Regressiotestauksen tehostaminen

Juha Holopainen
Selvitys
Tietojenkäsittelytieteen laitos
Kuopion yliopisto
02.09.2004

SISÄLLYS

1 JOHDANTO 3

2 REGRESSIOTESTAUKSEN KÄSITTEET 4

2.1 Testauksen määritelmä 4
2.2 Regressiotestauksen määritelmä 5
2.3 Testauksen ja regressiotestauksen erot 6
2.4 Regressiotestauksen käyttötilanteet 6

3 REGRESSIOTESTAUS KÄYTÄNNÖSSÄ 9

3.1 Regressiotestauksen vaiheet 9
3.2 Yleinen toimintamalli 11

4 REGRESSIOTESTAUSTA TEHOSTAVAT METODOLOGIAT 12

4.1 Yleistä 12
4.2 Regressiotestien valintatekniikat 12
4.2.1 Yleistä asiaa valintatekniikoista 12
4.2.2 Yleinen malli regressiotestien valintaan 13
4.2.3 Erilaisia regressiotestien valintatekniikoita 14
4.2.4 Kriteerit valintatekniikoiden vertailemiseksi 15
4.2.5 Gravesin tekemä vertailu valintatekniikoista 16
4.3 Testitapausten priorisointitekniikat 17
4.3.1 Yleistä asiaa priorisointitekniikoista 17
4.3.2 Erilaisia priorisointitekniikoita 17
4.3.3 Elbaumin tekemä vertailu priorisointitekniikoista 18
4.4 Testijoukon harventamistekniikat (reduction, minimization) 19
4.4.1 Yleistä asiaa harventamistekniikoista 19
4.4.2 Tehdyt tutkimukset 19
4.5 Tutkimus rakeisuuden ja testitapausten ryhmittelyn vaikutuksesta eri metodologioihin 20
4.5.1 Tutkimukseen valitut tekniikat 21
4.5.2 Tutkimustulokset 22

5 KOMPONENTTIPOHJAISTEN OHJELMISTOJEN REGRESSIOTESTAUS 27

5.1 Metasisällön hyödyntäminen 27
5.1.1 Artikkelissa esitetty esimerkkiohjelma 28
5.1.2 Koodiin pohjautuvat metasisällöt testitapausten valintaan 29
5.1.3 Määrityksiin pohjautuvat metasisällöt testitapausten valintaan 32
5.1.4 Arviointi 36
5.2 Regressiotestaus UML-kaavioita apuna käyttäen 37
5.2.1 Regressiotestaus korjaavan ylläpitoprosessin aikana 38
5.2.2 Regressiotestaus täydentävän ja mukautuvan ylläpitoprosessin aikana 41

6 OLIO-OHJELMIEN REGRESSIOTESTAUS 45

6.1 Kehitettyjä tekniikoita 45
6.1.1 Yleinen regressiotestien valintatekniikoiden käyttämä toimintamalli 45
6.2 Java-ohjelmointikielelle kehitetty regressiotestien valintatekniikka 47
6.2.1 Muuttujien ja olioiden tyypit 48
6.2.2 Sisäiset tai ulkoiset metodit 48
6.2.3 Sisäiset metodikutsut 49
6.2.4 Ulkoiset metodikutsut 50
6.2.5 Poikkeusten käsittely 51
6.2.6 Läpikäyntialgoritmi 52
6.2.7 Retest-valintajärjestelmä 53
6.2.8 Tekniikasta tehty tutkimus 54

7 REGRESSIOTESTAUKSEN AUTOMATISOINTI 56

8 LÄHTEET 57

1 JOHDANTO

Regressiotestaus on erittäin 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 keskiverroin noin seitsemässä prosentissa ohjelmistoon tehdyistä korjauksista (Jones, 1997).

Yleisin käytössä oleva tapa suorittaa regressiotestaus on testata ohjelmaa kaikilla aiemmin tehdyillä testitapauksilla. Tässä selvityksessä esitellään erilaisia keinoja testitapausten valitsemiseksi ja regressiotestauksen tehostamiseksi. Selvityksen toisessa luvussa kerrotaan, mitä regressiotestaus tarkoittaa sekä selvennetään muita regressiotestaukseen liittyviä käsitteitä. Kolmannessa luvussa kerrotaan, mitä vaiheita regressiotestaus käytännössä sisältää. Neljännessä luvussa kerrotaan regressiotestausta tehostavista metodologioista, joita ovat regressiotestien valintatekniikat, testitapausten priorisointitekniikat ja testijoukon harventamistekniikat. Luvun lopussa on esitetty tutkimus testijoukon koostumuksen vaikutuksesta edellä mainittuihin metodologioihin. Viidennessä luvussa keskitytään tarkastelemaan kahta, komponenttipohjaisten ohjelmistojen regressiotestauksen tehostamiseksi kehitettyä tekniikkaa, eli metasisällön sekä UML-kaavioiden hyödyntämistä. Kuudennessa luvussa kerrotaan olio-ohjelmien regressiotestauksesta ja esitetään Java-ohjelmointikielelle kehitetty regressiotestien valintatekniikka. Viimeisessä, eli seitsemännessä luvussa kerrotaan lyhyesti regressiotestauksen automatisoinnista.

2 REGRESSIOTESTAUKSEN KÄSITTEET

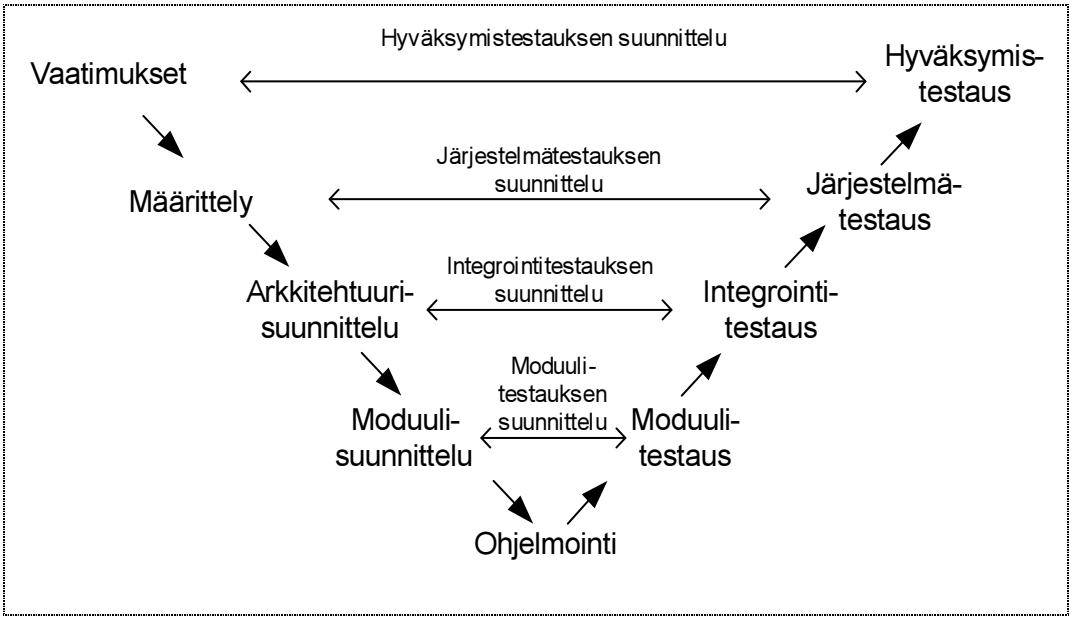
2.1 Testauksen määritelmä

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.

Ohjelman testaamiseksi sille luodaan *testitapauksia* (test case). Jokainen testitapaus sisältää ohjelmalle annettavat syötteet sekä ohjelman odotetun lopputilan annetuilla syötteillä. Määritellyistä testitapauksista kootaan *testijoukkoja* (test suite) ohjelman tai sen osien testaamiseksi. Testijoukko on joukko testejä, jotka eivät ole missään tietyssä järjestyksessä. Testijoukon sisältämistä testitapauksista kootaan *testijonoja,* joissa testien järjestys on määrätty. Testausprosessissa ohjelmaa testataan jokaisella testijoukon sisältämällä testitapauksella: Jotta testi voitaisiin ajaa, täytyy ohjelman suoritusympäristö ensin alustaa. Seuraavaksi ohjelma ajetaan testitapauksessa määritellyillä syötteillä ja lopuksi verrataan saatua lopputilaa odotettuun lopputilaan. Mikäli saatu lopputila 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 sen jälkeen tutkia kuinka suuren osan ohjelman entiteeteistä testitapaus käy läpi, kun testitapaus ajetaan. *Entiteetillä* tarkoitetaan jotakin ohjelman osaa, kuten esimerkiksi lauseita (lausekattavuus) tai ohjelman sisältämiä polkuja (polkukattavuus). Testijoukon kattavuudella tarkoitetaan sen sisältämien testitapausten joukon kattavuutta ohjelmassa. Kattavuuden perusteella voidaan myös päättää ohjelman riittävästä testauksesta, eli voidaan päättää, että testaus lopetetaan tiettyjen kattavuusehtojen täyttyessä.

Tässä selvityksessä mainitaan useaan otteeseen 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.



Kuva 1. Testauksen V-malli.

2.2 Regressiotestauksen määritelmä

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).

Regressiotestauksella tarkoitetaan muutetun ohjelman uudelleentestausta pyrkimyksenä selvittää, onko ohjelmaan tehty muutos aiheuttanut virheitä. Mikäli muutoksen seurauksena syntyy virheitä, sanotaan ohjelman regressoituvan eli taantuvan. Regressiotestauksella tarkoitetaan ohjelmaan tehdyn korjauksen toimivuuden tarkistamista ja sen tarkistamista, että korjaus ei ole sotkenut muiden ohjelman osien toimintaa (Kaner ym., 1999, s. 49-50).

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 sellaista testitapausta, jonka perustason ohjelma on läpäissyt, ja jonka deltakoosteen odotetaan läpäisevän. Testitapaus, jonka perustason ohjelma on läpäissyt, mutta jota deltakooste ei läpäise, paljastaa *regressiovirheen* (regression fault) (Binder, 1999, s.756).

Testitapauksen uudelleenkelpoistamisen (test case revalidation) tarkoituksena on löytää epäkelvot testitapaukset testijoukosta. Uudelleenkelpoistamisprosessissa tarkastellaan

testitapauksen syötettä ja odotettua tulosta. Mikäli syöte on kelvoton, testitapaus voidaan poistaa testijoukosta tai käyttää jatkossa negatiivisena testitapauksena. Kelvoton syöte voi ilmetä, jos testattavasta ohjelmasta esimerkiksi poistetaan tietty toiminto. Mikäli syöte on kelvollinen, mutta tulos on kelvoton, testitapauksen odotettu tulos pitää muuttaa vastaamaan nykyisiä olosuhteita. Testijoukkoon pitää tarvittaessa lisätä uusia testitapauksia riittävän testauskattavuuden varmistamiseksi (Onoma ym., 1998). Tässä selvityksessä huomio kiinnitetään vanhojen testitapausten uudelleenkäyttämiseen ja jätetään uusien testitapausten lisääminen huomiotta.

Uudelleenkelpoistaminen täytyy tehdä manuaalisesti, joten se vie paljon aikaa. Uudelleenkelpoistaminen voidaan tehdä ennen testausta tai testauksen 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.

2.3 Testauksen ja regressiotestauksen erot

Tavallinen testaus suoritetaan vain kerran jokaista järjestelmän versiota kohti, mutta regressiotestaus suoritetaan useaan kertaan järjestelmän elinkaaren aikana. Tavallisen testauksen syötteitä ovat määritykset, toteutukset sekä testisuunnitelmat, kun taas regressiotestauksessa käytetään (mahdollisesti) muutettuja määrityksiä, muutettua toteutusta ja vanhaa testisuunnitelmaa (jota tullaan päivittämään regressiotestauksessa). Usein järjestelmää kehitettäessä ei projektisuunnitelmassa huomioida regressiotestauksen vaatimaa aikaa ja resursseja, mistä johtuen regressiotestauksen tehokas suorittaminen on entistäkin tärkeämpää. (Paakki, 2000)

2.4 Regressiotestauksen käyttötilanteet

Sen jälkeen kun testattava kohde on testattu alkuperäisellä testijoukolla, voidaan regressiotestaus suorittaa missä tahansa kehitysvaiheessa (kuva 1). 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 tai luokkia.

- *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, mukautuvaa tai ennakoivaa (Gao ym., 2003):
  - *Korjaavan ylläpitoprosessin* tarkoituksena on korjata löydetyt virheet. Komponenttipohjaisissa järjestelmissä korjauksia tehdään yleensä yksittäisiin komponentteihin. Rajapinnat, komponentin yleinen rakenne ja määritykset pysyvät yleensä muuttumattomina.
  - *Täydentävän ylläpitoprosessin* tarkoituksena on parantaa tuotteen suorituskykyä ja muita laatutekijöitä. Prosessin 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.
  - *Mukautuvassa ylläpitoprosessissa* muutetaan tuotetta vastaamaan muuttuneen toimintaympäristön vaatimuksiin. Esimerkiksi Windows-ympäristössä toimiva ohjelma muutetaan toimimaan Linux-ympäristössä.
  - *Ennakoivan ylläpitoprosessin* tarkoituksena on ennakoivasti muuttaa tuotetta, ennen kuin virheitä pääsee tapahtumaan.

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 muutos on voinut vaikuttaa.

- Kun järjestelmään liitetään uusi toiminnallisuus, 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 moduuleiden kanssa integrointitasolla. Tällaista menetelmää kutsutaan *Monitasoiseksi Regressiotestaukseksi* (Multi-Level Regression Testing (MLRT)). MLRT-menetelmässä täysin samat testitapaukset voivat tulla läpikäytyä useaan kertaan, esimerkiksi kerran yksikkötasolla ja toistamiseen integrointitasolla. Tämä ei ole ajantuhlausta, 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. (Onoma ym., 1998, s. 85-86)

3 REGRESSIOTESTAUS KÄYTÄNNÖSSÄ

3.1 Regressiotestauksen vaiheet

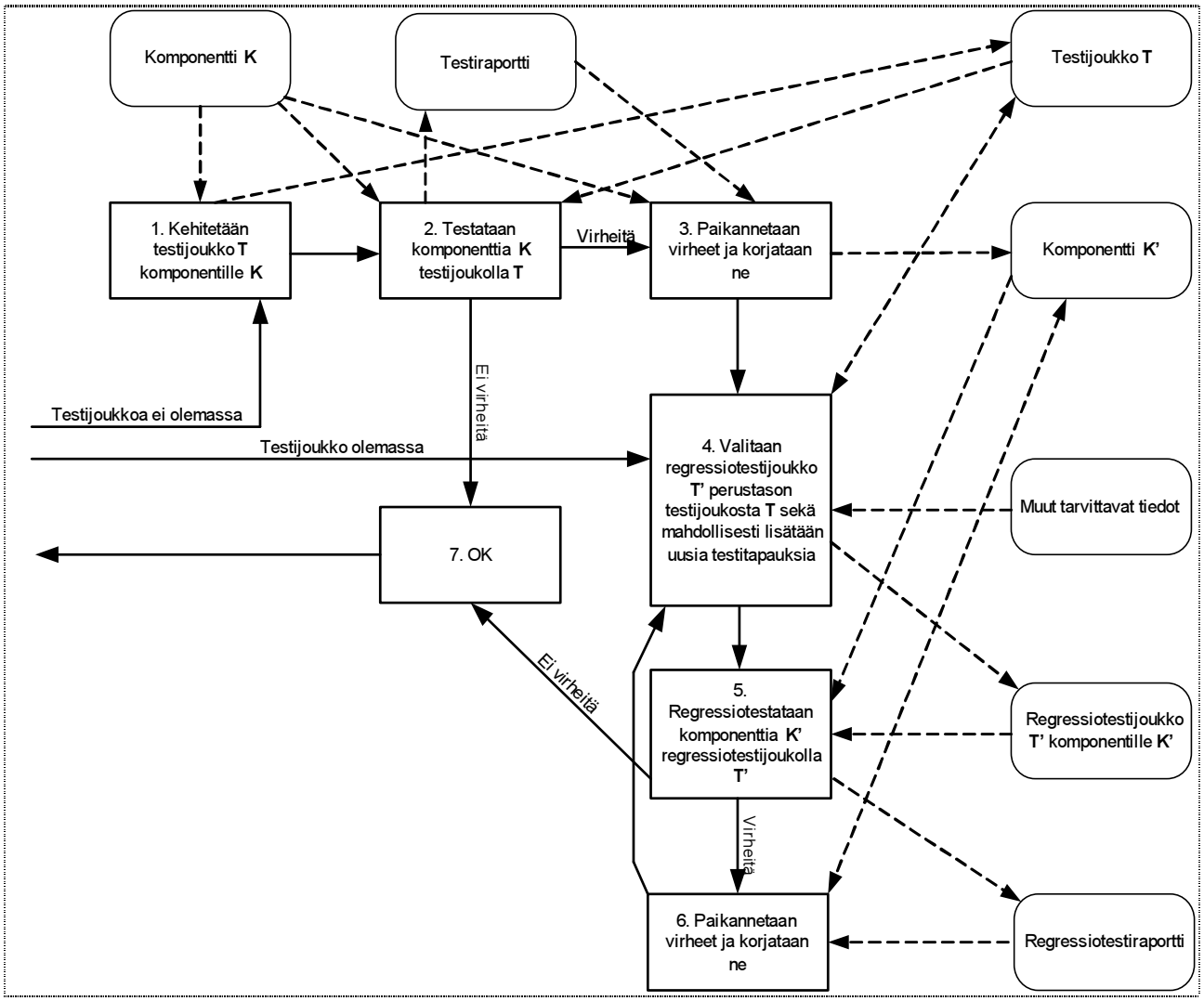
Binder on kirjassaan (Binder, 1999, s.761) listannut seuraavat regressiotestauksen päävaiheet, jotka ovat samoja, suoritettiinpa regressiotestaus missä vaiheessa ohjelmiston kehitystä tahansa. Aluksi poistetaan alkuperäisestä testijoukosta käyttökelvottomat testitapaukset. Käyttökelvottomalla testitapauksella 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. Toinen vaihtoehto on valita regressiotestijoukkoon kaikki alkuperäisen testijoukon testitapaukset. Tämän jälkeen luodaan testauskokoonpano ja ajetaan regressiotestijoukko. Lopuksi huomioidaan läpäisemättömät testit ja ryhdytään tarvittaviin toimenpiteisiin.

Olen esittänyt kuvassa 2 oman tulkintani regressiotestauksen eri vaiheista ja eri vaiheisiin liittyvistä syötteistä ja tulosteista. Tummennetut nuolet merkitsevät siirtymiä eri tilojen (laatikot) välillä. Pyöreäkulmaisilla laatikoilla merkitään eri tilojen syötteitä tai tuloksia. Pyöreäkulmainen laatikko X on syötteenä tilalle Y, jos laatikosta X on piirretty katkoviivalla nuoli laatikkoon Y. Jos katkoviivanuoli on piirretty laatikosta Y laatikkoon X, on laatikko X syntynyt tai muuttunut tilalaatikossa Y tehtyjen toimenpiteiden tuloksena.

1. Kun komponenttia **K** testataan ensimmäisen kerran, sille ei ole olemassa testijoukkoa. Tällöin kehitetään testijoukko **T** komponentille **K**.
2. Komponenttia **K** testataan kehitetyllä testijoukolla **T**. Jos testauksessa ei löydetä virheitä, testaus voidaan lopettaa (siirrytään tilaan 7). Mikäli virheitä löytyy, siirrytään paikantamaan virheitä ja korjaamaan niitä (tila 3).
3. Löytyneet virheet paikannetaan ja korjataan, jonka seurauksena komponentista **K** syntyy muutettu versio **K'**.
4. Valitaan alkuperäisen testijoukon **T** sisältämistä testitapauksista regressiotestijoukko **T'** komponentin **K'** regressiotestaamiseksi. "Muut tarvittavat tiedot" -syötelaatikko ilmentää eri valintatekniikoiden tarvitsemia tietoja, joiden avulla testit voidaan valita. Esimerkiksi kattavuuteen perustuva valintatekniikka tarvitsisi kattavuustiedot ohjelmasta ja testitapauksista. Tässä vaiheessa voidaan luoda uusia testitapauksia testaamaan uutta tai

muuttunutta toiminnallisuutta. Mikäli uusia testitapauksia lisätään regressiotestijoukkoon **T'**, täytyy nämä testitapaukset lisätä myös alkuperäiseen testijoukkoon **T.**

5. Regressiotestataan komponenttia **K'** regressiotestijoukolla **T'**. Jos virheitä ei löytynyt, voidaan testaus lopettaa (siirrytään tilaan 7).
6. Regressiotestauksessa löytyneet virheet paikannetaan ja korjataan. Tällöin saadaan komponentista uusi muutettu versio **K'**. Tämän jälkeen siirrytään valitsemaan testitapauksia regressiotestijoukkoon (tila 4). Jos komponenttiin tehdään tulevaisuudessa muutoksia, voidaan testauksessa käyttää hyväksi aiemmin luotua testijoukkoa **T**, eli siirtyä suoraan tilaan 4.
7. Testaus on tältä osin päättynyt ja voidaan siirtyä eteenpäin.



Kuva 2. Testauksen ja regressiotestauksen vaiheet.

3.2 Yleinen toimintamalli

Suurin osa Onoman tutkimusryhmän tutkimista yrityksistä noudattaa regressiotestauksessaan seuraavia askelia (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 muutoksen jälkeiseen testaukseen 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.
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 muutos, joka johti regressiovirheen syntymiseen. Onoma huomasi tutkimuksissaan, että joissakin harvoissa tapauksissa virhe yksinkertaisesti jätettiin huomiotta. Tämä tapahtui yleensä silloin, kun löydetty virhe ei ollut vakava ja sen korjaamiseen ei riittänyt aikaa.

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, mutta usein regressiotestaukseen valitaan mukaan kaikki olemassa olevat testitapaukset. Useissa tutkimuksissa on tutkittu kuinka regressiotestien valintaa ja regressiotestausta yleensä voitaisiin tehostaa.

Tässä luvussa esitellään eri metodologiat, joiden alle regressiotestausta tehostavat tekniikat kuuluvat. Samalla tulee esitellyksi joitakin yleisimpiä tekniikoita. Osion lopussa on esitetty tutkimus, jossa tutkittiin testijoukon rakeisuuden ja testijoukon sisältämien testitapausten samankaltaisuuden vaikutusta eri metodologioihin.

Regressiotestauksen kustannustehokkuuden parantamiseksi on olemassa useita erilaisia metodologioita. Näiden joukosta voidaan löytää kolme jo olemassa olevia testitapauksia hyödyntävää metodologiaa:

- *Regressiotestien valintatekniikat* (regression test selection) käyttävät uudelleen vain osan olemassa olevista testitapauksista.
- *Testitapausten priorisointi* (test case prioritization) -metodologia järjestävää testitapaukset niin, että testauksen kannalta hyödyllisimmät testitapaukset suoritetaan ensin.
- *Testijoukon harventamiseen* (test-suite reduction, test-suite minimization) tähtäävä metodologia pyrkii alentamaan tulevien regressiotestausvaiheiden kustannuksia poistamalla testitapauksia testijoukosta pysyvästi.

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. Valintatekniikoita on olemassa monenlaisia ja niistä jokaisella on sekä hyvät

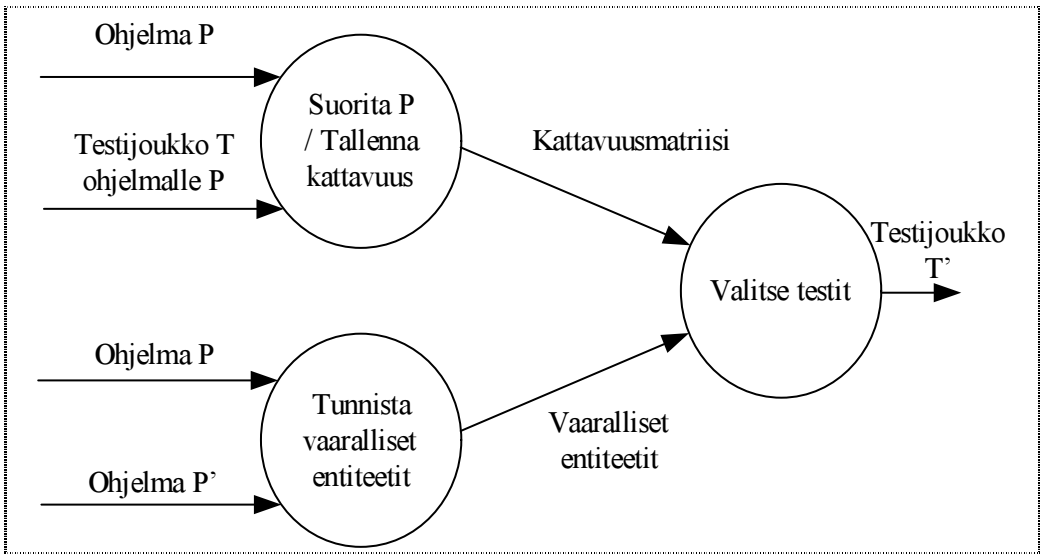
että huonot puolensa. Regressiotestitapausten valintatekniikan valinta riippuu tilanteesta, jossa regressiotestausta suoritetaan.

Ei ole olemassa yhtä valintatekniikkaa, jota voisi käyttää kaikissa tilanteissa, koska yksikään tämänhetkisistä valintatekniikoista ei ole täydellinen jokaisella osa-alueella. Mitä kattavampi testijoukosta 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).

4.2.2 Yleinen 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. Tässä selvityksessä keskitytään tarkemmin vain ensimmäiseen ongelmaan, eli ohjelman **P'** testaamiseen tarkoitetun testijoukon **T'** kokoamiseen testijoukon **T** sisältämistä testitapauksista.

Kuvassa 3 on esitetty tavallinen regressiotestien valintatekniikka. Ensin ohjelmaa **P** testataan testijoukolla **T**, jolloin tekniikka kerää tavanomaisten tietojen (testi läpi/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)



Kuva 3. Yleinen malli regressiotestien valintaan (Harrold ym., 2001).

4.2.3 Erilaisia regressiotestien valintatekniikoita

Regressiotestien valintatekniikoita tarkasteltaessa pitää ottaa huomioon kaksi seikkaa. Ensinnäkin, miten tunnistaa ohjelman vaaralliset entiteetit, eli ohjelman osat, joihin ohjelmaan tehty muutos on voinut vaikuttaa. Vaarallisten entiteettien tunnistamiseksi on kehitetty useita lähestymistapoja. Esimerkiksi palomuuritekniikoiden (firewall approach) pyrkimyksenä on löytää ohjelman vaaralliset lauseet ja funktiot, kun taas tietovirtatekniikat (data-flow approach) ja ohjelman viipalointitekniikat (program-slicing approach) pyrkivät löytämään vaaralliset määrittely-käyttö (definition-use) -parit. Määrittely-käyttö -parilla tarkoitetaan, että tietty muuttuja on ensin määritelty ohjelmassa ja myöhemmin kyseistä muuttujaa on käytetty ohjelmassa. Muuttujan määrittelyn ja käytön välissä saa olla (ja yleensä onkin) muita lauseita, mutta välissä olevat lauseet eivät saa käsitellä kyseistä muuttujaa millään tavalla.

Edellä mainittuja tekniikoita voidaan soveltaa joko staattisesti tai dynaamisesti. Staattisessa lähestymistavassa vaaralliset entiteetit pyritään löytämään 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.

Toinen 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 technique). Minimointitekniikkaan perustuvien regressiotestien valintatekniikoiden pyrkimyksenä on luoda kaikki ohjelman vaaralliset entiteetit

kattava testijoukko mahdollisimman vähillä testitapauksilla. Esimerkiksi Fischerin esittämässä tekniikassa (Fischer ym., 1981) käytetään lineaarisia yhtälöitä testijoukkojen kokoamiseen. Hartmann tarjoaa artikkelissaan (Hartmann ym., 1990) lisäyksiä Fischerin tekniikkaan.

Kattavuuteen perustuvat tekniikat (mm. Bates ym., 1993; Binkley, 1995). Nämä tekniikat valitsevat regressiotestitapaukset jonkin kattavuuskriteerin perusteella. Toisin kuin minimointitekniikat, jotka valitsevat minimimäärän testitapauksia, kattavuuteen perustuvat tekniikat valitsevat kaikki vaarallisia entiteettejä testaavat testitapaukset mukaan regressiotestijoukkoon. Esimerkiksi *tietovirtatekniikkaan* (dataflow technique) perustuvat valintatekniikat valitsevat regressiotestijoukkoon kaikki sellaiset testitapaukset, jotka testaavat ohjelman muuttuneita määrittely-käyttö -pareja. Tietyt tekniikat testaavat myös poistettuja määrittely-käyttö -pareja.

Turvalliset tekniikat (safe techniques) (mm. Chen ym., 1994; Rothermel ym., 1997a). Turvalliseksi valintatekniikaksi nimitetään sellaista tekniikkaa, joka valitsee alkuperäisestä testijoukosta kaikki sellaiset testit, jotka voivat mahdollisesti paljastaa virheen. Suurinta osaa valintatekniikoista ei ole suunniteltu turvallisiksi, mukaan lukien minimointi- ja tietovirtatekniikka. (Graves ym., 2001)

4.2.4 Kriteerit valintatekniikoiden vertailemiseksi

Tässä selvityksessä tarkastellaan valintatekniikoiden eroja Gregg Rothermelin ja Mary Jean Harroldin esittämien kriteerien pohjalta (Rothermel ym., 1994):

- *Inklusiivisuus* kertoo kuinka hyvin valintatekniikka osaa valita alkuperäisestä testijoukosta virheitä mahdollisesti paljastavia testejä mukaan regressiotestijoukkoon. Turvallinen valintatekniikka ottaa mukaan kaikki sellaiset testit, jotka voivat paljastaa virheen. Turvallisella valintatekniikalla valitun testijoukon kyky löytää ohjelmasta virheitä riippuu siitä, miten huolellisesti alkuperäinen testijoukko on koottu.
- *Tarkkuus* 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* 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. 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* kertoo, kuinka useassa tilanteessa valintatekniikan käyttö on mahdollista.

4.2.5 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 yhdeksän C-kielisen ohjelman eri versioita. Jokaisella ohjelmalla oli valmiina useita erilaisia testijoukkoja. 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 inklusiivisyys kasvoi, mutta hitaammin.

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).

4.3 Testitapausten priorisointitekniikat

4.3.1 Yleistä asiaa priorisointitekniikoista

Testauksen tarkoituksena on paljastaa ohjelman sisältämät virheet. Jos virheet pystytään havaitsemaan testauksen alkuvaiheessa, säästetään aikaa ja resursseja, koska virheiden korjaaminen voidaan tällöin aloittaa aiemmin. Virheiden aikaisen löytymisen tärkeys korostuu isojen testijoukkojen kohdalla. Testitapausten priorisoinnista on hyötyä myös siinä tapauksessa, että testausprosessi joudutaan syystä tai toisesta keskeyttämään, jolloin loppupään testitapauksia ei ehditä käymään läpi.

Testitapausten priorisointitekniikat auttavat testaajia saamaan testitapaukset sellaiseen järjestykseen, että ohjelman sisältämät virheet löytyvät mahdollsiimman aikaisessa vaiheessa. Priorisointitekniikat eivät poista testijoukosta yhtään testiä, vaan niiden tarkoituksena on järjestää testitapaukset testaustehokkuuden maksimoimiseksi (Srivastava ym., 2002). Tutkittujen priorisointitekniikoiden tehokkuus riippuu muun muassa tekniikoiden kohteena olevan testijoukon ja testattavan ohjelman ominaisuuksista. Tähän päivään mennessä ei ole saatu kehitettyä menetelmää, joka kertoisi mikä olemassa olevista priorisointitekniikoista soveltuisi parhaiten tiettyyn tapaukseen.

Asiasta on tehty tutkimuksia, joiden tulokset eivät ole täysin yhdenmukaisia. Joissain tutkimuksissa (Elbaum ym., 2002; Rothermel ym., 2001; Srivastava ym., 2002) tutkitut menetelmät ovat lisänneet virheiden löytymisnopeutta merkittävästi, mutta toisissa tutkimuksissa (Elbaum ym., 2001; Elbaum ym., 2003) virheiden löytymisnopeus on vaihdellut suuresti ohjelman ja testijoukon ominaisuuksien mukaan. (Elbaum ym., 2004)

4.3.2 Erilaisia priorisointitekniikoita

Erilaisia testitapausten priorisointitekniikoita on olemassa monenlaisia. Artikkelissa (Elbaum ym., 2002) on listattu 16 erilaista priorisointitekniikkaa sekä kaksi tekniikoiden vertailua helpottamaan tehtyä tekniikkaa.

4.3.3 Elbaumin tekemä vertailu priorisointitekniikoista

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

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 aiemmin tehdyissäkin 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, testijoukon 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)

4.4 Testijoukon harventamistekniikat (reduction, minimization)

4.4.1 Yleistä asiaa harventamistekniikoista

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 ohjelmakomponentteja. Testijoukosta voidaan poistaa tällaisia testitapauksia pysyvästi ilman, että testijoukon kattavuus laskee. Tällaisten testitapausten etsimistä ja poistamista kutsutaan testijoukon harventamiseksi (test-suite minimization/reduction)

Testijoukon harventamisella voidaan vähentää testijoukon ajamis-, kelpoistamis- ja ylläpitokustannuksia, mutta toisaalta testijoukon kyky havaita virheitä todennäköisesti vähenee. (Rothermel ym., 1998)

4.4.2 Tehdyt tutkimukset

Testijoukon harventamisen tuomista hyödyistä on tehty joitakin tutkimuksia. Yhdessä 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.

Toisessa tutkimuksessa (Rothermel ym., 1998) tutkittiin joukkoa yhden virheen sisältämiä ohjelmaversioita Harrold-Gupta-Soffa -harvennustekniikkaa 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 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) tutkitaan erityisen vaativan modified condition/decision coverage (MC/DC)-kattavuuskriteerin täyttävän testijoukon harventamista. Blackin työryhmän tekemässä tutkimuksessa (Black ym., 2004) tutkitaan, kuinka testijoukko voitaisiin minimoida samalla, kun testijoukon kyky havaita virheitä maksimoitaisiin.

4.5 Tutkimus rakeisuuden ja testitapausten ryhmittelyn vaikutuksesta eri metodologioihin

Luvuissa 4.2, 4.3 ja 4.4 mainittuihin 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 samankaltaisuuden 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 samankaltaisuus (test input grouping) kertoo, miten samankaltaisia kunkin testitapauksen syötteet ovat kyseisen testitapauksen muiden syötteiden kanssa. Artikkelin kirjoittajat ovat jakaneet löytämänsä alkeistason syötteet eri lokeroihin sen mukaan, mitä ohjelman toimintoa ne testaavat. Lokerossa olevat alkeistason syötteet on sen jälkeen koostettu testijoukon rakeisuuden määrittämän kokoisiksi testitapauksiksi. Testijoukon sisältämien testisyötteiden samankaltaisuus ilmoitetaan prosenttilukuna sen mukaan, miten homogeenisiä testitapausten sisältämät alkeistason syötteet ovat. Jos esimerkiksi tietylle ohjelman toiminnolle ei löydy testijoukon rakeisuuden edellyttämää määrää alkeistason syötteitä, laskee kyseisen testitapauksen ja myös testitapauksen sisältämän testijoukon testisyötteiden samankaltaisuus.

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.

4.5.1 Tutkimukseen valitut tekniikat

Rothermelin tutkimuksessa tutkittiin testijoukon rakeisuuden ja testisyötteiden samankaltaisuuden vaikutusta regressiotestauksessa käytettävien metodologioiden kustannustehokkuuteen. Testaa kaikki uudelleen -tekniikka toimi kontrollitekniikkana, johon muita verrattiin. Regressiotestien valintatekniikoista Rothermel otti tutkimukseensa mukaan neljä eri tekniikkaa:

- *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 toimii Rothermelin tutkimuksen kontrollitekniikkana.
- *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, 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 muutettu osa on ainakin yhden testitapauksen testaama.

Testitapausten priorisointi -metodologiasta Rothermel valitsi tutkimukseensa kolme priorisointitekniikkaa:

- *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 muuttuneita 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.

Testijoukon harventaminen -metodologiasta Rothermel valitsi tutkimukseensa kaksi tekniikkaa:

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

4.5.2 Tutkimustulokset

Kuten tieteellisissä tutkimuksissa aina, täytyy näitäkin tutkimustuloksia tarkasteltaessa ottaa huomioon joitakin tulosten yleistettävyyteen vaikuttavia asioita. Tutkimuskohteina käytettiin vain kahta erilaista ohjelmaa, mikä heikentää tulosten yleistettävyyttä. Toisaalta ohjelmista saadut

tulokset olivat verrattain yhteneviä, mikä antaa viitteitä päinvastaisesta. Yleistettävyyttä parantaa myös se, että tutkimukseen valittiin suhteellisen suuri otos todellisia, perättäin julkaistuja versioita tutkituista ohjelmista. Kooltaan tutkitut ohjelmat olivat samantapaisia muiden yleisessä käytössä olevien ohjelmien kanssa.

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., 1997b; Rothermel ym., 1998) 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 regressiotestauksen testaa kaikki uudelleen-, regressiotestien valinta- sekä regressiotestien harvennus-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-harvennustekniikkaa emp-server -ohjelmaan.

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

4.5.2.1 Testaa kaikki uudelleen-tekniikka

Rothermelin tutkimuksessaan saamista tutkimustuloksista 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)

4.5.2.2 Regressiotestien valintatekniikat

Turvalliset regressiotestien valintatekniikat eivät jätä 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 tekniikkaa 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 samankaltaisuudella 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.

4.5.2.3 Testijoukon harventamistekniikat

GHS-testijoukon harvennustekniikka ja minimointiin pyrkivä regressiotestien valintatekniikka ovat testitapausten valikoinnissa periaatteiltaan samanlaisia, eli molemmat valitsevat minimimäärän testitapauksia. Ero on siinä, että minimointitekniikka valitsee käytettävät testitapaukset muutettujen ohjelman osien perusteella, kun taas testijoukon harvennustekniikka käy läpi kaikki ohjelman osat.

GHS-harvennustekniikkaa käytettäessä testijoukon rakeisuuden vaikutukset olivat lähes samanlaisia kuin regressiotestien valintatekniikoista minimointitekniikan kohdalla. Rakeisuuden vaikutukset testijoukon suoritusaikaan olivat eri ohjelmien kohdalla jonkin verran erilaisia: emp-server

-ohjelman tapauksessa testijoukon läpiviemisaika laski, ja bash-ohjelman tapauksessa testijoukon läpiviemisaika nousi. Virheiden löytymisen todennäköisyys laski bash-ohjelman kohdalla enemmän kuin emp-server -ohjelman kohdalla. Yksi GHS-harvennustekniikan eroista minimointitekniikan tuloksiin verrattuna, oli testisyötteiden samankaltaisuuden merkittävä vaikutus testijoukon kykyyn havaita virheitä. Samankaltaisten testisyötteiden sijoittaminen samaan testitapaukseen helpotti virheiden löytymistä merkittävästi verrattuna siihen, että testisyötteet olisi sijoitettu satunnaisesti eri testitapauksiin. Bash-ohjelman tapauksessa testitapausten ryhmittelyllä ei ollut vaikutusta suuntaan

tai toiseen. Rothermelin tutkimuksessaan saamista tuloksista voidaan päätellä testisyötteiden oikeanlaisella ryhmittelyllä olevan merkitystä joissain tapauksissa.

4.5.2.4 Testitapausten priorisointitekniikat

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.

4.5.2.5 Muutokset testitapausten virheiden havaitsemiskykyyn

Testaa kaikki uudelleen -tekniikkaa käyttämällä ei rakeisuuden muuttamisella huomattu olevan vaikutusta testijoukon kykyyn havaita virheitä. Harvennettujen testijoukkojen ja valittujen osatestijoukkojen kohdalla testijoukkojen virheiden löytymistehokkuus laski.

5 KOMPONENTTIPOHJAISTEN OHJELMISTOJEN REGRESSIOTESTAUS

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 tarjoajat. Komponentin käyttäjillä ei ole tietoa komponentin toteutuksesta, vaan he ovat riippuvaisia komponenttien kokoajien tarjoamasta tiedosta. Tämä vaikeuttaa komponenttisysteemin testausta. (Gao ym., 2003)

5.1 Metasisällön hyödyntäminen

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 rakentaja 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. Tällaista tietoa kutsutaan komponentin *metasisällöksi*. Metasisältö sisältää tietoa (metadata) komponentista sekä metodeja (metamethods) tiedon keräämiseen. Tällä hetkellä komponenteista jaettu tieto on yksipuolista, eikä sitä voi käyttää apuna regressiotestauksessa.

Tässä luvussa käsitellään Orson tutkimusta (Orso ym., 2001), jossa esitetään kaksi lähestymistapaa metasisällön esittämiselle ja käyttämiselle komponentin regressiotestauksessa tarvittavien testitapausten valinnassa. Toinen esiteltävistä lähestymistavoista käyttää ohjelmakoodia ja toinen ohjelman määrittelyjä. Havainnollisuuden vuoksi esitellään esimerkkiohjelma, joka käyttää yhtä komponenttia. Java-kielellä kirjoitettujen VendingMachine-pääohjelman sekä Dispenser-komponentin koodi on esitetty kuvassa 4, mutta oletetaan että komponentin lähdekoodi ei ole näkyvissä pääohjelman kehittäjälle.

5.1.1 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, joka taas johtaa ohjelman virheelliseen toimintaan.

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 pitää käydä läpi uudelleen kaikki ohjelman testitapaukset. Seuraavaksi esitetään kaksi tekniikkaa, joiden avulla pääohjelman kehittäjä pystyy valitsemaan testijoukostaan vain tarvittavat testitapaukset.

```
 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 4. Esimerkkiohjelman ja -komponentin ohjelmakoodi (Orso ym., 2001).

5.1.2 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., 1997a; Rothermel ym., 2000), joista tähän on valittu käytettäväksi Rothermelin ja Harroldin kehittämää tekniikkaa (Rothermel ym., 1997a) 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 komponentin metodeja kutsuvat pääohjelman kaaret muuttuneiksi 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. Jotta regressiotestitapausten valintaprosessia voitaisiin tehostaa, tarvitaan komponentista 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 tehdyt 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 testitapausten määrä saatiin vähennettyä kahteen, kun se ilman metatietoa saadussa testijoukossa oli 20.

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

Testitapaus #	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)

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

5.1.3 Määrityksiin pohjautuvat metasisällöt testitapausten valintaan

Tässä esitetään Orson artikkelissaan esittämä metasisältöön perustuva tekniikka käytettäväksi määrityksiin pohjautuvien regressiotestitapausten valintatekniikoiden kanssa. Yksi tällainen tekniikka on kategorioihinjakotekniikka (category-partition method), joka tuottaa järjestelmän toiminnallisten osien testausmäärityksiä eli *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 5).

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 seuraavat kategoriat. Syötekategoriat credit ja selection sekä ympäristökategoria availability.
4. Jokainen kategoria jaetaan erillisiin ekvivalenssiluokkiin. Ekvivalenssiluokat sisältävät arvoja, joiden olisi tarkoitus vaikuttaa testattavaan järjestelmään samansuuntaisesti. Esimerkin tapauksessa kategoria credit on jaettu ekvivalenssiluokkiin nolla, riittämätön, riittävä ja liikaa (kuva 5).
5. Jokaiselle toiminnalliselle osalle tehdään *testimääritys*, joka koostuu seuraavista osista:
    - Luettelo kategorioista.
    - Luettelo jokaisen kategorian sisältämistä ekvivalenssiluokista.
    - Joukko *testikehyksiä*. Testikehykseen valintaan jokaisesta kategoriasta joko yksi tai ei yhtään ekvivalenssiluokkaa ja määritellään yksi looginen kokoonpano syöteparametreja ja ympäristötekijöitä, joilla toiminnallista osaa voidaan testata. Esimerkiksi kuvan 6 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 5 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. Tietty 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 6 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 testaavat testitapaukset.

Kuvassa 5 esitetään Disperser-komponentin *dispense*-metodin mahdolliset kategoriat, ekvivalenssiluokat ja niiden sisältämät rajoitukset. Kuvassa 6 on kuvan 5 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**			
	Syöte:		
		credit	
			-nolla [if availability = saatavissa]
			-riittämätön [if availability = saatavissa]
			-riittävä [if availability = saatavissa]
			-liikaa [if availability = saatavissa]
		selection	
			-kelvollinen [sisältö kelvollinen]
			-kelvoton [virhe]
	Ympäristötekijä:		
		availability	
			[if selection = kelvollinen]
			-saatavissa [sisältö saatavissa]
			-ei-saatavissa [if selection = kelvollinen] [virhe]

Kuva 5. 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 6. Dispenser-komponentin testikehykset (X-kirjaimella merkityt arvot ovat merkityksettömiä) (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		

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

DejaVu-ohjelmasta voitaisiin rakentaa metasisältöä ymmärtävä versio $\text{DejaVu}_{\text{MA}}$, joka voisi toimia saman toimintamallin (Orso ym., 2000) mukaan kuin koodiin pohjautuvan versionkin tapauksessa paitsi, että kaarien sijaan mitattaisiin 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 testikehysten kattavuustiedot (taulukko 3). Kun saadaan muutettu Dispenser-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

5.1.4 Arviointi

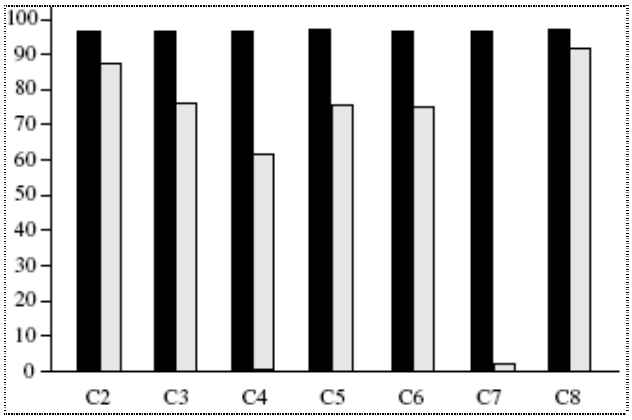
Tutkiakseen metasisällön hyötyjä komponenttipohjaisten ohjelmistojen testauksessa, Orso ym. suorittivat seuraavan tutkimuksen. Testattavaksi ohjelmaksi valittiin Java-kielisen SIENA-palvelimen kahdeksan eri versiota. SIENA (Scalable Internet Event Notification Architecture) koostuu kuudesta komponentista (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 7 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 paria metodia ja testitapausta. Muissa versioissa regressiotestijoukkojen kokoerot olivat pienempiä, erojen vaihdellessa välillä 6%-37%.

Kuvaa 7 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 4. 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 7. META- ja NOMETA-valintatekniikoilla valittujen testijoukkojen koko (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

Taulukko 4. META ja NOMETA-valintatekniikoiden valitsemien testijoukkojen suoritusajat (minuutit:sekunnit) (Orso ym., 2001).

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., 1997b) ovat kuitenkin osoittaneet valintaprosessiin kuluvan ajan olevan verrattain alhainen. Lisäksi pitää huomata, että taulukon 4 tulokset kertovat ainoastaan testijoukon ajamiseen kuluvan ajan, mutta testausprosessissa testitapaukset pitää myös kelpoistaa, eli varmistaa, että ne ovat käyttökelpoisia. Jos kelpoistamiseen kuluva aika otettaisiin huomioon, pienemmän testijoukon edut kasvaisivat.

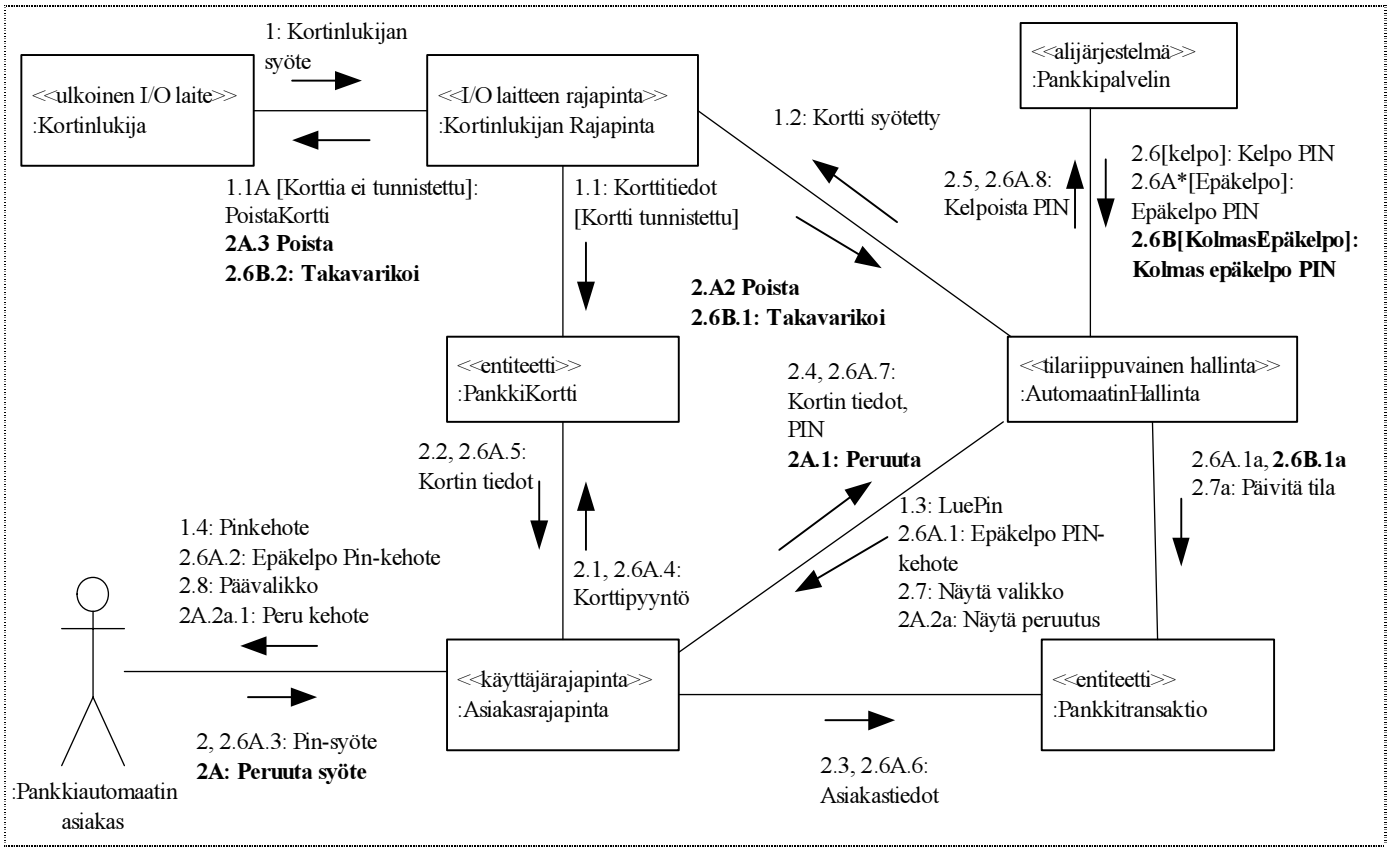
Suhteellisen pienestä testiohjelmasta johtuen saavutetut aikasäästöt voidaan laskea minuuteissa ja sekunneissa. Käytännössä isompien ohjelmien regressiotestaus saattaa kestää useita tunteja tai päiviä, joten tällöin metasisällön käyttö voi johtaa huomattaviin säästöihin.

5.2 Regressiotestaus UML-kaavioita apuna käyttäen

Tässä luvussa kerrotaan Gaon kirjassaan (Gao ym., 2003) esittämiä tapoja hyödyntää UML-kaavioita regressiotestauksessa. Ensin tarkastellaan regressiotestausta korjaavan ylläpitoprosessin aikana ja sen jälkeen täydentävän ja mukautuvan ylläpitoprosessin aikana.

5.2.1 Regressiotestaus korjaavan ylläpitoprosessin aikana

Kun komponentti on toimitettu asiakkaalle, ohjelmasta löydetään usein uusia virheitä. Näiden virheiden korjaamiseksi tähtäävää prosessia kutsutaan korjaavaksi ylläpitoprosessiksi. Koska komponentin käyttäjillä ei ole käytössään komponentin lähdekoodia, systeemi- ja integrointitason regressiotestauksen suorittamiseksi tarvitaan keino kuvata muutoksia. Gao neuvoo kirjassaan käyttämään UML-kielen (Unified Modeling Language) tarjoamia kaavioita muutosten kuvaamiseksi. Esimerkiksi 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 8. Yhteistyökaavio Pankkikortin PIN-numeron kelpoistamiseen liittyen (Gao ym., 2003).

Seuraavaksi luetellaan UML-kielen kaavioita ja Gaon kirjassaan kertomia neuvoja, kuinka kyseisiä kaavioita voidaan hyödyntää ohjelman lähdekoodiin tehtyjen muutosten kuvaamisessa. Esimerkkinä käytetään Gaon kirjassaan esittämiä kaavioita pankkiautomaatin toiminnasta.

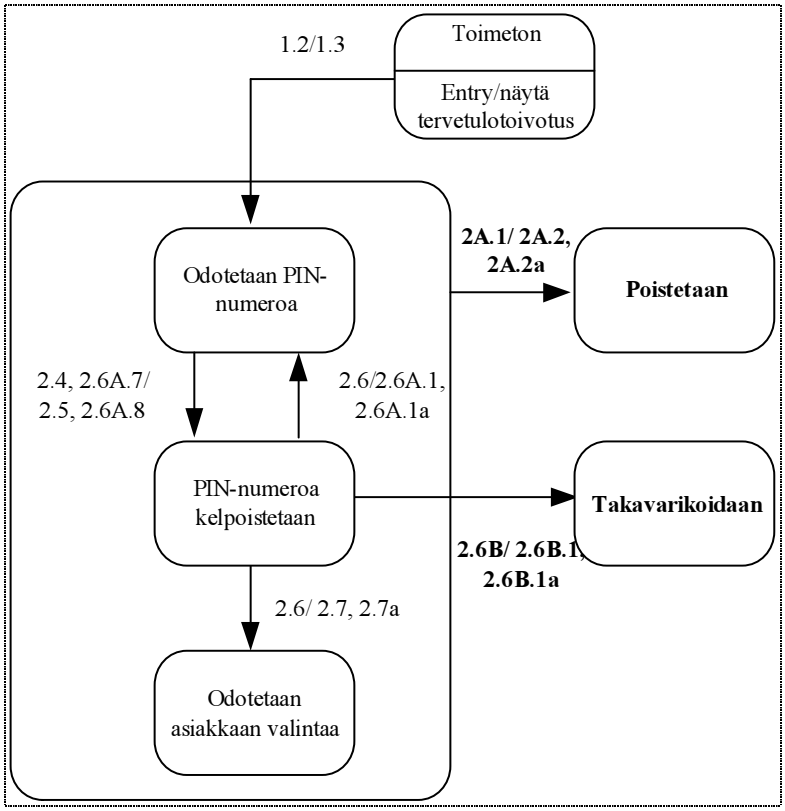
Luokkakaaviot tarjoavat komponentin sisältöä kuvaavan luokkahierarkian ja tietoa eri luokista. Luokkahierarkiasta on helppo selvittää, mihin luokkiin ohjelmaan tehty muutos voi vaikuttaa.

Yhteistyökaaviot havainnollistavat komponentin sisältämien olioiden välistä vuorovaikutusta. Kuva 8 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 18 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. Tällaisia viestejä kutsutaan *vaarallisiksi viesteiksi*.
- *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 8 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 tilakaavioon 9 on lisätty kaksi uutta tilaa *Poistetaan* ja *Takavarikoidaan*.



Kuva 9. Tilakaavio pankkiautomaattiohjelman kontrollikaaviosta (Gao ym., 2003).

Kaavioita voidaan käyttää apuna suunniteltaessa testausstrategioita. Minimointitekniikkaa käyttämällä valittaisiin regressiotestijoukkoon minimimäärä testitapauksia, jotka yhdessä kattaisivat kaikki muutetut osat. Kaavioiden perusteella on myös mahdollista kehittää kattavuuteen perustuva testijoukko, jolloin mukaan otetaan kaikki ne testitapaukset, jotka testaavat jotakin muuttunutta ohjelman osaa. Minimointitekniikka ei ole yhtä perinpohjainen kuin kattavuuteen perustuva tekniikka, mutta toisaalta se on kattavuuteen perustuvaa tekniikkaa halvempi.

Sen lisäksi, että varmistetaan, ettei muutos ole aiheuttanut virheitä muutettuun luokkaan, tulee varmistaa, ettei muutos vaikuttanut virheellisesti muiden komponentin sisältämien luokkien toimintaan. Tämänlaiset virheet voivat olla kahdenlaisia (Gao ym., 2003):

- *Muutosten vaikutukset kontrollivirtaan*. Muutettuja ohjelman osia voidaan kutsua useissa eri tilanteissa. Esimerkiksi kuvaan 8 lisätty toimintojoukko 2A, 2A.1, 2A.2 ja 2A.3 voidaan aloittaa kolmesta eri tilanteesta: (1) odotetaan PIN-numeroa, (2) PIN-numeroa kelpoistetaan ja (3) odotetaan asiakkaan valintaa. Tästä johtuen on tarpeellista testata kaikki tilakaaviossa näkyvät tilanteet, joihin muutos on mahdollisesti vaikuttanut. Pyrittäessä korkeaan laatuun pitää ohjelmasta testata uudelleen myös kaikki polut, jotka liittyvät sellaisiin tiloihin tai siirtymiin, joihin muutos on voinut vaikuttaa.

- *Muutosten vaikutukset tietoriippuvuussuhteisiin* (data dependence). Komponentin rajapinnan kutsu on oikeasti komponentin implementoiman funktion kutsu. Tästä johtuen, kun rajapinnassa v_1 määritellyllä funktiolla on tietoriippuvaisuussuhde toisessa rajapinnassa v_2 määritettyyn funktioon, voi rajapintojen kutsujärjestyksellä olla vaikutusta lopputulokseen. Yhteistyökaaviossa entiteetti-olioon menevä viesti merkitsee yleensä olion sisältämien tietojen päivätystä, 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.

5.2.2 Regressiotestaus täydentävän ja mukautuvan ylläpitoprosessin aikana

Korjaava ja täydentävä ylläpitoprosessi voi muuttaa ohjelmistokomponentin vaatimuksia. Gao keskittyi kirjassaan kuitenkin vain sellaisiin tapauksiin, joissa vaatimukset eivät muutu.

5.2.2.1 Kontrollivirtojen yhtäläisyysarviointi

Yhteistyökaavioissa poluilla esitetään erilaisia kontrollijonoja. Esimerkiksi kuva 10 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 10) tilillä olevat rahat voivat olla joko riittäviä tai riittämättömiä. Kontekstirajoite (context constraint) on yhdistelmä vartijoita, jotka liittyvät komponentin rajapinnan suoritukseen. Esimerkiksi kuvassa 10 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:

- *Komponentin vartijat ja kontekstirajoitukset pysyvät samoina kuin vanhassa komponentissa*. Tällöin regressiotestauksessa on tarpeellista testata jokainen rajapinta uudelleen.
- *Muutettu komponentti sisältää uusia vartijoita, jotka osaltaan ohjaavat suoritusta*. Jos esimerkiksi vanhalla komponentilla on kaksi vartijaa (tili ja rahat) ja kolme erilaista kontekstirajoitetta (kuva 10):
  1. [Kelvollinen tili] ja [Riittävät rahat]

2. [Kelvollinen tili] ja [Riittämättömät rahat]
3. [Kelvoton tili]

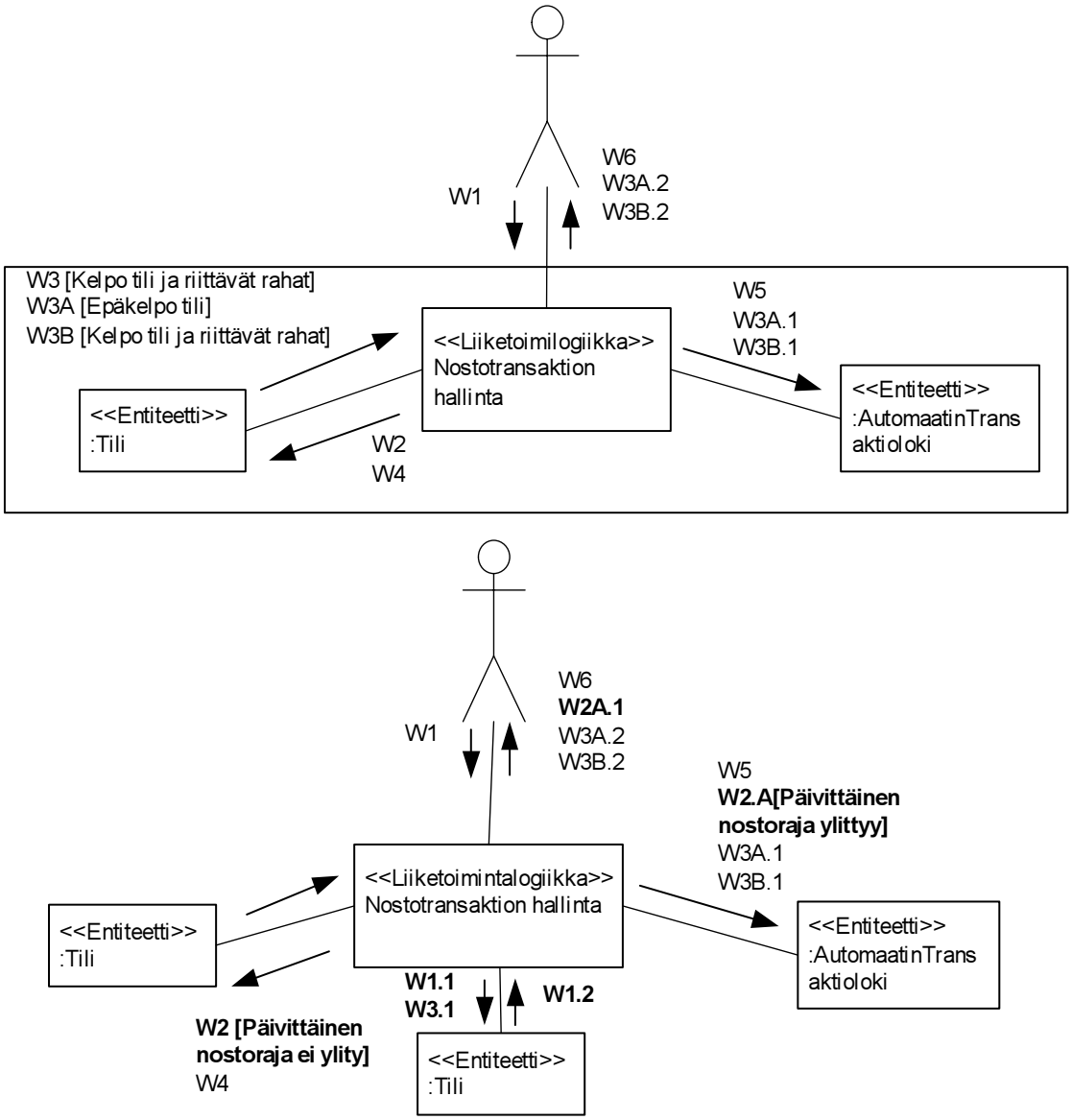
- Muutetulla komponentilla on yksi uusi vartija [Päivittäinen nostoraja]. Tällöin muutetulla komponentilla on neljä kontekstirajoitetta:

1. [Päivittäinen nostoraja ei ylity]
2. [Päivittäinen nostoraja ei ylity], [Kelvollinen tili] ja [Riittävät rahat]
3. [Päivittäinen nostoraja ei ylity], [Kelvollinen tili] ja [Riittämättömät rahat]
4. [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. Tässä tapauksessa kannattaa testata kohta 1 ja yksi kohdista 2, 3 tai 4.

- *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 toimia seuraavalla tavalla. Testaa kaikki uudet vartijat ja testaa kaikki sellaiset kontekstirajoitukset, jotka eroavat alkuperäisen komponentin kontekstirajoituksista, kun niistä on poistettu uudet vartijat.

Joissain tapauksissa vartijoiden järjestyksellä on vaikutusta komponentin toimintaan, joka pitää ottaa huomioon regressiotestausta suunniteltaessa.



W1: Rahojen nostopyyntö (transaktion yksityiskohta)
W2: Tarkista tilin tiedot
W3: Veloita (tili #, määrä)
W3: Tilin tiedot
W3A: Epäkelpo tili
W3A.1: Lokitransaktio
W3A.2: Vastaus nostopyyntöön (epäkelpo tili)
W3B: Riittämättömät rahat
W3B.1: Lokitransaktio
W3B.2: Vastaus nostopyyntöön (riittämättömät rahat)
W4: Nosta rahat tililtä
W5: Lokitransaktio
W6: Vastaus nostopyyntöön

W1.1: Tarkista päivittäinen nostoraja (Kortin tunnus, määrä)
W1.2: Vastaus nostorajakyselyyn
W2A: Lokitransaktio
W2A.1: Vastaus nostopyyntöön (päivittäinen nostoraja ylittyy)
W3.1: Päivitä päivittäinen nostoraja (kortin tiedot, määrä)
(Uudet viestit)

Kuva 10. Yhteistyökaavio liittyen nostotapahtumaan pankkiautomaatista (Gao ym., 2003).

5.2.2.2 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.

6 OLIO-OHJELMIEN REGRESSIOTESTAUS

6.1 Kehitettyjä tekniikoita

Olio-pohjaisten ohjelmien testauksen avuksi on kehitetty useita regressiotestien valitsemistekniikoita (Harrold ym., 2001; Hsia ym., 1997; Kung ym., 1994a; Kung ym., 1994b; Rothermel ym., 2000; White ym., 1997). Näistä tekniikoista kolme on turvallisia (Harrold ym., 2001; Rothermel ym., 2000; White ym., 1997). Kahdessa viimeksi mainitussa tekniikassa on kuitenkin joitakin puutteita ja rajoituksia. Rothermelin ym. kehittämä tekniikka ei osaa käsitellä koodin sisältämiä virheiden käsittely-rakenteita. Algoritmia voidaan käyttää ainoastaan valmiisiin ohjelmiin tai luokkiin, joille on tehty ajurit. Lisäksi algoritmi voi valita testijoukkoon paljon turhia testitapauksia. Whiten ym. kehittämä tekniikka olettaa muutettujen luokkien koodin olevan saatavilla ja päättelee koodin perusteella, mitkä luokista kannattaa uudelleentestata. Koska tekniikka valitsee testitapaukset luokkatasolla, voi regressiotestijoukkoon päätyä turhiakin testejä. Tämäkään tekniikka ei osaa käsitellä koodin virheiden käsittely-rakenteita.

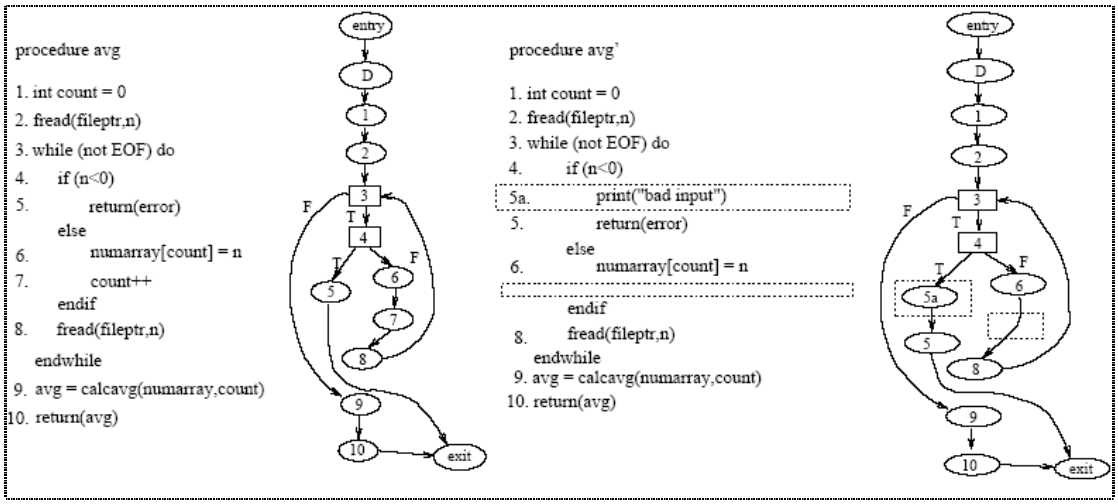
6.1.1 Yleinen regressiotestien valintatekniikoiden käyttämä toimintamalli

Turvalliset regressiotestien valintatekniikat käyttävät yleensä seuraavanlaista toimintamallia (Rothermel ym., 1997a), jonka muunnelmaa myös myöhemmin esitetty Java-kielelle tarkoitettu valintatekniikka käyttää. Esimerkissä käytetään termiä vaarallinen entiteetti kuvaamaan ohjelman osaa (esimerkiksi funktio, luokka tai kaari), johon tehty muutos on voinut vaikuttaa. Tässä esimerkissä entiteeteillä tarkoitetaan kaaria.

Kuvassa 11 vasemmalla on esitetty esimerkkiohjelman **avg** koodi sekä koodin pohjalta tehty kontrollivirtakaavio. Kuvassa oikealla esitettyyn muutettuun versioon **avg'** on lisätty lause 5a ja poistettu lause 7. Algoritmi käy läpi alkuperäistä sekä muunneltua kaaviota samanaikaisesti aloittaen kuvan kaavioissa ylhäällä olevista aloitussolmusta ja jatkaen solmujen läpikäymistä, kunnes poikkeama kaavioiden välillä löytyy. Kun algoritmi on edennyt kuvan 11 kaavioissa neljänteen solmuun asti se huomaa, että kaaren T (tosi) päässä oleva solmu on poikkeava eri kaavioissa. Tällöin algoritmi lisää kaaren (4, 5) vaarallisten entiteettien joukkoon ja lopettaa tämän polun läpikäymisen. Algoritmi palaa solmuun numero 4 ja tarkastelee F (false)-kaaren päässä olevaa solmua. Koska kyseisen kaaren päässä oleva solmu on molemmissa kaavioissa sama (solmu

numero 6), algoritmi siirtyy seuraavaksi sinne. Kuudennesta solmusta lähtevä kaari vie muutetussa versiossa eri solmuun kuin alkuperäisessä, joten kaari (6, 7) lisätään vaarallisten entiteettien joukkoon ja algoritmi lopettaa polun läpikäymisen. Algoritmi palaa vielä solmuun numero 3 ja läpikäy vielä solmut 9 ja 10 ennen loppusolmua. Esimerkissä D-kirjaimella merkityt solmut ovat määrittelysolmuja. Määrittelysolmut sisältävät tiedot kaikista koodissa olevista määrittelyistä. Jos esimerkiksi alun perin int-tyyppinen count-muuttuja määriteltäisiin uudelleen long-tyyppiseksi, etsintäalgoritmi huomaisi eron määrittelysolmussa ja lisäisi kaaren (entry, 1) vaarallisten entiteettien joukkoon.

Kun vaaralliset kaaret on tunnistettu, valinta-algoritmi valitsee regressiotestijoukkoon lisättävät testitapaukset kattavuusmatriisin ja vaarallisten entiteettien joukon perusteella.



Kuva 11. Kuvassa vasemmalla esimerkkiohjelman **avg** alkuperäisen version koodi ja kontrollivirtakaavio. Kuvassa oikealla muutetun version **avg'** koodi ja kontrollivirtakaavio (Harrold ym., 2001).

Testitapaus	Syöte	Odotettu tuloste
1	tyhjä tiedosto	0
2	-1	virhe
3	1 2 3	2

Taulukko 5. Ohjelman **avg** testijoukko (Harrold ym., 2001).

Jokaisen testitapauksen kattamat kaaret kirjataan muistiin ohjelmaa suoritettaessa. Tässä tapauksessa testitapaukset 1, 2 ja 3 kattavat kaaren (entry, 1), testitapaukset 1 ja 3 kattavat kaaren (3, 9) ja testitapaukset 2, 3 kattavat kaaren (3, 4). Taulukossa 6 näytetään tämän testijoukon kattavuus ohjelmassa **avg**.

Kaari	Testitapaus
(entry, 1), (1, 2), (2, 3)	1, 2, 3
(3, 9), (9, 10), (10, exit)	1, 3
(3, 4)	2, 3
(4, 5), (5, exit)	2
(4, 6), (6, 7), (7, 8), (8, 3)	3

Taulukko 6. Taulukossa 5 esitetyn testijoukon kaarikattavuus ohjelmassa **avg** (Harrold ym., 2001).

Taulukon 6 esittämän kaarikattavuusmatriisin ja vaarallisten entiteettien joukon perusteella voidaan päätellä regressiotestijoukkoon valittavat testitapaukset. Esimerkin tapauksessa vaarallisia kaaria olivat (4, 5) ja (6, 7), joten regressiotestijoukkoon valitaan testitapaukset 2 ja 3.

Edellä mainittua tekniikkaa ei voi suoraan käyttää Java-ohjelmille, vaan tekniikkaan pitää tehdä joitakin lisäyksiä ja muutoksia. (Harrold ym., 2001)

6.2 Java-ohjelmointikielelle kehitetty regressiotestien valintatekniikka

Tässä luvussa esitetään Harroldin artikkelissa (Harrold ym., 2001) esitetty Java-ohjelmointikielelle kehitetty regressiotestien valintatekniikka, joka ottaa tehokkaasti huomioon kielen eri piirteet, kuten moniperiytymisen, dynaamisen sitomisen ja virheiden käsittelyn. Kuten luvussa 6.1.1 esitetty yleinen valintatekniikka, myös Javalle tehty regressiotestien valintatekniikka koostuu kolmesta pääaskeleesta. Ensin se rakentaa ohjelmasta kontrollivirtakaavion. Seuraavaksi se käy kaavion läpi ja tunnistaa vaaralliset kaaret. Lopuksi se valitsee kattavuusmatriisin avulla alkuperäisestä testijoukosta ne testitapaukset, jotka testaavat vaarallisia kaaria. Tehokkuuden vuoksi tekniikka ei analysoi ohjelman käyttämiä koodikomponentteja, kuten kirjastoja, joita ei ole muutettu. Tämän vuoksi testattava ohjelma voidaan nähdä jaettuna kahteen osaan: analysoitavaan osaan sekä ulkopuoliseen osaan, jota ei analysoida. Jatkossa käytämme analysoitavista luokista nimitystä *sisäiset luokat* ja muista ohjelmaan kuuluvista luokista nimitystä *ulkoiset luokat*. Edellä mainittujen luokkien sisältämiä metodeja kutsutaan vastaavasti *sisäisiksi metodeiksi* ja *ulkoisiksi metodeiksi*.

Aiemmin esitetty kontrollivirtakaavio on riittämätön ilmentääkseen kaikkia Javan kielirakenteita. Seuraavaksi esitetään *Java Interclass Graph* (JIG)-niminen esitysmuoto, joka on suunniteltu Java-kieltä silmällä pitäen, mutta jota voi käyttää myös muillakin oliokielillä kirjoitettujen ohjelmien mallintamiseen.

6.2.1 Muuttujien ja olioiden tyypit

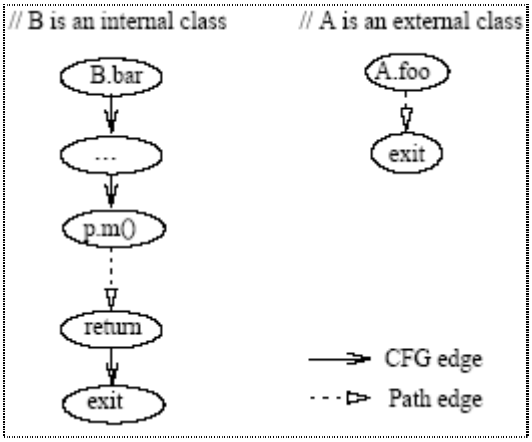
Aiemmin esitetyssä kaaviossa globaaliin määrittelyyn (entry-solmun jälkeinen D-solmu) tehty muutos johtaa kaikkien testitapausten valintaan. Paremman tarkkuuden saavuttamiseksi JIG-mallissa muuttujat kirjoitetaan nimen ja tyypin yhdistelmänä. Esimerkiksi **double**-tyyppinen muuttuja **a** merkitään **a_double**. Näin muutos vaikuttaa vasta siinä kohdassa kaaviota, jossa muuttujaan viitataan.

JIG-mallissa tiedot luokkien hierarkiasta lisätään kaikkiin kohtiin, joissa luokasta luodaan ilmentymä (esimerkiksi kutsumalla **new**-metodia) käyttämällä *yleispätevää luokan nimeä* (globally-qualified class name). Yleispätevä luokan nimi sisältää luokan koko periytymisketjun, alkaen juuresta (java.lang.Object). Yleispätevä luokan nimi sisältää myös luokan implementoimat rajapinnat. Jos luokka implementoi useamman rajapinnan, rajapintojen nimet lisätään yleispätevään luokan nimeen aakkosjärjestyksessä. Jos esimerkiksi paketissa **foo** oleva luokka **B** perii samassa pakkauksessa olevan luokan **A**, joka implementoi paketissa **bar** olevan rajapinnan **I**, luokan **B** yleispätevä luokan nimi on **java.lang.Object:bar.I:foo.A:foo.B**.

Käyttämällä yleispäteviä luokan nimiä on mahdollista tunnistaa luokkahierarkioissa tapahtuvat muutokset. Edellä mainittua mallia käyttämällä luokkahierarkiaan tehdyt muutokset otetaan huomioon vasta silloin, kun luokasta luodaan ilmentymä.

6.2.2 Sisäiset tai ulkoiset metodit

JIG sisältää kontrollivirtakaavion jokaisesta analysoitavan luokan sisäisestä metodista. Kontrollivirtakaavio eroaa aiemmin mainitusta kahdella tavalla. Ensinnäkin jokaista kutsukohtaa kuvataan kutsu- ja paluusolmulla. Toiseksi kutsu- ja paluusolmujen välissä on polku, joka kuvaa polkua kutsutun metodin läpi. Kuvassa 12 vasemmalla olevassa kaaviossa p.m()-solmu on kutsusolmu, ja se on yhdistetty paluusolmuun (return). Kuvassa oikealla on ulkoisen metodin kutsu. Koska oletamme tässä, että ulkoisiin luokkiin ei ole tullut muutoksia, on luokan koodia turha analysoida ja esittää tarkemmin. Siispä tällaista kutsua merkitään kokoon taitetulla kontrollivirtakaaviolla, jossa on vain metodin sisääntulo- ulostulosolmu sekä polku solmujen välillä.



Kuva 12. Kuvassa vasemmalla sisäisen metodin kutsun esitys ja oikealla ulkoisen metodin kutsun esitys (Harrold ym., 2001).

6.2.3 Sisäiset metodikutsut

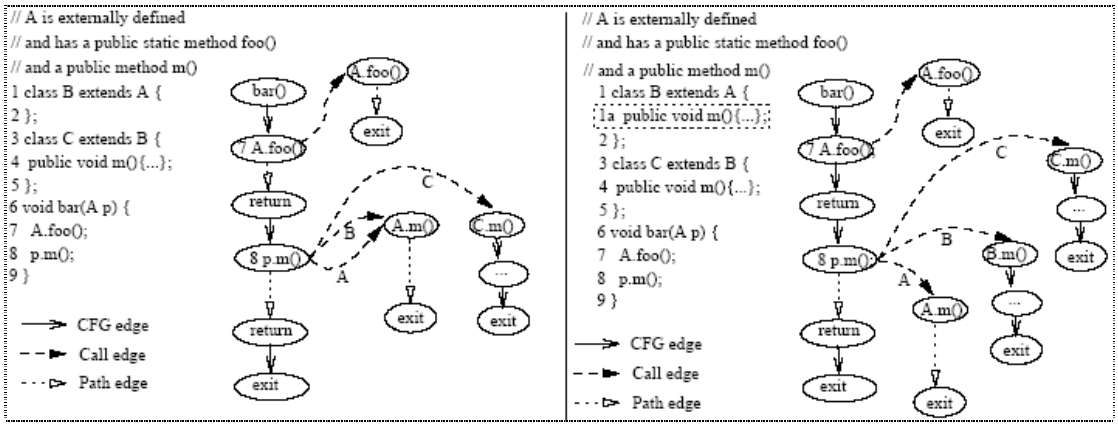
JIG-mallissa jokaista kutsukohtaa ilmennetään polkukaaren yhdistämillä kutsu- ja paluusolmuilla. Kutsusolmu on yhdistetty myös kutsutun metodin sisääntulosolmuun. Ellei kutsu ole virtuaalinen, kutsusolmusta lähtee vain yksi kaari. Kuvassa 13 on kuvattu kolme luokkaa: **A** (ulkoinen), **B** ja **C** sekä metodi **bar**. Luokka **B** perii luokan **A** ja luokka **C** luokan **B** ja määrittelee uudelleen metodin **m**.

Esimerkissä metodi **bar** kutsuu ensin staattista metodia **A.foo.** Koska **A.foo** on staattinen metodi, tapahtumaan ei liity dynaamista sidontaa ja näin ollen solmulla numero 7 on vain yksi lähtevä kutsu **A.foo**-metodin sisääntulosolmuun. Jos kutsu on virtuaalinen, kutsuvasta solmusta lähtee kaari kaikkiin sellaisiin metodeihin, jotka voidaan sitoa kutsuun. Kutsukaaret nimetään sen mukaan, missä metodin toteutus sijaitsee. Esimerkissä kutsu **bar**-metodissa esiintyvä kutsu **p.m** on virtuaalinen kutsu. Riippuen **p**:n (jonka staattinen tyyppi on **A**) dynaamisesta tyypistä, kutsu **m** sidotaan eri metodeihin: jos **p**:n tyyppi on joko **A** tai **B**, kutsu on sidottu **A.m**:ään. Jos **p**:n tyyppi on **C,** kutsu on sidottu **C.m**:ään. Näin ollen kutsusolmusta lähtee kolme kaarta: kaksi niistä (A ja B) on yhdistetty **A.m**:n sisääntulosolmuun ja kolmas kaari (C) **C.m**:n sisääntulosolmuun.

Jotta virtuaaliset metodikutsut voitaisiin esittää oikein, meidän täytyy laskea jokaiselle virtuaaliselle kutsukohdalle metodit, joihin kutsu voidaan sitoa. Tämä voidaan tehdä esimerkiksi käyttämällä erilaisia tyypinpäättely-algoritmeja (type-inferencing algorithms) (esimerkiksi Bacon ym., 1996; Dean ym., 1995; Tip ym., 2000) tai points-to-analysis algoritmeja (esimerkiksi Liang ym., 2001). Esitettävässä tekniikassa käytetään luokkahierarkia-analyysiä (Dean ym., 1995).

Kuvan 13 oikeassa puoliskossa esitettyyn ohjelmaan on tehty muutos lisäämällä uusi metodi **m()** luokkaan **B**. Tämä vaikuttaa lauseessa kahdeksan tehtyyn metodikutsuun silloin kun **p** on **B**. Kun algoritmi vertaa vanhan ja uuden ohjelmaversion kahdeksannesta solmusta lähteviä kaaria se

huomaa, että B-kirjaimella merkityn kaaren kohdesolmu eroaa versioiden kesken. Näin ollen kyseinen kaari merkitään vaaralliseksi.



Kuva 13. Kuvan vasen puolisko esittää foo()-metodin sisäisiä metodikutsuja, jotka käyttävät luokkia A ja C. Kuvan oikea puolisko esittää foo()-metodin sisäisiä metodikutsuja, jotka käyttävät muuttuneita luokkia B ja C (Harrold ym., 2001).

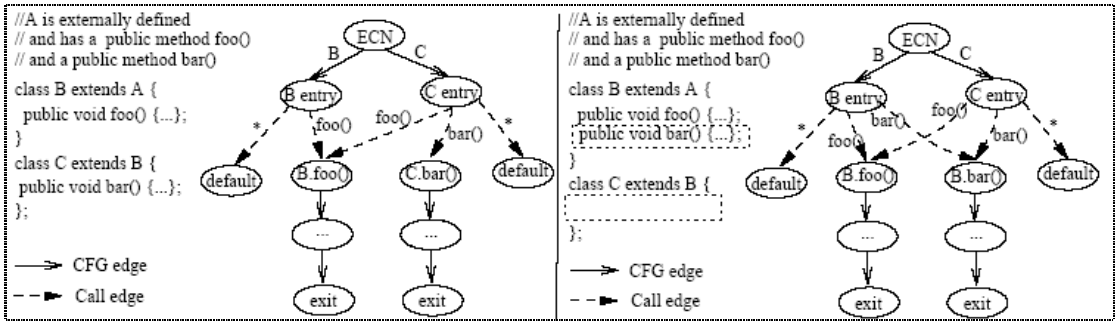
6.2.4 Ulkoiset metodikutsut

Sisäisiin luokkiin tehdyt näennäisen harmittomat muutokset voivat vaikuttaa sisäisten ja ulkoisten luokkien väliseen yhteistoimintaan. Tämän vuoksi keskeneräisten ohjelmien tapauksessa pitää huomioida analysoimattomien osien vaikutus järjestelmän toimintaan.

Kuvassa 14 on esitetty asiaan liittyvä esimerkki. Ulkoista koodia on merkitty solmulla **ECN**. **ECN**-solmusta lähtee yksi kaari jokaista sellaista sisäistä luokkaa kohti, johon voidaan päästä käsiksi ulkoisista luokista. Kaaret päättyvät luokkien sisääntulosolmuihin ja ne merkitään kohdeluokan nimellä.

Ulkoiset metodit voivat kutsua vain sellaisia sisäisiä metodeja, jotka uudelleenmäärittelevät ulkoisen metodin. Tämän vuoksi pitää määritellä luokan sisääntulosolmu (1) jokaiselle luokalle, joka määrittelee uudelleen ainakin yhden ulkoisen metodin, (2) jokaiselle luokalle, joka perii ainakin yhden sellaisen metodin, joka määrittelee uudelleen ulkoisen metodin. Esimerkiksi kuvassa 14 luokka **C** määrittelee uudelleen metodin **A.bar** ja perii metodin **B.foo**, joka puolestaan määrittelee uudelleen metodin **A.foo**. Näin ollen meidän pitää luoda luokan sisääntulosolmu luokalle **C** ja yhdistää se metodeihin **B.foo** ja **C.bar**, joita molempia voidaan kutsua ulkoisesta koodista **C**-tyypin olion kautta. Tämän lisäksi luodaan jokaiselle sisääntulosolmulle oletussolmu (default) ja yhdistetään solmut tähdellä "*" merkityllä kaarella. **A**:n oletussolmu kuvaa kaikkia sellaisia metodeja, joita voidaan kutsua **A**-tyypin olion kautta, mutta jotka on määritelty ulkoisesti. Saadun esitysmuodon avulla voidaan oikeaoppisesti käsitellä sellaisia muutoksia, jotka lisäävät tai poistavat ulkoisia metodeja uudelleen määrittteleviä sisäisiä metodeja.

Kuvassa 14 oikealla esitetystä ohjelmasta on poistettu **bar()**-metodi luokasta **C** ja lisätty uusi **bar()**-metodi luokkaan **B**. Tehty muutos voi vaikuttaa ulkoiseen kutsuun, silloin kun kutsun kohteena on joko **B** tai **C**. Algoritmin verratessa molempien ohjelmaversioiden "C entry"-solmusta lähteviä kaaria huomataan, että "bar()"-nimisen kaaren kohdesolmu on muuttunut. Tämän seurauksena kaari merkitään vaaralliseksi.



Kuva 14. Esimerkki ulkoisen metodikutsun kuvaamisesta (Harrold ym., 2001).

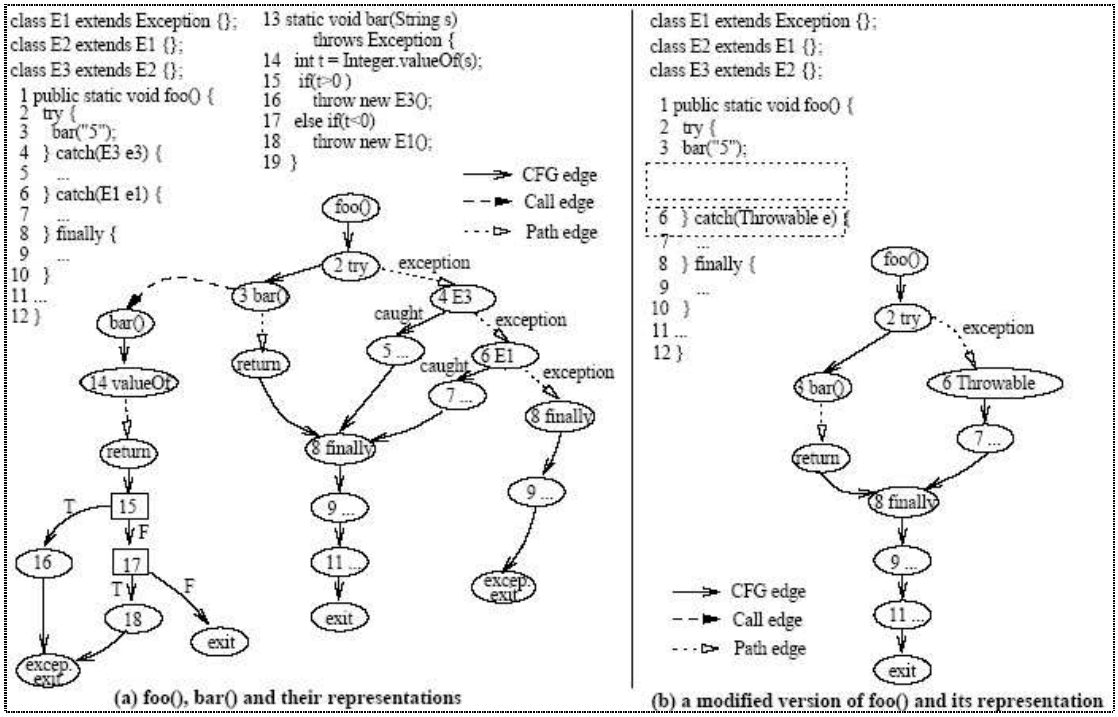
6.2.5 Poikkeusten käsittely

JIG-mallissa kuvataan tarkasti try-lohko sekä jokainen catch- ja finally-lohko. Kuvassa 15 on esitetty esimerkki virheiden käsittelyrakenteita sisältävän ohjelman kuvaamisesta. Jokaista try-lausetta kohti tehdään yksi try-solmu (esimerkissä solmu numero 2). Try-solmusta lähtee kaari ensimmäiseen try-lohkon sisältämään lauseeseen. Jokaista catch-lausetta kohti tehdään yksi catch-solmu, joka nimetään catch-lauseen käsittelemän virheen tyypin mukaan. Catch-solmusta lähtee caught-niminen kaari catch-lohkon ensimmäiseen lauseeseen.

Esimerkiksi kaari (2, 4) edustaa kaikkia niitä polkuja, jotka käyvät lauseessa 2 ja saavuttavat lauseet 16 tai 18, tai lauseen Integer.valueOf() heittämän virhelausekkeen. Eri catch-lohkot yhdistetään toisiinsa "exception"-kaarilla.

Lopuksi tehdään finally-solmu, josta yhdistetään kaari finally-lohkon ensimmäiseen lauseeseen. Jokaisen try- ja catch-lohkon loppulauseesta yhdistetään kaari finally-solmuun. Finally-lohkon loppulauseke yhdistetään try-lausetta seuraavaan lauseeseen. Sellaisten virheiden varalta, joita ei käsitellä missään try- tai catch-lohkossa tehdään finally-lauseesta kopio. Viimeinen catch-solmu yhdistetään tehtyyn finally-solmuun. Jos try-lausetta ei ole ympäröity toisella try-lauseella, kopioidun finally-lauseen loppu yhdistetään solmuun nimeltä *exceptional exit*. Exceptional exit-solmu mallintaa sitä tilannetta jossa käsittelemätön virhe aiheuttaa poistumisen metodista. Jos finally-lohkoa ei ole olemassa, eikä try-lausetta ole ympäröity toisella try-lauseella, try-lohkon viimeinen catch-solmu yhdistetään exceptional exit-solmuun exception-nimisellä kaarella.

Käyttämällä saatua kuvausta, ohjelman poikkeusten käsittelyyn tehdyt muutokset voidaan löytää käymällä samanaikaisesti läpi alkuperäistä ja muutettua kaaviota. Kuvassa 15 oikealla on esitetty foo()-metodin muutettu versio, josta on poistettu lauseet 4 ja 5, ja jonka kuudennen lausekkeen käsittelemän virheen tyyppiä on muutettu. Vertaamalla vanhan ja uuden version try-solmusta lähteviä kaaria, huomataan "exception"-kaareen tulleen muutoksia, joten kyseinen kaari merkitään vaaralliseksi.



Kuva 15. Poikkeusten käsittelyn kuvaaminen (Harrold ym., 2001).

6.2.6 Läpikäyntialgoritmi

Vaaralliset kaaret löytävä algoritmi kutsuu kuvassa 16 esitettyä Compare()-metodia main()-metodille, ECN-solmulle ja kaikille staattisten metodien sisääntulosolmuille. Compare()-metodille annetaan syötteenä sekä alkuperäisen että muutetun version JIG-kuvauksen vastaavat solmut. Suorituksen päätteeksi vaaralliset kaaret on listattu joukossa ε.

```
Procedure        Compare(N,N')
input            N: a node in the JIG for original program P
                 N': a node in the JIG for modified program P'
global output    E: set of dangerous edges for P
 begin Compare
1.   mark N "N'-visited"
2.   foreach edge e' leaving N' do
3.     e = match(N,e')
4.     if e is null then continue
5.     C = e.getTarget()
6.     C' = e'.getTarget()
7.     if ¬NodesEquiv(C,C') then
8.       E = E ∪ e
9.     elseif C is not marked "C'-visited"
10.      Compare(C,C')
11.    endif
12.  endfor
13.  foreach edge e leaving N and
           not matched to any edge leaving N' do
14.    E = E ∪ e
15.  endfor
 end Compare
```

Kuva 16. Compare-aliohjelma (Harrold ym., 2001).

Kun ohjelmasta on tehty JIG-mallin mukainen kuvaus, voidaan ohjelman koodia käsittelemällä tai suoritusympäristöä muuttamalla nauhoittaa kunkin testitapauksen läpikäymät kaaret. Kattavuustiedon avulla regressiotestien valintatekniikka löytää vaaralliset testitapaukset.

6.2.7 Retest-valintajärjestelmä

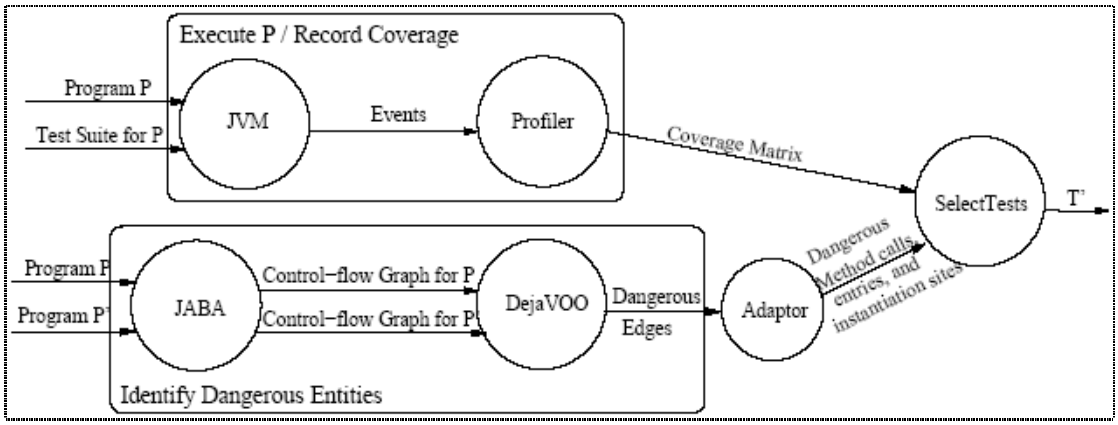
Kuvassa 17 esitetään regressiotestien valintajärjestelmä nimeltä Retest, joka perustuu kuvassa 3 esitettyyn malliin. Retest koostuu kolmesta pääkomponentista: dynaamista tietoa keräävästä Profiler-komponentista, staattisen analyysin DejaVOO-komponentista ja testien valintaan keskittyvästä komponentista nimeltä SelectTests. Retest sisältää myös Adaptor-komponentin, joka muuttaa DejaVOO:n tuottaman tulosteen samantyyppiseksi Profilerin tuottaman kanssa. Kuvassa esitetty JVM-moduuli ilmentää Java-tulkkia (Java Virtual Machine), joka mahdollistaa Java-ohjelman suorittamisen eri ympäristöissä.

Profiler-komponentti käyttää rajapintaa nimeltä Java Virtual Machine Profiler Interface (JVMPI) (Sun Microsystems, 2004) kerätäkseen kattavuustiedot ohjelmasta **P**, kun **P** ajetaan testijoukon **T** sisältämillä testitapauksilla. JVMPI:n rajoituksista johtuen nykyinen Profiler-komponentti ei osaa tallentaa tietoa yksittäisten lausekkeiden suorituksesta, lukuun ottamatta luokan ilmentymän luomista, metodin sisääntulokohtia ja metodikutsuja. Profiler ei myöskään osaa tallentaa tietoja poikkeuksista ja niiden käsittelystä. Virtuaalimetodikutsun kohdalla Profiler tallentaa myös vastaanottajan tyypin niin, että se voi päätellä mitä metodia oikeastaan kutsuttiin.

DejaVOO-moduuli implementoi luvuissa 6.2.1 - 6.2.6 esitetyn analysointialgoritmin, käyttäen apunaan JABA-menetelmää (Java Architecture for Bytecode Analysis) rakentaakseen ohjelmista **P**

ja **P'** JIG-kuvaukset ja kerätäkseen muut tarvittavat tiedot. JABA-menetelmä on Aristotle Research Group-tutkimusryhmän kehittämä Java-ohjelmien bytecode-tason analysointiin tarkoitettu arkkitehtuuri. DejaVOO:n tulosteena on lista vaarallisista kaarista, jotka voivat olla joko kontrollivirtakaavion kaaria (CFG-edge) tai kutsukaaria (call edge). Adaptor-moduuli käsittelee näitä kaaria eri tavalla.

Yllä esitetty Retest-järjestelmä olisi tarkempi, jos se osaisi käyttää aiemmin luvussa esitettyä tekniikkaa. Epätarkkuus johtuu siitä, ettei Profiler ei osaa tallentaa jokaisen yksittäisen lausekkeen tietoja. Näin ollen Retest voi valita tarvittavaa enemmän testitapauksia, jos muutos tapahtuu lausekkeissa, joka on vain muutaman, lausekkeen sisältävää metodia testaavan testitapauksen läpikäymä. Esimerkiksi jos harvoin toistuvan virheen käsittelijää muutetaan, Retest valitsee jokaisen testitapauksen, joka testaa metodia, joka sisältää kyseisen virheiden käsittelijän. Tämä on turhaa, koska virhe ei ilmene suurimmassa osassa näistä testitapauksista.



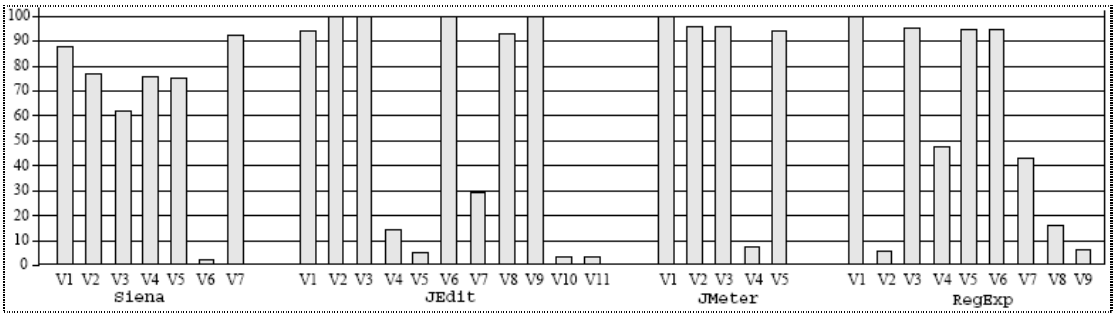
Kuva 17. Retest-regressiotestien valintajärjestelmä (Harrold ym., 2001).

6.2.8 Tekniikasta tehty tutkimus

Luvuissa 6.2.1 - 6.2.6 esitetyn regressiotestien valintatekniikan testaamiseksi suoritettiin tutkimus, jossa tekniikkaa testattiin neljästä eri ohjelmasta (Siena, JEdit, JMeter ja RegExp) tehtyihin versioihin. Tutkimuksessa käytettiin luvussa 6.2.7 esitettyä järjestelmää, joka implementoi aiemmin esitetyn valintatekniikan.

Testijoukon pienentyminen. Retest-ohjelman valitseman testijoukon koko eri ohjelmaversioiden tapauksissa on esitetty kuvassa 18. Tuloksista voidaan nähdä, että valitun testijoukon koko vaihtelee eri ohjelmien ja ohjelmaversioiden välillä. Esimerkiksi Siena-ohjelman yhdestä versiosta tekniikka valitsi vähemmän kuin kaksi prosenttia alkuperäisestä testijoukosta, mutta muiden versioiden kohdalla tekniikka valitun testijoukon koko vaihteli 60–90 prosentin välillä. Saadut tulokset ovat

samantyyppisiä kuin ei-oliopohjaisille ohjelmistoille tehdyissä testeissä (Graves ym., 2001; Rothermel ym., 1998).

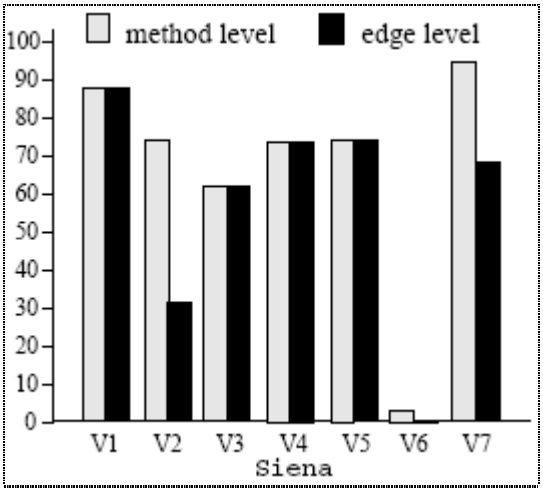


Kuva 18. X-akselilla on lueteltu testattavien ohjelmien eri versiot ja Y-akselilla valintatekniikan valitseman regressiotestijoukon koko (Harrold ym., 2001).

Rakeisuuden vaikutus valitun testijoukon kokoon. Tutkimuksen toisen osan tavoitteena oli selvittää, kuinka paljon valitun testijoukon koko pienenee, jos vaarallisten metodien sijaan valitaan vaaralliset kaaret. Jos tietty vaarallinen kaari saavutetaan kaikissa metodin läpi kulkevissa poluissa, tämä metodi tulee valituksi sekä metodi- että kaarikattavuudessa. Mikäli metodin läpi löytyy jokin polku, joka ei kata vaarallista solmua, on testijoukossakin ainakin yksi sellainen testitapaus, joka voidaan jättää pois.

Neljässä testiohjelmassa on yhteensä 51 vaarallista metodia ja näiden joukosta löydetään 25 sellaista metodia, joista löytyy polkuja, jotka eivät kata vaarallista solmua. Näin ollen jotkin näistä metodeista ovat turhia regressiotestauksen kannalta ja voidaan poistaa testijoukosta.

Tutkimuksessa määritettiin käsityönä Siena-ohjelman testijoukosta kaaritasolla valittavat testitapaukset. Kuvassa 19 on esitetty sekä metodi- että kaaritason valintamenetelmän valitsemien testisarjojen koot. Neljän ohjelmaversion kohdalla molemmat valintatekniikat tuottivat samankokoisen testijoukon, mutta kolmen ohjelmaversion kohdalla kaaritason valintatekniikka valitsi huomattavasti pienemmän testijoukon kuin metoditason valintatekniikka.



Kuva 19. Valittujen testitapausten määrä käyttäen metodi- ja kaaritason testien valintaa (Harrold ym., 2001).

7 REGRESSIOTESTAUKSEN AUTOMATISOINTI

Projektin kuluessa regressiotestauksessa käytettävien testitapausten määrä kasvaa nopeasti, mistä johtuen testien ajaminen manuaalisesti vie kohtuuttomasti aikaa ja resursseja. Lisäksi samojen testien uudelleenajaminen voi tuntua testaajasta tarpeettomalta ja monesti testitapaukset jäävät ajamatta. Regressiotestauksen automatisointi tarjoaa monia etuja:

- Automaattinen testaus mahdollistaa testitapauksen toistamisen tarkalleen samoilla syötteillä kuin edelliselläkin kerralla.
- Virheet löytyvät helpommin ja virheellisten korjausten määrä vähenee, koska tehtyään korjauksen ohjelmoijan on helppo tarkistaa, että korjaus todella toimii.
- Testausraporttien tuottaminen nopeutuu ja raportit yhdenmukaistuvat.
- Ohjelmien tuottamien suurten tulostemäärien tarkistaminen helpottuu. Ihmistestaajan keskittymiskyky ei riitä suurten tulostemäärien tarkastamiseen.

Testitapauksen muuttaminen sellaiseen muotoon, että sen voi ajaa automaattisesti, vie monta kertaa enemmän aikaa 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. Koska regressiotestaus suoritetaan monta kertaa ohjelmistontuotantoprojektin aikana, on usein tarvittavien testitapausten automatisointi kuitenkin välttämätöntä (Binder, 1999, s. 802-803).

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 siihen 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.

8 LÄHTEET

Aristotle Research Group. <URL:http://www.cc.gatech.edu/aristotle>. Viitattu 3.9.2004.

Bacon D., Sweeney P. Fast static analysis of C++ virtual function calls. *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages and, Applications*, s. 324-341, 1996.

Bates S., Horwitz S. Incremental Program Testing Using Program Dependence Graphs. *Proceedings of the 20th ACM 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.

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

Dean J., Grove D., Chambers C. Optimizations of object-oriented programs using static class hierarchy analysis. *European Conference on Object-Oriented Programming*, s. 77-101, 1995.

Elbaum S., Rothermel G., Kanduri S., Malishevsky A. G. Selecting a Cost-Effective Test Case Prioritization Technique. *Software Quality Journal* Vol. 12, No. 3, September (ei vielä ilmestynyt), 2004. Saatavilla pdf-muodossa osoitteesta <URL:http://csce.unl.edu/~grother/papers/sqj04.pdf>. Viitattu 30.8.2004.

Elbaum S., Malishevsky A., Rothermel G. Test Case Prioritization: A Family of Empirical Studies. *IEEE Transactions on software engineering*, Vol. 28, No. 2, 2002.

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., Kallakuri P., Malishvesky 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, 2003.

Fewster M., Graham D. *Software Test Automation*. Addison-Wesley, 1999.

Fischer K., Raji F., Chrusckicki A. A methodology for retesting modified software. *Proceedings of the National Telecommunications Conference*, Vol. 1, 1981.

Gao J.Z., Tsao J., Wu Y. *Testing and Quality Assurance for Component-Based Software*. Artech House, 2003.

Graves T., Harrold M., 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, Jones J, Li T., Liang T., 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.

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

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

Hartmann J., Robson D. Techniques for selective revalidation. *IEEE Software*, Vol. 7, No. 1, 1990.

Horgan J., London S. A data flow coverage testing tool for C. *Proc. of the Symp. on Assessment of Quality Softw. Dev. Tools*, s. 2-10, 1992.

Hsia P., Li X., Kung D., Hsu C-T., Li L., Toyoshima Y., Chen C. A technique for the selective revalidation of OO software. *Software Maintenance: Research and Practice*, 1997.

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

Jones J., Harrold M. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. *IEEE Transactions on Software Engineering*. Vol. 29, No. 3, 2003.

Kaner C., Falk J., Nguyen H. *Testing Computer Software*. Wiley, 1999.

Kung D., Gao J., Hsia P., Toyoshima Y., Chen C. Firewall regression testing and software maintenance of object-oriented systems. *Journal of Object-Oriented Programming*, 1994a.

Kung D., Gao J., Hsia P., Wen Y., Toyoshima Y. Change impact identification in object-oriented software maintenance. *Proceedings of the International Conference on Software Maintenance '94*, s. 202-211, 1994b.

Liang D., Pennings M., Harrold M. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for java. *Proceedings of the ACM Workshop on Program Analyses for Software Tools and Engineering*, 2001.

Myers Glenford J. *The Art of Software Testing*. John Wiley and Sons, 1979.

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

Orso A., Harrold M. 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., Rosenblum D. Component Metadata for Software Engineering Tasks. *EDO '00, Lecture Notes in Computer Science*, Vol. 1999, s. 126-140, Springer-Verlag / ACM Press, November 2000.

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

Rosenblum D ja Weyuker E. J. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*. 23, 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, 1997a.

Rothermel G., Untch R., Chu C., Harrold M. J. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering*, Vol. 27, No. 10, s. 929-948, 2001.

Rothermel G., Elbaum S., Malishevsky A., Kallakuri P. 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., Elbaum S., Malishevsky A., 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., Harrold M., Dedhia J. Regression Test Selection for C++ Programs. *Journal of Software Testing, Verification, and Reliability*, Vol. 10, No. 2, 2000.

Rothermel G., Harrold M. Empirical Studies of a safe regression test selection technique. *Transactions on Software Engineering and Methodology*, Vol.6, No. 2, s. 173-210, 1997b.

Rothermel G., Harrold M., 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*, 1998.

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.

Sun Microsystems. Java Virtual Machine Profiler Interface. <URL:http://java.sun.com/j2se/1.3/docs/guide/jvmpi/jvmpi.html>. Viitattu 3.9.2004.

Tip F. Palsberg J. Scalable propagation-based call graph construction algorithms. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, s. 281-293, 2000.

White L., Abdullah K. A firewall approach for regression testing of object-oriented software. *Proceedings of 10th Annual Software Quality Week*, 1997.

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