,10),(15,16),(28,29)
19	(9,10),(15,16),(20,21),(21,23),(28,29)
20	(9,10),(15,16),(28,29)
21	(9,10),(28,29)
22	(9,10),(15,16),(28,29)
23	(9,10),(15,16),(28,29)
24	(9,10),(15,16),(28,29)
25	(9,10),(15,16),(28,29)

Komponentin kehittäjä huomaa virheen ja korjaa sen lisäämällä lauseen "val = 0" rivien 54 ja 55 väliin ja toimittaa korjatun komponentin VendingMachine-ohjelman kehittäjälle. Ilman tietoa komponenttiin tehdyistä muutoksista, kehittäjän täytyy käydä läpi uudelleen kaikki ohjelman testitapaukset. Seuraavaksi esitetään kaksi tekniikkaa, joiden avulla pääohjelman kehittäjä pystyy valitsemaan testijoukostaan vain tarvittavat testitapaukset.

6.1.2 Koodiin pohjautuvat metasisällöt testitapausten valintaan

Tässä esitetään Orson artikkelissaan esittämä metasisältöön perustuva tekniikka käytettäväksi koodiin pohjautuvien regressiotestitapausten valintatekniikoiden kanssa. Koodiin perustuvat valintatekniikat valitsevat regressiotestauksessa käytettävät testitapaukset niiden kattavuuden perusteella. Kattavuudella voidaan tarkoittaa monia eri asioita, esimerkiksi lausekattavuutta, polkukattavuutta tai kaarikattavuutta. Tässä tapauksessa käytetään kaarikattavuutta, jolloin eri testitapausten kattavuus mitataan niiden läpikäymien *merkityksellisten kaarten* perusteella. Merkityksellisiksi kaariksi kutsutaan metodin sisäänmenokohtia (kaaret (9,10), (15,16), (20,21) ja (28,29)) sekä päätöslauseiden kaaria (kaaret (21,22) ja (21,23)). Taulukossa 2 on eri testitapausten kattavuus VendingMachine-ohjelman merkityksellisten kaarten osalta.

Koodiin perustuvia regressiotestien valintamenetelmiä on kehitetty useita (Chen ym., 1994; Harrold ym., 2001; Rothermel ym., 1997; Rothermel ym., 2000), joista tähän on valittu käytettäväksi Rothermelin ja Harroldin kehittämää tekniikkaa (Rothermel ym., 1997) käyttävä työkalu nimeltään DejaVu. DejaVu käy samanaikaisesti läpi alkuperäisen sekä muutetun version kontrollivirtakaaviot (control-flow graph), tunnistaa samalla muuttuneet kaaret ja valitsee muuttuneita kaaria testaavat testitapaukset.

Ilman tietoa komponentin toteutuksesta, DejaVu ei voi luoda komponentille kontrollivirtakaaviota. Näin ollen DejaVu merkitsee kaikki muutetun komponentin metodeja kutsuvat pääohjelman kaaret vaarallisiksi ja valitsee regressiotestijoukkoon mukaan kaikki kyseisiä kaaria testaavat testitapaukset. Ensin DejaVu vertaa vanhaa, muuttumatonta komponenttia käyttävän VendingMachine-ohjelman kontrollivirtakaaviota muuttunutta komponenttia käyttävän VendingMachine-ohjelman kontrollivirtakaavioon ja huomaa, että muutos on vaikuttanut kaareen (28,29). Tämän jälkeen DejaVu valitsee kaikki ne testitapaukset, jotka testaavat kyseistä kaarta (testitapaukset 2, 4, 6, 8, 10-25).

Hyödyntämällä komponenttiin liitettyä metasisältöä, voidaan regressiotestijoukkoa pienentää. Regressiotestitapausten valintaprosessin tehostamiseksi komponentista pitää olla saatavilla kolmenlaista metasisältöä. Ensinnäkin komponentin käyttäjän tulee tietää testijoukon kaarikattavuus komponentin sisällä. Toiseksi tulee tietää komponentin versio. Kolmanneksi pitää olla keino kysyä komponentilta, mihin komponentin kaariin teh-

dyt muutokset ovat vaikuttaneet. Komponentin rakentaja voi tarjota edellä mainitut tiedot liittämällä komponenttiin metatietoa sekä metametodeja.

DejaVu-ohjelmasta voitaisiin rakentaa metasisältöä ymmärtävä versio $\text{DejaVu}_{\text{MA}}$. Tämä ohjelmaversio osaisi rakentaa matriisin "testitapaukset"-"katetut kaaret" keräämällä listan katetuista komponentin kaarista jokaiselle testitapaukselle. $\text{DejaVu}_{\text{MA}}$ voisi hakea matriisin rakentamiseksi tarvittavat tiedot kommunikoimalla komponentin kanssa seuraavan toimintamallin (Orso ym., 2000) mukaisesti:

1. Hae komponentista kattavuuteen liittyvien metasisältöjen tyypit ja talleta ne listaan.
2. Tarkista sisältääkö saatu lista regressiotestauksessa tarvittavat metasisällöt. Mikäli näin on, jatketaan seuraavaan askeleeseen.
3. Hae komponentista tiedot, kuinka metametodeilla päästään käsiksi komponentin sisältämään metatietoon.
4. Käyttämällä hyväksi edellisessä askeleessa kerättyjä tietoja, aktivoidaan komponentin sisään rakennetut palvelut, jotta kattavuustiedot voitaisiin hakea komponentista.
5. Nyt komponentin kattavuuspalvelut ovat aktivoituina, joten voidaan aloittaa kattavuustiedon kerääminen testijoukon sisältämille testitapauksille:
    a. Alusta komponentin sisäänrakennettu kattavuus, jotta voidaan selvittää testitapauksen *t* kattavuus.
    b. Aja testitapaus *t*.
    c. Selvitä testitapauksen *t* kattavuus.

Testitapausten valintaprosessi on seuraavanlainen: $\text{DejaVu}_{\text{MA}}$ (1) selvittää muuttamattoman Dispenser-komponentin version, (2) selvittää mihin kaariin muutettuun versioon tehdyt muutokset ovat vaikuttaneet, ja käyttäen hyväkseen vaiheessa 1 kerättyä tietoa (3) valitsee kerätyn matriisin ja vaikutettujen kaarten perusteella regressiotestijoukkoon lisättävät testitapaukset. Esimerkin tapauksessa Dispenser-komponenttiin tehdyt muutokset ovat vaikuttaneet ainoastaan kaareen (53, 54), ja testitapauksista ainoastaan 4 ja 14 testaavat kyseistä kaarta. Näin ollen metatietoa käyttämällä tarvittavien testitapa-

usten määrä saatiin vähennettyä kahteen, kun se ilman metatietoa saadussa testijoukossa oli 20.

6.1.3 Määrityksiin pohjautuvat metasisällöt testitapausten valintaan

Tässä esitetään Orson artikkelissaan esittämä metasisältöön perustuva tekniikka käytettäväksi määrityksiin pohjautuvien regressiotestitapausten valintatekniikoiden kanssa. Tekniikasta tehty tutkimus on liitteenä (liite C).

Yksi tällainen tekniikka on kategorioihinjakotekniikka (category-partition method), joka tuottaa järjestelmän toiminnallisten osien *testikehyksiä* (test frame). Tekniikka koostuu seuraavista vaiheista (Paakki, 2000):

1. Tunnistetaan määrityksiä tutkimalla ohjelman *toiminnalliset osat*, eli osat joita on mahdollista testata itsenäisesti. Esimerkkiohjelmasta voidaan löytää toiminnallinen osa (metodi) **dispense** (kuva 8).
2. Tunnistetaan toiminnallisten osien syötteet ja ympäristötekijät. Ympäristötekijöillä tarkoitetaan järjestelmän tilaa sillä hetkellä, kun kyseistä toiminnallista osaa suoritetaan. Selvittämällä ympäristötekijät voidaan testattava toiminnallinen osa testattaessa tuoda samaan tilaan, kuin se olisi koko järjestelmän ollessa käynnissä. Tämän vaiheen jälkeen testitapaus koostuu syötteistä ja ympäristöarvoista.
3. Määritetään syöte- ja ympäristökategoriat. *Kategorialla* tarkoitetaan syötteen tai ympäristötekijän ominaispiirrettä. Jokainen kategorian sisältämä arvo vaikuttaa järjestelmän toimintaan tietyssä syötteestä/ympäristöstä riippuvassa tilanteessa. Esimerkiksi voimme määrittää edellä valitulle toiminnalliselle osalle kaksi syötekategoriaa **credit** ja **selection** sekä ympäristökategorian **availability**.
4. Jokainen kategoria jaetaan erillisiin ekvivalenssiluokkiin. Ekvivalenssiluokat sisältävät arvoja, joiden on tarkoitus vaikuttaa testattavaan järjestelmään samansuuntaisesti. Esimerkin tapauksessa kategoria **credit** on jaettu ekvivalenssiluokkiin nolla, riittämätön, riittävä ja yli (kuva 8).
5. Jokaiselle toiminnalliselle osalle tehdään *testimääritys*, joka koostuu seuraavista osista:

- Luettelo kategorioista.
- Luettelo jokaisen kategorian sisältämistä ekvivalenssiluokista.
- Joukko *testikehyksiä*. Testikehykseen valitaan jokaisesta kategoriasta joko yksi tai ei yhtään ekvivalenssiluokkaa ja määritellään yksi looginen kokoonpano syöteparametreja ja ympäristötekijöitä, joilla toiminnallista osaa voidaan testata. Esimerkiksi kuvan 9 testikehys numero neljään on valittu **selection**-kategoriasta ekvivalenssiluokka "kelvollinen", **availability**-kategoriasta ekvivalenssiluokka "saatavissa" ja **credit**-kategoriasta ekvivalenssiluokka "riittämätön".

6. Testimääritys varustetaan (annotate) toisiinsa yhteydessä olevien valintojen välisillä *rajoituksilla*, esimerkiksi toteamalla rajoitus, että tietyt ekvivalenssiluokkien yhdistelmät eivät voi esiintyä samassa testikehyksessä. Kuvan 8 esimerkissä **availability**-kategorian ekvivalenssiluokka "saatavissa" voi olla samassa testikehyksessä **selection**-kategorian "kelvollinen"-ekvivalenssiluokan kanssa. Ekvivalenssiluokka "saatavissa" ei voi esiintyä samassa testikehyksessä ekvivalenssiluokan "kelvoton" kanssa, koska ohjelmalle ei voida keksiä sellaiset ehdot täyttäviä syötteitä.

Aivan kuten koodiin pohjautuvat regressiotestien valintatekniikat, myös määrityksiin pohjautuvat tekniikat tallentavat alkuperäisen testijoukon sisältämien testitapausten kattavuuden ohjelman entiteettien joukossa. Kategorioihinjakotekniikassa entiteetit ovat ohjelmasta tehtyjä testikehyksiä. Testikehys katsotaan tietyn testitapauksen kattamaksi, (1) jos testitapauksen toiminnallisille osille kohdistetut kutsut vastaavat testikehyksessä olevia ekvivalenssiluokkia ja (2) komponentin tila vastaa testikehyksen ympäristöarvoja. Esimerkiksi taulukossa 1 esitetty testitapaus numero 2 kattaa kuvassa 9 esitetyn testikehyksen numero 3, koska molemmissa tapauksissa **selection** on "kelvollinen", **availability** on "saatavissa" ja **credit** on "nolla".

Jotta tietyn testitapauksen kattavuus komponentissa voitaisiin laskea, täytyy koodia käsitellä testikehysten mukaan. Tällä tavalla voidaan tunnistaa jokaisen testitapauksen kattamat testikehykset. Kun tiedämme, mihin testikehyksiin ohjelmaan tehty muutos on

voinut vaikuttaa, voimme valita regressiotestijoukkoon tällaisia testikehyksiä testaavia testitapauksia.

Kuvassa 8 esitetään Disperser-komponentin **dispense**-metodin mahdolliset kategoriat, ekvivalenssiluokat ja niiden sisältämät rajoitukset. Kuvassa 9 on kuvan 8 määrityksistä saadut testikehykset.

Testitapausten valitsemiseksi komponentista tulee olla saatavilla kolmenlaista metasisältöä. Ensinnäkin tulee tietää testijoukon testikehyskattavuus komponentin sisällä. Toiseksi tulee tietää komponentin versio. Kolmanneksi pitää olla keino kysyä komponentilta, mihin komponentin testikehyksiin tehdyt muutokset vaikuttivat. Komponentin rakentaja voi tarjota edellä mainitut tiedot liittämällä komponenttiin metatietoa sekä metametodeja.

Toiminnallinen osa **dispense**		
Syötekategoriat:		
credit		
	-nolla	[if availability = saatavissa]
	-riittämätön	[if availability = saatavissa]
	-riittävä	[if availability = saatavissa]
	-yli	[if availability = saatavissa]
selection		
	-kelvollinen	[sisältö kelvollinen]
	-kelvoton	[virhe]
Ympäristökategoria:		
availability		
		[if selection = kelvollinen]
	-saatavissa	[sisältö saatavissa]
	-ei-saatavissa	[if selection = kelvollinen] [virhe]

Kuva 8. Dispenser-komponentin eri kategoriat, valinnat ja rajoitukset (Orso ym., 2001).

Toiminnallinen osa **dispense**			
1	selection: kelvoton	availability: X	credit: X
2	selection: kelvollinen	availability: ei-saatavissa	credit: X
3	selection: kelvollinen	availability: saatavissa	credit: nolla
4	selection: kelvollinen	availability: saatavissa	credit: riittämätön
5	selection: kelvollinen	availability: saatavissa	credit: riittävä
6	selection: kelvollinen	availability: saatavissa	credit: yli

Kuva 9. Dispenser-komponentin testikehykset (X-kirjaimella merkityt arvot ovat merkityksettömiä) (Orso ym., 2001).

Taulukko 3. Testitapausten kattamat Disperser-komponentin testikehykset (Orso ym., 2001).

Testitapaus #	Katetut testikehykset	Testitapaus #	Katetut testikehykset
1		14	4, 6
2	3	15	3
3		16	3, 6
4	4	17	2
5		18	2
6	5	19	2
7		20	2
8	6	21	1
9		22	1
10	6	23	1
11	3	24	1
12	3, 5	25	1
13	3		

DejaVu-ohjelmasta voidaan rakentaa metasisältöä ymmärtävä versio $\text{DejaVu}_{\text{MA}}$, joka toimii saman toimintamallin (Orso ym., 2000) mukaan kuin koodiin pohjautuva versio paitsi, että kaarien sijaan mitataan testikehysten kattavuutta.

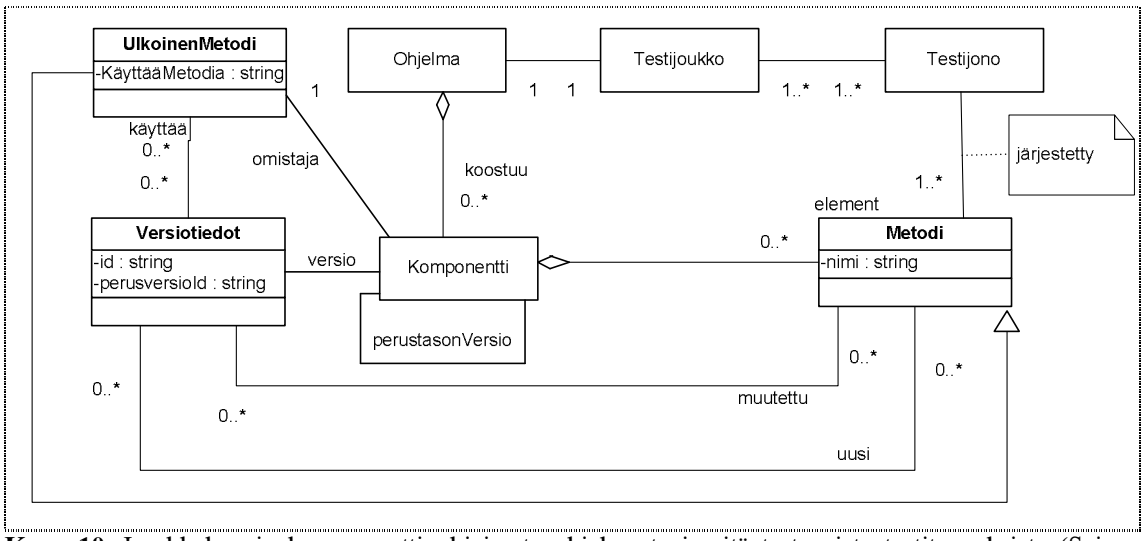
Testitapausten valintaprosessi on seuraavanlainen: $\text{DejaVu}_{\text{MA}}$ (1) selvittää muuttamattoman Dispenser-komponentin version, (2) käyttäen hyväkseen vaiheessa 1 kerättyä tietoa selvittää mihin kaariin muutettuun versioon tehdyt muutokset ovat vaikuttaneet, (3) valitsee kerätyn matriisin ja vaikutettujen testikehysten perusteella regressiotestijoukkoon lisättävät testitapaukset. Esimerkin tapauksessa ajetaan ensin kaikki testitapaukset ja kerätään testitapausten kattavuustiedot (taulukko 3). Kun saadaan muutettu Dispenser-komponentin versio, päätellään mihin testikehyksiin tehdyt muutokset ovat vaikuttaneet. Huomataan että muutokset ovat vaikuttaneet ainoastaan testikehys numero neljään. Lopuksi päätellään "testitapaukset"-"testikehykset" -matriisin avulla regressiotestijoukkoon valittavat testitapaukset. Taulukon 3 tietojen pohjalta valitaan testitapaukset 4 ja 14.

Kuten koodipohjaisen lähestymistavan tapauksessa, myös määrityksiin pohjautuvan lähestymistavan tapauksessa metasisällön käyttäminen johtaa testijoukon merkittävään pienenemiseen.

6.2 Luokkakaavion hyödyntäminen komponenttiin tehtyjen muutosten selvittämisessä

Artikkelissa (Sajeev ym., 2003) kerrotaan COTS-komponenteista (Commercial Off-The-Shelf) koottujen järjestelmien määrittelyihin pohjautuvasta regressiotestauksesta. Luvussa 6.1 esitellyssä Orson tekniikassa (Orso ym., 2001) tiedot komponentteihin tehdyistä muutoksista ilmoitetaan komponenttien metatiedoissa. Sajeevin artikkelissa tiedot komponentteihin tehdyistä muutoksista ilmoitetaan tarkoitusta varten tehdyn luokkakaavion avulla. Sajeevin tekniikassa huomioidaan paremmin mahdollisuus käyttää muutettuja metodeja epäsuorasti, eli toisen metodin välityksellä. Orson tekniikassa vaaditaan, että metatiedot sisältävät komponentin sisäisen haarakattavuuden (branch coverage) ja mahdollisuuden kysyä komponentilta metametodien avulla muutosten seurauksena vaarallisiksi muuttuneita haaroja. (Sajeev ym., 2003)

Kuvassa 10 on esitetty luokkakaavio komponenttipohjaisesta ohjelmasta, sen testitapauksista ja ohjelman komponenttien versiotiedoista. Ohjelmalle on olemassa testijoukko, joka sisältää yhden tai useamman testijonon. Testijono on järjestetty lista metodikutsuja. Ohjelma koostuu yhdestä tai useammasta komponentista ja komponentti koostuu yhdestä tai useammasta metodista. Komponentilla on versiotiedot, jotka sisältävät senhetkisen version tunnisteen (id) ja komponentin perusversion tunnisteen (perusversioId). Komponentin versiotiedot sisältävät myös tiedot komponentin muuttuneista rajapintametodeista sekä komponentin uusista rajapintametodeista. Näiden lisäksi komponentin versiotiedoissa kerrotaan myös komponentin käyttämät metodit, jotka sijaitsevat jossakin toisessa komponentissa. (Sajeev ym., 2003)



Kuva 10. Luokkakaavio komponenttipohjaisesta ohjelmasta ja sitä testaavista testitapauksista (Sajeev ym., 2003).

Regressiotestaukseen pyritään valitsemaan sellaiset testijonot, jotka sisältävät kutsuja muuttuneisiin metodeihin. Sajeevin tekniikka valitsee regressiotestauksessa käytettävät testit kahdessa vaiheessa. Ensin valitaan kaikki sellaiset testijonot, jotka kutsuvat suoraan muutettua metodia. Tämä voidaan tehdä käymällä läpi testijonoja ja valitsemalla kaikki sellaiset testijonot, jotka sisältävät kutsun metodiin, joka löytyy komponenttien muutettu-listalta. Seuraavaksi valitaan kaikki sellaiset testijonot, jotka kutsuvat metodia, joka kutsuu suorasti tai epäsuorasti muutettua metodia. Tämä voidaan tehdä valitsemalla kaikki sellaiset testijonot, jotka sisältävät kutsun sellaisen metodiin, joka suoraan tai epäsuorasti kutsuu muutettua metodia. (Sajeev ym., 2003)

7 YHTEENVETO

Tämän tutkielman ensimmäinen tavoite oli selventää regressiotestaus-termiä ja regressiotestausprosessia, koska regressiotestauksen määritelmä on monille epäselvä ja varsinkin Suomessa asiaa on tutkittu vähän. Tutkielman toinen tavoite oli kertoa, kuinka regressiotestausta voitaisiin tehostaa. Parhaat tavat regressiotestauksen tehostamiseen ovat testauksen automatisointi ja ajettavan regressiotestijoukon pienentäminen valintatekniikoiden avulla. Tässä tutkielmassa keskityttiin regressiotestauksen tehostamiseen pienentämällä ajettavaa testijoukkoa valintatekniikoiden avulla. Myös muita keinoja regressiotestauksen tehostamiseen käsitellään tutkielmassa lyhyesti. Regressiotestauksen automatisointi on laaja aihe, joten se päätettiin ohittaa tässä tutkielmassa maininnalla.

Regressiotestauksen, -testijoukon ja -testitapauksen tarkka määritteleminen on vaikeaa, koska termeille ei ole olemassa mitään yleistä, kaikkien noudattamaa määritelmää. Monet yleisessä käytössä olevat määritelmät eivät päde kaikissa mahdollisissa tilanteissa.

Yksi yleinen tapa määritellä regressiotestaus on seuraava: Regressiotestaus-termillä tarkoitetaan ohjelman uudelleentestausta muutoksen jälkeen pyrkimyksenä varmistaa, ettei ohjelma ole muutoksen seurauksena taantunut, eli ettei muutos ole aiheuttanut ohjelmaan uusia virheitä.

Ongelmaksi muodostuu tilanne, jossa uudella regressiotestitapauksella löydetään ohjelmasta virhe, joka ei ole syntynyt ohjelmaan tehdyn muutoksen seurauksena, vaan on ollut ohjelmassa alusta asti. Voidaanko ohjelman tällöin sanoa taantuneen, koska ohjelmassa olevien virheiden lukumäärä ei ole tosiasiassa kasvanut? Tällaista tilannetta ei pääsisi syntymään, mikäli regressiotestauksella tarkoitettaisiin ainoastaan vanhojen testitapausten uudelleenkäyttöä.

Regressiotestauksen määritelmän rajoittaminen ainoastaan vanhojen testitapausten uudelleenkäytöksi on kuitenkin ongelmallista. Tällöin pitäisi päättää, tarkoitetaanko regressiotestitapauksella ainoastaan vanhan, täysin muuttamattoman testitapauksen uudelleenkäyttöä, vai olisiko testitapausta mahdollista muuttaa (uudelleenkelpoistaa) ja

kutsua sitä yhä regressiotestitapaukseksi? Miten suuria muutoksia vanhaan testitapaukseen voitaisiin tehdä, että sitä voisi vielä kutsua regressiotestitapaukseksi?

Tässä tutkielmassa regressiotestauksen on määritelty tarkoittavan ohjelmaan tehdyn muutoksen oikeellisuuden sekä muutetussa ohjelmassa ennen muutosta olleen toiminnallisuuden oikeellisuuden tarkistamiseksi. Edellä olevassa määritelmässä muutoksen käsitetään olevan sellainen, joka ei lisää ohjelmaan uutta toiminnallisuutta. Uuden toiminnallisuuden testaamisen tulkitaan kuuluvan tavallisen testauksen piiriin.

Lisäksi ongelmia aiheuttaa progressiivisen testauksen määrittely, koska tutkielman aineistossa ei kyseistä termiä käytetä erottamaan ei-regressiotestausta regressiotestauksesta. Sen sijaan aineistossa progressiiviseen testaukseen viitataan sanalla testaus. Koska testaus tarkoittaa tietyssä asiayhteydessä koko testausprosessia, aiheuttaa termi välillä ymmärtämisongelmia.

Testaa kaikki uudelleen -tekniikka on hyvä ja suositeltava tekniikka silloin, kun koko testijoukon ajaminen ei vie kohtuuttomasti aikaa. Valintatekniikoita käytettäessä joudutaan tekemään kompromisseja turvallisuuden ja testijoukon koon välillä. Minimointitekniikoilla saavutetaan pienin regressiotestijoukko, mutta toisaalta tällöin osa virheistä jää huomaamatta. Turvallisia tekniikoita käyttämällä löydetään kaikki löydettävissä olevat virheet, mutta ajettava testijoukko jää verrattain suureksi. Ennen turvallisten tekniikoiden käyttöä pitää arvioida, peittävätkö pienentyneestä regressiotestijoukosta saatavat hyödyt turvallisten tekniikoiden analysointikustannukset. Tutkimuksissa (Rothermel ym. 1998a) on huomattu, että analysointikustannukset ovat verrattain pienitöisiä verrattuna testitapausten ajamiseen kuluvaan aikaan. Tämä tukee osaltaan turvallisten tekniikoiden käyttöä.

Turvallisia valintatekniikoita suositellaan käytettäviksi tilanteissa, joissa ohjelmaan ei ole tullut kovin paljoa muutoksia ja alkuperäisen testijoukon koko on suuri. Mitä enemmän ohjelmaan on tehty muutoksia, sitä suuremmalla todennäköisyydellä turvallinen valintatekniikka valitsee lähes kaikki testitapaukset. Tällöin valittu testijoukko sisältää suuren määrän testitapauksia, jotka eivät käytännössä löydä ohjelmasta virheitä. Minimointitekniikoilla valittuja testijoukkoja voi käyttää ohjelman toimivuuden nopeaan tarkastamiseen regressiotestausprosessin aikana. Ennen regressiotestausprosessin lopet-

tamista suositellaan, että ohjelmaa testataan kertaalleen koko testijoukolla (Kaner ym., 1999).

Määrityksiin pohjautuvat valintatekniikat ja koodiin pohjautuvat valintatekniikat täydentävät toisiaan, joten regressiotestausprosessin aikana kannattaa mahdollisuuksien mukaan käyttää molempia. Määrityspohjaisten tekniikoiden käyttö on suositeltavaa korkeammilla testaustasoilla (kuva 1), mutta alemmilla tasoilla (moduulitestaus) koodipohjaisilla tekniikoilla voidaan saavuttaa parempia tuloksia.

Valintatekniikoiden valitsemien testijoukkojen suuruutta ei voida tarkasti ennustaa etukäteen, koska testausprosessiin liittyvät eri muuttujat vaikuttavat lopputulokseen. Tämä vaikeuttaa osaltaan testauksen aikataulutuksen suunnittelua. Testaa kaikki uudelleen -tekniikassa tiedetään ennalta, miten kauan vanhan testijoukon ajaminen kesti viime kierroksella. Tällöin on helpompi arvioida, kuinka kauan testijoukon uudelleenajaminen kestää.

Testijoukkojen harventamistekniikoiden käyttö on kannattavaa, kun ohjelma on käynyt läpi useita testauskierroksia, joissa ohjelman testijoukkoon on lisätty uusia testitapauksia. Harventamistekniikoista on tehty suhteellisen vähän tutkimuksia ja osa saaduista tutkimustuloksista on keskenään ristiriitaisia, johtuen testausympäristön monista eri muuttujista. Harventamistekniikoiden käyttäminen on hyödyllistä, kun testijoukko pääsee kasvamaan suureksi. Testijoukon sisältämien testitapausten pois heittämisessä kannattaa kuitenkin olla tarkkana, koska harventamistekniikat vähentävät testitapauksia yleensä kattavuuden perusteella, joten testijoukon kyky löytää virheitä saattaa laskea.

Priorisointitekniikat ovat turvallisia käyttää, koska ne ainoastaan uudelleenjärjestävät testitapaukset. Priorisointitekniikoita voidaan käyttää koko alkuperäiseen testijoukkoon tai valintatekniikoilla valittuun testijoukon osaan. Priorisointitekniikat ovat hyödyllisiä silloin, kun ohjelman testijoukon ajaminen kestää kauan. Suurin osa priorisointitekniikoista tehdyistä tutkimuksista vertailee priorisointitekniikoita sen perusteella, miten nopeasti ne löytävät virheitä. Näissä tutkimuksissa saadut tulokset ovat osittain ristiriitaisia, johtuen testausympäristön monista eri muuttujista. Priorisointitekniikoille voidaan kuitenkin asettaa muitakin päämääriä, kuin virheiden löytäminen mahdollisimman aikaisessa vaiheessa testijoukon ajoprosessia.

Automatisointi on regressiotestauksessa välttämätöntä, koska suuri osa regressiotestauksessa ajettavista testitapauksista ajetaan useaan kertaan ohjelman elinkaaren aikana. Regressiotestausprosessin automatisointi päätettiin kuitenkin jättää pois tämän tutkielman aihepiiristä.

COTS-komponenttien regressiotestaaminen aiheuttaa ylimääräisiä haasteita. Komponentin valmistaja pystyy tarkastelemaan komponentin toteutusta ja voi näin ollen käyttää regressiotestauksessaan hyväksi luvuissa 4 ja 5 mainittuja tekniikoita. COTS-komponentin käyttäjällä, eli asiakkaalla ei yleensä kuitenkaan ole mahdollisuutta tarkastella komponentin toteutusta. Tämän vuoksi komponentin valmistajan täytyisi lisätä komponentteihin ylimääräisiä tietoja, joista kerrottiin tarkemmin luvussa 6. Tällä hetkellä tällaisten tietojen lisääminen on harvinaista, mutta tällaisten tietojen lisääminen voi antaa valmistajalle ylimääräisen kilpailuedun muihin komponenttien valmistajiin nähden.

8 LÄHTEET

Bates S., Horwitz S. Incremental program testing using program dependence graphs. *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, 1993.

Binder R. *Testing object-oriented systems*. Addison-Wesley, 1999.

Binkley D. Reducing the cost of regression testing by semantics guided test case selection. *Proceedings of International Conference on Software Maintenance*, 1995.

Black J., Melachrinoudis E., Kaeli D. Bi-criteria models for all-uses test suite reduction. *Proceedings of the 26th International Conference on Software Engineering*, 2004.

Bohner S., Arnold R. *Software change impact analysis*, Wiley-IEEE Computer Society Press, 1996.

Briand L. C., Labiche Y., Soccar G. Automating impact analysis and regression test selection based on UML designs, *International Conference on Software Maintenance (ICSM'02)*, s. 252, 2002.

Chen Y., Probert R., Sims P. Specification-based regression test selection with risk analysis, *Proceedings of the 2002 conference of the centre for advanced studies collaborative research*, 2002.

Chen Y-F., Rosenblum D S., Vo K-P. TestTube: A system for selective regression testing. *Proceedings of the 16th international conference on Software engineering*, 1994.

Elbaum S., Gable D., Rothermel G. The impact of software evolution on code coverage information. *Proceedings of the International Conference on Software Maintenance*, s. 169-179, 2001a.

Elbaum S., Gable D., Rothermel G. Understanding and measuring the sources of variation in the prioritization of regression test suites. *Proceedings of the Seventh International Software Metrics Symposium. Institution of Electrical and Electronics Engineers*, 2001b.

Elbaum S., Kallakuri P., Malishevesky A. G., Rothermel G., Kanduri S. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification and Reliability*, Vol. 13, No. 2, s. 65-83, 2003.

Elbaum S., Malishevsky A. G., Rothermel G. Test case prioritization: A family of empirical studies. *IEEE Transactions on software engineering*, Vol. 28, No. 2, s. 159-182, 2002.

Elbaum S., Rothermel G., Kanduri S., Malishevsky A. G. Selecting a cost-effective test case prioritization technique. *Software Quality Journal* Vol. 12, No. 3, 2004.

Fewster M., Graham D. *Software test automation*, Addison-Wesley, 1999.

Gao J.Z., Tsao J., Wu Y. *Testing and quality assurance for component-based software*. Artech House, 2003.

Graves T. L., Harrold M. J., Kim J-M., Porter A., Rothermel G. An Empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, Vol. 10, No. 2, s. 184-208, 2001.

Harrold M. J., Gupta R., Soffa M. L. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 1993.

Harrold M. Testing evolving software. *Journal of Systems and Software*, Vol. 47, No. 2-3, s. 173-181, 1999.

Harrold M.J, Jones J, Li T., Liang D., Orso A., Pennings M., Sinha S., Spoon S., Gujarathi A. Regression test selection for java software. *ACM SIGPLAN Notices, Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, Vol. 36, No. 11, 2001.

Holopainen J. *Regressiotestauksen tehostaminen*. PlugIT-hankkeen selvityksiä ja raportteja, Kuopion yliopisto, 2004.
<URL:http://www.cs.uku.fi/research/Teho/RegressioselvitysJuha04.pdf>.
Viitattu 25.4.2005.

Horgan J., London S. ATAC: A data flow coverage testing tool for C. *Proc. of the Symosium. on Assessment of Quality Software Development Tools*, s. 2-10, 1992.

Jones C. *Software quality: analysis and guidelines for success*. International Thompson Computer Press, 1997.

Jones J. A., Harrold M. J. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*. Vol. 29, No. 3, s. 195-209, 2003.

Kaner C., Falk J., Nguyen H. *Testing computer software*. Wiley, 1999.

Kim J-M., Porter A., Rothermel G. An empirical study of regression test application frequency. *Proceedings of the 22nd International Conference on Software Engineering*, s. 126-135, 2000.

Law J., Rothermel G. Whole program path-based dynamic impact analysis. *Proceedings of the 25th international conference on software engineering*, 2003.

Myers G., *The art of software testing*. John Wiley and Sons, 1979.

Onoma A. K., Tsai W-T., Poonawala M., Suganuma H. Regression testing in an industrial environment. *Communications of the ACM*, Vol. 41, No. 5, 1998.

Orso A., Apiwattanapong T., Harrold M-J. Leveraging field data for impact analysis and regression testing. *Proceedings of the 11th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, s. 128-137, 2003.

Orso A., Apiwattanapong T., Law J., Rothermel G., Harrold M-J. An empirical comparison of dynamic impact analysis algorithms, *Proceedings of the 26th international conference on software engineering*, s. 491-500, 2004.

Orso A., Harrold M. J., Rosenblum D. S., Rothermel G., Huynsook D., Soffa M-L. Using component metacontent to support the regression testing of component-based software. *IEEE International Conference on Software Maintenance (ICSM'01),* 2001.

Orso A., Harrold M. J.., Rosenblum D. S. Component metadata for software engineering tasks. *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000), LNCS Vol. 1999*, s. 129-144, 2000.

Paakki J. *Ohjelmistojen testaus*, Helsingin yliopisto, 2000.

Rosenblum D. S., Weyuker E. J. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, Vol. 23, No. 3, s. 146-156, 1997.

Rothermel G, Harrold M. J. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM),* Vol. 6, No. 2, 1997.

Rothermel G, Harrold M. J. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering,* Vol. 22, No. 8, s. 529-551, 1996.

Rothermel G., Elbaum S., Malishevsky A. G., Kallakuri P., Qiu X., On test suite composition and cost-effective regression testing. *Technical Report 03-60-04,* Department of Computer Science, Oregon State University, 2003.

Rothermel G., Elbaum S., Malishevsky A., Kallakuri P., Davia B. The impact of test suite granularity on the cost-effectiveness of regression testing. *Proceedings of the 24th International Conference on Software Engineering*, 2002.

Rothermel G., Harrold M. J. Empirical studies of a safe regression test selection technique *IEEE Transactions on Software Engineering*, Vol. 24, No. 6, 1998a.

Rothermel G., Harrold M. J., Dedhia J. Regression test selection for C++ programs. *Journal of Software Testing, Verification, and Reliability*, Vol. 10, No. 2, 2000.

Rothermel G., Harrold M. J., Ostrin J., Hong C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. *Proceedings of the International Conference on Software Maintenance*, s. 34-43, 1998b.

Rothermel G., Untch R. H., Chu C., Harrold M. J. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, Vol. 27, No. 10, s. 929-948, 2001.

Sajeev A. S. M., Wibowo B. Regression test selection based on version changes of components, *Tenth Asia-Pacific Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference*, s. 78, 2003.

Srivastava A., Thiagarajan J. Effectively prioritizing tests in development environment. ACM SIGSOFT Software Engineering Notes, *Proceedings of the international symposium on Software testing and analysis*, Vol. 27 Issue 4, 2002.

Tikir M. M., Hollingsworth J K. Efficient instrumentation for code coverage testing. *ACM SIGSOFT Software Engineering Notes, Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, Vol. 27, No. 4, 2002.

Virkanen H. *Ohjelmistojen testaus ja virheenjäljitys*, Pro gradu -tutkielma, Kuopion yliopisto, Informaatioteknologian ja Kauppatieteiden laitos, 2002.

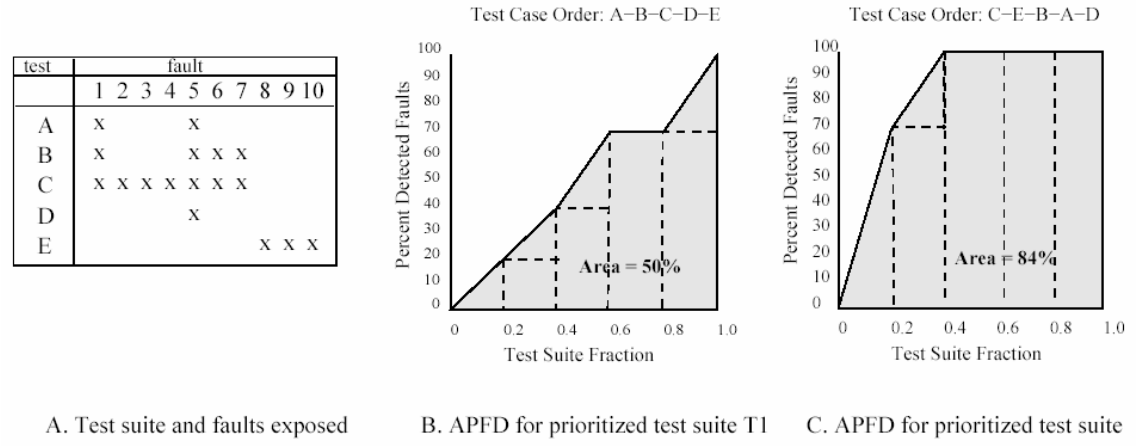
Wong W. E., Horgan J. R.., London S., Mathur A. P. Effect of test set minimization on fault detection effectiveness. *17th International conference of software engineering*, s. 41-50, 1995.

Wu Y., Offutt J. Maintaining evolving component-based software with UML, *Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, 2003.

Liite A

Elbaumin tekemä vertailu priorisointitekniikoista

Tähän mennessä laajimman tutkimuskohdejoukon kattaneessa tutkimuksessa (Elbaum ym., 2004) käytettiin priorisointitekniikoiden tehokkuuserojen selvittämiseksi APFD-metriikkaa (Average Percentage Faults Detected) (Rothermel ym., 2001). APFD-metriikalla mitattuna eri priorisointitekniikat voivat saada pisteitä väliltä 0-100. Suurempi numeroarvo merkitsee nopeampaa virheiden löytymistä.



Kuva 1. Esimerkki APFD-metriikasta (Rothermel ym., 2001).

Esimerkiksi kuvassa 1 vasemmalla on esitetty erään ohjelman testitapauksia ja mitä virheitä niillä löydettiin. Kuvassa keskellä olevassa kaaviossa on ilmennetty järjestämättömän testijoukon löytämien virheiden määrää testijoukon ajamisen aikana. Kuvassa oikealla olevassa kaaviossa on ilmennetty priorisointitekniikalla järjestetyn testijoukon löytämien virheiden määrää testijoukon ajamisen aikana. Mitä suurempi kaavion tummennettu alue on, sitä nopeammin testijoukko on löytänyt virheet. Keskimmäisessä kaaviossa APFD-luku on 50 ja oikeanpuoleisessa kaaviossa APFD-luku on 84. Testitapausten ajojärjestys C-E-B-A-D löytää kaikki esimerkin virheet jo kahden testitapauksen ajamisen jälkeen, joten se on APFD-metriikalla mitattuna tehokkaampi verrattuna testitapausten ajojärjestykseen A-B-C-D-E.

Elbaumin tutkimus keskittyi seuraaviin neljään tekniikkaan:

- *Total function coverage prioritization* (total). Tämä tekniikka lajittelee testitapaukset niiden funktiokattavuuden perusteella niin, että suurimman funktiokattavuuden omaavat testitapaukset sijoittuvat testijoukon alkupäähän.
- *Additional function coverage prioritization* (addtl). Tämä tekniikka valitsee aluksi suurimman funktiokattavuuden omaavan testitapauksen. Seuraavaksi tekniikka valitsee testitapauksen, joka kattaa suurimman osan vielä kattamattomista funktioista. Tätä toistetaan kunnes kaikki testaukseen sisällytettävät funktiot ovat jonkin testitapauksen kattamia. Prosessi aloitetaan alusta jäljelle jääneiden testitapausten kohdalla ja jatketaan kunnes kaikki testitapaukset on järjestetty.
- *Total binary-diff function coverage prioritization* (total-diff). Tämä tekniikka valitsee testitapaukset niin, että ensiksi suoritetaan sellaiset testitapaukset, jotka kattavat mahdollisimman suuren osan muuttuneista funktioista. Kun valitut testitapaukset kattavat kaikki muuttuneet funktiot, loput testitapaukset järjestetään käyttämällä total function coverage prioritization -tekniikkaa.
- *Additional binary-diff function coverage prioritization* (addtl-diff). Tämä tekniikka valitsee ensin testitapauksen, joka kattaa suurimman osan muuttuneista funktioista. Seuraavaksi valitaan testitapaus, joka kattaa suurimman osan vielä kattamattomista, muuttuneista funktioista. Tätä toistetaan kunnes kaikki testaukseen sisällytettävät, muuttuneet funktiot ovat jonkin testitapauksen kattamia. Jäljelle jääneet testitapaukset järjestetään käyttämällä niihin additional function coverage prioritization -tekniikkaa.

Elbaumin tutkimuksessa priorisointitekniikoiden tehokkuudesta saatiin samantapaisia tuloksia kuin aiemminkin tehdyissä tutkimuksissa. Priorisointitekniikoita käyttämällä saavutettava hyöty vaihtelee suuresti riippuen ohjelman sekä tehdyn muutoksen ominaisuuksista, testijoukon piirteistä sekä edellisten välisistä vuorovaikutussuhteista. Tällä hetkellä ei siis ole mahdollista varmasti tietää, onko valittu priorisointitekniikka varmasti paras mahdollinen tiettyyn tilanteeseen, koska vielä ei tarkkaan tiedetä, miten eri muuttujat vaikuttavat tekniikan tehokkuuteen (Elbaum ym., 2004). Ohjelman, testijou-

kon ja muutosten laadun vaikutusta priorisointitekniikoiden tehokkuuteen (APFD-metriikalla mitattuna) tutkittiin artikkelissa (Elbaum ym., 2001).

Elbaum sai tutkimuksessaan (Elbaum ym., 2004) tulokseksi todennäköisyyksiä, joita voidaan käyttää apuna päätettäessä, mikä tekniikka on todennäköisesti sopivin tiettyyn tapaukseen. Todennäköisyydet riippuivat siitä, miten suureen kustannustehokkuuteen tähdättiin ja miten paljon aikaa käytettiin testijoukon ja ohjelman analysointiin. Elbaumin artikkeli tarjoaa asiasta kiinnostuneelle runsaasti suuntaa antavia todennäköisyyslaskelmia. (Elbaum ym., 2004)

Lähteet:

Elbaum S., Gable D., Rothermel G. Understanding and measuring the sources of variation in the prioritization of regression test suites. *Proceedings of the Seventh International Software Metrics Symposium. Institution of Electrical and Electronics Engineers*, 2001.

Elbaum S., Rothermel G., Kanduri S., Malishevsky A. G. Selecting a cost-effective test case prioritization technique. *Software Quality Journal* Vol. 12, No. 3, 2004.

Rothermel G., Untch R. H., Chu C., Harrold M. J. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, Vol. 27, No. 10, s. 929-948, 2001.

Liite B

Gravesin tekemä vertailu valintatekniikoista

Gravesin tutkimuksessa (Graves ym., 2001) tutkittiin luvussa 4.2.3 mainittuihin lähestymistapoihin kuuluvien tekniikoiden eroja testien valinnassa. Tutkimukseen valittiin vertailun vuoksi mukaan kaksi yksinkertaista ja yleisesti käytössä olevaa tapaa valita testit.

Satunnaisuuteen perustuvat tekniikat (ad hoc/random techniques). Kun testaajilla ei ole käytössään valintatyökalua eikä aikaa uudelleentestata kaikkia testitapauksia, testaajat usein yrittävät valita käytettävät testitapaukset parhaan vaistonsa mukaan. Toinen tapa on arpoa käytettävät testitapaukset.

Testaa kaikki uudelleen -tekniikka (retest-all technique). Tässä tekniikassa yksinkertaisesti valitaan testijoukkoon kaikki alkuperäisen testijoukon testitapaukset.

Gravesin tutkimuksessa tutkittiin erilaisia yhdeksästä C-kielisestä ohjelmasta tehtyjä versioita. Tuloksia tarkasteltaessa pitää ottaa huomioon, että tutkimustulokset ovat todennäköisyysarvioita, jotka perustuvat testattavista ohjelmista saatuihin tuloksiin. Tutkimuksessa huomattiin, että valitut ohjelmat, tehtyjen muutosten laatu sekä testijoukkojen koostumus vaikuttivat testituloksiin merkittävästi. Näiden asioiden tarkkaa vaikutusta tutkimustuloksiin ei ole vielä tarkasti selvitetty.

Satunnaisuuteen perustuvat regressiotestien valintatekniikat olivat keskimäärin hyvin inklusiivisiä (random25-tekniikalla inklusiivisuusmediaani oli 88%). Kun testitapausten valintaprosenttia kasvatettiin (random50 ja random75), myös inklusiivisuus kasvoi, mutta hitaammin. Random-tekniikan perässä oleva numeroarvo kertoo, kuinka monta prosenttia alkuperäisen testisarjan testitapauksista valitaan.

Turvallinen tekniikka oli aina 100-prosenttisen inklusiivinen, mutta valitun testijoukon koko vaihteli suuresti (nollasta sataan prosenttiin). Keskimäärin tekniikka valitsi 60-prosenttia alkuperäisen testijoukon testitapauksista. Gravesin tutkimuksessa havaittiin myös, että keskimäärin vain vähän suuremmat satunnaistekniikalla valitut testijoukot olivat lähes yhtä inklusiivisiä kuin turvallisella tekniikalla valitut testijoukot.

Tietovirtatekniikat ja turvalliset tekniikat olivat kustannustehokkuudeltaan keskimäärin hyvin samanlaisia, huomasivat usein samat virheet ja tuottivat samankokoisen testijoukon.

Minimointitekniikan inklusiivisuus vaihteli suuresti, mutta se valitsi aina pienen testijoukon. Keskimäärin minimointitekniikan inklusiivisuus oli vain 16%. Koska minimointitekniikka valitsi hyvin pienen testijoukon, se voi olla hyödyllinen sellaisissa tilanteissa, joissa testauksen suorittaminen on hyvin kallista ja huomaamatta jääneiden virheiden ei katsota aiheuttavan mittavia haittoja. Analysointikustannuksiltaan erittäin halpa satunnaisvalintatekniikka tuotti vain vähän suuremmilla testijoukoilla yleensä yhtä inklusiivisiä testijoukkoja kuin minimointitekniikka. On kuitenkin mahdotonta valita minimointitekniikan inklusiivisuutta vastaava satunnaistekniikka, ellei tiedetä kuinka suuren testijoukon minimointitekniikka valitsee testattavassa ohjelmassa. Yksi tapa lähestyä tätä ongelmaa olisi käyttää regressiotestauksen valintatekniikoiden ennustemallia (esimerkiksi Rosenblum ym., 1997).

Lähteet:

Graves T. L., Harrold M. J., Kim J-M., Porter A., Rothermel G. An Empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, Vol. 10, No. 2, s. 184-208, 2001.

Rosenblum D. S., Weyuker E. J. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, Vol. 23, No. 3, s. 146-156, 1997.

Liite C

Orson suorittama arviointi metasisältötekniikkaan liittyen

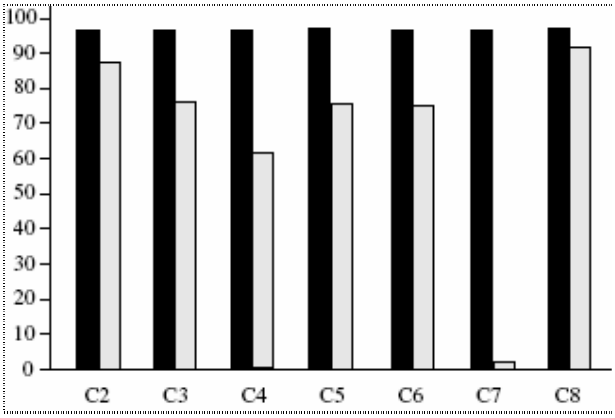
Tutkiakseen metasisällön hyötyjä komponenttipohjaisten ohjelmistojen testauksessa, Orso suoritti työryhmineen seuraavan tutkimuksen. Testattavaksi ohjelmaksi valittiin Java-kielisen SIENA-palvelimen kahdeksan eri versiota. SIENA (Scalable Internet Event Notification Architecture) koostuu kuudesta komponentista (joissa on yhdeksän luokkaa ja noin 1500 koodiriviä) sekä 17 muusta luokasta, jotka sisältävät noin 2000 koodiriviä. Ohjelmaversioiden testauksessa Orso työryhmineen vertaili kahta koodiin pohjautuvaa regressiotestien valintatekniikkaa:

- *Ei metasisältöä.* Järjestelmän kehittäjä tietää ainoastaan, että yksi tai useampi ulkopuolisen tahon toimittama komponentti on muuttunut. Jotta järjestelmän toimivuus voitaisiin varmistaa, kehittäjän on testattava uudelleen kaikki testitapaukset, jotka testaavat komponentteihin yhteydessä olevaa koodia. Tästä tekniikasta käytetään jatkossa lyhennettä NOMETA.
- *Metodi-tason regressiotestien valinta metasisältöä apuna käyttäen.* Kehittäjällä on käytössään tarpeeksi metasisältöä, jotta hän pystyy valitsemaan tarvittavat testitapaukset komponenttien muutettujen osien perusteella. Tästä tekniikasta käytetään jatkossa lyhennettä META.

Kuvassa 1 on esitetty SIENA-ohjelman eri versioille suoritetun testitapausten valinnan tulokset. Kuvassa X-akselilla on esitetty ohjelman eri versiot ja Y-akselilla valitun regressiotestijoukon koko suhteessa alkuperäiseen testijoukkoon, kun testitapaukset valittiin käyttämällä NOMETA-tekniikkaa (musta palkki) ja META-tekniikkaa (harmaa palkki). Kuvasta näkyy, että metatietoa hyödyntävä META-tekniikka valitsi aina pienemmän testijoukon kuin NOMETA-tekniikka. Ohjelman C7-version tapauksessa valittujen regressiotestijoukkojen kokoero oli valtava: NOMETA-tekniikka valitsi 97% alkuperäisen testijoukon testitapauksista, kun taas META-tekniikka valitsi 1.5% testitapauksista. Merkittävä ero johtuu siitä, että C7-versiossa ohjelmaan tehdyt muutokset olivat hyvin pieniä ja koskettivat ainoastaan muutamaa metodia ja testitapausta. Muissa versi-

oissa regressiotestijoukkojen kokoerot olivat pienempiä, erojen vaihdellessa välillä 6%-37%.

Kuvaa 1 tarkasteltaessa pitää ottaa huomioon, että testijoukon koko ei ole suoraan verrannollinen testijoukon ajamiseen kuluvaan aikaan. Tämä johtuu siitä, että eri testitapausten ajamiseen tarvittava aika vaihtelee. Ei voida siis sanoa, että kymmenen prosenttia vähemmän testitapauksia sisältävän testijoukon ajamiseen kuluu aikaa kymmenen prosenttia vähemmän. Testijoukkojen ajamiseen kuluvat ajat on merkitty taulukkoon 1. Taulukon eri sarakkeisiin on merkitty ohjelman eri versiot, sekä NOMETA- ja META-tekniikoiden keräämän testijoukon ajamiseen kuluva aika minuuteissa ja sekunneissa. Kaksi viimeistä riviä kertovat testijoukkojen ajamiseen kuluneen keskimääräisen sekä kokonaisajan.



Kuva 1. META- ja NOMETA-valintatekniikoilla valittujen testijoukkojen koko (Orso ym., 2001).

Taulukko 1. META- ja NOMETA-valintatekniikoiden valitsimien testijoukkojen suoritusajat (minuutit:sekunnit) (Orso ym., 2001).

Versio	NOMETA Suoritusaika	META Suoritusaika
C2	19:44	18:45
C3	19:51	16:57
C4	19:51	13:07
C5	19:52	17:44
C6	19:52	16:40
C7	19:51	00:15
C8	19:49	19:26
keskiarvo	19:50	14:07
yhteensä	138:50	102:54

Hyödyllinen valintatekniikka vähentää testaukseen kuluvaa aikaa ja kustannuksia. Tässä tutkimuksessa ei kuitenkaan mitattu testitapausten valintaan kuluvaa aikaa, joten ei voida sanoa miten paljon kustannustehokkaampi META-tekniikka oli NOMETA-tekniikkaan verrattuna. Toiset tutkimukset (Rothermel ym., 1998) ovat kuitenkin osoittaneet valintaprosessiin kuluvan ajan olevan verrattain alhainen. Lisäksi pitää huomata, että taulukon 1 tulokset kertovat ainoastaan testijoukon ajamiseen kuluvan ajan, mutta testausprosessissa testitapaukset pitää myös kelpoistaa, eli varmistaa, että ne ovat käyt-

tökelpoisia. Jos kelpoistamiseen kuluva aika otettaisiin huomioon, pienemmän testijoukon edut kasvaisivat.

Suhteellisen pienestä testiohjelmasta johtuen saavutetut aikasäästöt voidaan laskea minuuteissa ja sekunneissa. Käytännössä isompien ohjelmien regressiotestaus saattaa kestää useita tunteja tai päiviä, joten tällöin metasisällön käyttö voi johtaa huomattaviin säästöihin.

Lähteet:

Orso A., Harrold M. J., Rosenblum D. S., Rothermel G., Huynsook D., Soffa M-L. Using component metacontent to support the regression testing of component-based software. *IEEE International Conference on Software Maintenance (ICSM'01),* 2001.

Rothermel G., Harrold M. J. Empirical studies of a safe regression test selection technique *IEEE Transactions on Software Engineering*, Vol. 24, No. 6, 1998.