OLIO-OHJELMIEN TESTAUS

Anu Partanen
Pro Gradu -tutkielma
Tietojenkäsittelytieteen laitos
Kuopion yliopisto
04.07.2003

Esipuhe

Tämä tutkielma on tehty osana terveydenhuollon PlugIT-tutkimusprojektia. Tutkimusprojektin rahoittajina toimivat Teknologian kehittämiskeskus (Tekes) sekä terveydenhuollon ohjelmistoyritykset ja kuusi sairaanhoitopiiriä.

Haluan kiittää ohjaajaani FT Anne Eerolaa rakentavista neuvoista, kannustuksesta ja ennen kaikkea kärsivällisyydestä. Lisäksi kiitän tarkastajana toiminutta FM Tomi Tikkasta hyvistä kommenteista.

Kiitän lämpimästi myös Nikoa tuesta ja kannustuksesta tutkielman teon jokaisessa vaiheessa.

Tampereella 4.7.2003

_____

Anu Partanen

TIIVISTELMÄ

KUOPION YLIOPISTO, Informaatioteknologian ja kauppatieteiden tiedekunta
Tietojenkäsittelytieteen koulutusohjelma
Tietojenkäsittelytieteen linja

ANU PARTANEN: Olio-ohjelmien testaus
Pro gradu -tutkielma, 70 sivua
Pro gradu -tutkielman ohjaaja:
FT, professori (mvs) Anne Eerola

Heinäkuu 2003

Avainsanat: olio-ohjelmointi, periytyminen, polymorfismi, testaus, metoditestaus, luokkatestaus, integraatiotestaus

Olio-ohjelmoinnin suosion lisääntyessä myös sen testausmenetelmien tutkiminen on alkanut kiinnostaa. Aikaisemmin testausta ei ole pidetty niin tärkeänä, mutta viime vuosina on huomattu testauksen sekä uusien testausmenetelmien kehittämisen tärkeys. Tutkielmassa käsitellään aluksi hieman olio-ohjelmointia ja sen erityispiirteitä. Olio-ohjelmointia verrataan proseduraaliseen ohjelmointiin, jotta huomataan ohjelmointitapojen erot. Näiden erojen sekä olio-ohjelmoinnin erityispiirteiden vuoksi täysin samojen testausmenetelmien käyttö proseduraalisessa ohjelmoinnissa ja olio-ohjelmoinnissa on mahdotonta.

Tutkielmassa tarkastellaan olio-ohjelmien testausta metoditestauksesta aina integrointitestaukseen asti. Metoditestauksesta käsitellään perustestaustapoja. Luokkatestaukseen paneudutaan vähän tarkemmin ja käsitellään eri testaustapoja, jotka riippuvat luokan tyypistä. Periytymishierarkioiden testausta käsitellään myös itsenäisenä kokonaisuutena, koska hierarkioiden ohjelmointi on monesti monimutkaista ja tämän takia virhealtista. Lopuksi tarkastellaan vielä olio-ohjelmoinnin integrointitestausta ja siinä huomioonotettavia asioita kuten tilojen ja polymorfismin testausta.

Tutkielmassa pyritään antamaan selitys, miksi olio-ohjelmien testaukseen eivät käy samat testausmenetelmät kuin proseduraalisten ohjelmien testaukseen. Tutkielmassa tarkastellaan yleisesti olio-ohjelmien testausta ja lisäksi käsitellään olio-ohjelmille soveltuvia testausmenetelmiä testauksen eri vaiheissa. Testausmenetelmiä on pyritty käsittelemään käytännönläheisellä tasolla teoriaa kuitenkaan unohtamatta.

SISÄLLYSLUETTELO

1 Johdanto

Nykypäivän suosituimpia ohjelmointityylejä on olio-ohjelmointi. Se on kasvattanut koko ajan suosiotaan 80-luvulta lähtien, jolloin se alkoi yleistyä. Aikaisemmin paljon käytetty proseduraalinen ohjelmointi on vähentynyt olio-ohjelmoinnin hyväksi havaittujen piirteiden vuoksi.

Olio-ohjelmoinnin yleistyttyä sen eri tutkimusalueet ovat myös kasvattaneet kiinnostusta. On tutkittu erilaisia suunnittelu- ja ohjelmointimenetelmiä. Olio-ohjelmien testauksen tutkiminen on jäänyt vähemmälle. Viime vuosina olio-ohjelmien testauksen tutkiminen on kuitenkin alkanut lisääntyä.

Olio-ohjelmien testauksen tutkiminen on erityisesti tärkeää siksi, että aikaisemmin suositun proseduraalisen ohjelmoinnin testaustavat eivät sellaisenaan sovi olio-ohjelmoinnille. Proseduraalisten ohjelmien ja olio-ohjelmien rakenteissa ja piirteissä on paljon eroja. Proseduraaliset ohjelmat koostuvat pää- ja aliohjelmista, kun taas olio-ohjelmat rakentuvat ylä- ja alaluokista. Proseduraalisissa ohjelmissa on paljon helpommin rajattavissa olevia pieniä yksiköitä. Testaus voidaan aloittaa aliohjelmien yksittäisellä testaamisella. Olio-ohjelmissa testattavan yksikön rajaaminen on hankalampaa, koska olioilla on paljon riippuvuuksia toisiin olioihin [WiH92]. Ohjelmien rakenteiden takia samojen testausmenetelmien käyttö proseduraalisissa ohjelmissa ja olio-ohjelmissa on mahdotonta. Proseduraalisten ohjelmien testauksesta löytyy kuitenkin joitakin tapoja, joita voidaan hyödyntää olio-ohjelmien testauksessa. Ainakin testaustapoja voidaan käyttää joissain tilanteissa pohjana kehitettäessä niitä edelleen.

Olio-ohjelmien testaukseen lisää tutkimista ja kehittämistä tuovat myös olio-ohjelmien erityispiirteet; kapselointi, periytyminen ja polymorfismi. Näitä piirteitä ei proseduraalisista ohjelmista löydy. Kapselointi tarkoittaa tietojen ja toimintojen yhdistämistä. Olio suojaa tietonsa ulkopuolisilta. Tämä aiheuttaa testaukseen havainnoimisongelman. Olion tietoihin pääsee ulkopuolelta käsiksi vain olion omien metodien kautta. Periytyminen on olio-ohjelmoinnin ehkä hyödyllisin piirre. Se edesauttaa ohjelman osien uudelleenkäytettävyyttä, koska luokan toimintoja voidaan periyttää toisille luokille, eikä näin ollen tarvitse ohjelmoida uudestaan samoja toimintoja. Periytymishierarkioiden testauksessa on tärkeää, että perittyjen piirteiden testauksesta ei luovuta heikoin perustein. Helposti voidaan kuvitella, että joltain toiselta luokalta perittyjä piirteitä ei tarvitse enää testata luokassa, joihin piirteet on peritty. Piirteethän on jo testattu niiden alkuperäisessä luokassa. Täytyy kuitenkin ottaa huomioon, että piirteet voivat toimia eri tavalla erilaisessa ympäristössä. Polymorfismi aiheuttaa testaukseen myös lisää tutkimista. Polymorfinen piirre voi edustaa montaa eri luokkaa. Ohjelman käännösaikana ei voida tietää, ollaanko kutsuttu polymorfisen metodin alkuperäistä vai uudelleenmääriteltyä toteutusta.

Olio-ohjelmissa luokka määrittelee olion. Olio on luokan ilmentymä, jolla on aina jokin tila. Olion attribuutit kertovat, missä tilassa olio kulloinkin on. Esimerkiksi taulukko-olio voi olla täysi tai tyhjä tai jotain siltä väliltä. Olion täytyy aina voida säilyttää tämä tila oikeana. Olion attribuuteilla täytyy siis olla kussakin tilassa määritysten mukaiset arvot. Testauksessa tärkeä asia onkin olioiden tilojen tarkkailu. Täytyy testata, että olion tila pysyy oikeana, muuttuu silloin, kun pitääkin ja pysyy muuttumattomana, kun tilan ei ole tarkoitus muuttua. Testauksen tavoitteena on todistaa, että ohjelmalle annetut syötteet saavat aikaan oikean tuloksen vahingoittamatta kuitenkaan järjestelmän tilaa.

Tutkielman toisessa luvussa käydään läpi yleisiä asioita olio-ohjelmoinnista. Selvitetään, miten olio-ohjelma rakentuu ja minkälaisia suhteita olioiden välillä voi olla. Lisäksi käsitellään olio-ohjelmoinnin kolmea erityispiirrettä; kapselointia, periytymistä ja polymorfismia. Kolmannessa luvussa keskitytään olio-ohjelmien ja proseduraalisten ohjelmien välisiin eroihin ja erityisesti niiden testauksen eroihin. Neljännessä luvussa aloitetaan varsinaisesti olio-ohjelmien testauksen käsitteleminen. Luvussa mietitään perusmenetelmiä metodien testaukseen. Lisäksi käsitellään apuohjelmia, joita metoditestauksessa tarvitaan. Luvussa viisi laajennetaan testauksen tutkimista metodeista luokkiin ja mietitään, mitä luokkatestauksessa pitää ottaa huomioon. Esitellään luokkien neljä eri modaalisuutta ja niiden vaikutusta testaukseen. Kuudennessa luvussa käydään läpi periytymishierarkian testausta. Mietitään, millaisia asioita luokkahierarkian testauksessa pitää ottaa huomioon ja esitellään inkrementaalinen testausmenetelmä. Seitsemäs luku keskittyy integrointitestaukseen luokkien välillä. Luvussa käsitellään testausjärjestystä luokkien yhdistämisessä sekä integraatiotestauksen menetelmiä. Tutkitaan muun muassa tilojen testausta ja polymorfismin testausta. Kahdeksannessa luvussa esitetään yhteenveto tutkielman tuloksista.

2 Olio-ohjelmointi

Oliopohjainen ohjelmointi on saavuttanut nykypäivinä johtoaseman käytetyistä ohjelmointiparadigmoista. Olioperustainen ohjelmointi on syntynyt jo 60-luvulla, mutta varsinaisesti sen läpimurto tapahtui 80-luvun puolivälissä. Tällöin alettiin kehittämään enemmän oliopohjaisia kieliä ja menetelmiä. Olio-ohjelmoinnin suosio perustuu sen mahdollisuuksiin uudelleenkäytettävyyden parantamisessa. Nykyajan ohjelmistot ovat yhä monimutkaisempia ja niiltä vaaditaan hyvää ylläpidettävyyttä. Uudelleenkäytettävyys on noussut tärkeäksi ominaisuudeksi. Olio-ohjelmoinnin perusidea on, että ohjelmisto jaetaan luokkiin, jotka edustavat sovellukseen liittyvän maailman tapahtumia eli olioita. Nämä ovat yksiköitä, joilla on tiettyjä ominaisuuksia ja tietty käyttäytyminen [Kos98].

2.1 Olio-ohjelmoinnin peruskäsitteet

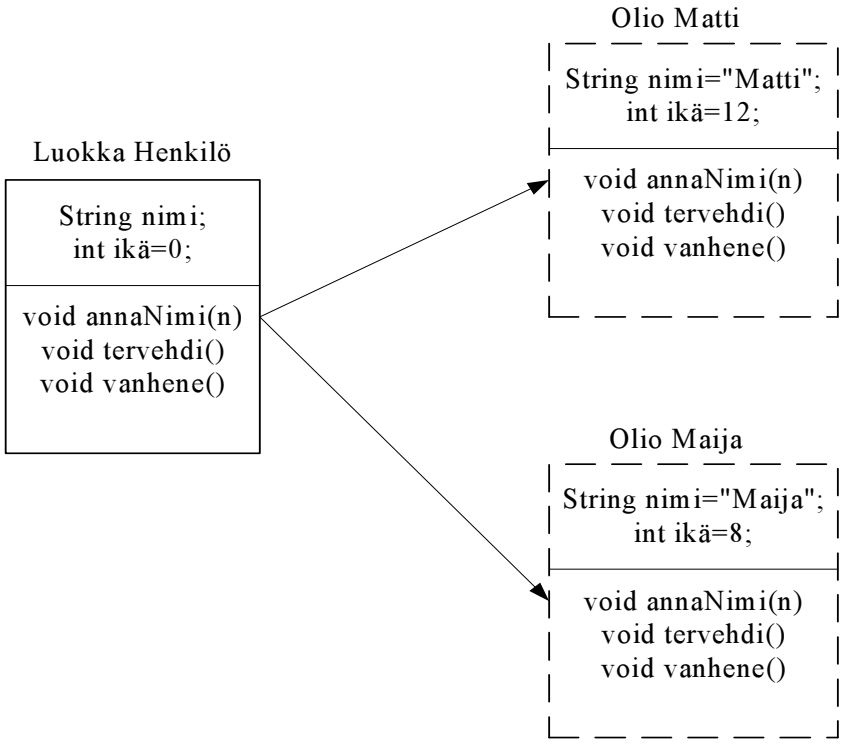
Olio-pohjaisessa ohjelmoinnissa kaikki tiedot ja toiminnot ovat *luokan* (class) sisällä. Luokka on olio-ohjelmoinnin perusyksikkö. Luokka sisältää attribuutteja ja metodeita. *Attribuutit* ovat tietokenttiä. Niissä säilytetään tietoa. *Metodit* puolestaan ovat operaatioita, jotka toteuttavat luokan toiminnot. Seuraavassa esimerkissä on luokka nimeltä *Henkilö*. Tällä luokalla on kaksi attribuuttia, *nimi* ja *ikä* sekä neljä metodia *palautaNimi, palautaIkä*, *tervehdi* ja *vanhene*, joista kaksi ensimmäistä ovat aksessoria. *Aksessori* on metodi, jolla tutkitaan tai muutetaan luokan attribuutteja. Luokalla on myös *kostruktori* nimeltä *Henkilö*. Konstruktorilla luodaan luokasta uusi olio. Tässä tapauksessa konstruktorin avulla uudelle oliolle asetetaan myös nimi. [McS01]

```
  Public class Henkilö{
    String nimi;
    Int ikä = 0;

    Henkilö(String n){
      nimi = new String(n);
    }

    String palautaNimi(){
      return nimi;
    }
    String palautaIkä(){
      return ikä;
    }

    void tervehdi() {
      System.out.println(”Moi, minä olen ”+nimi);
    }
    void vanhene(){
      ikä = ikä + 1;
   }
}
```

Luokan ilmentymää kutsutaan *olioksi*. Olion ominaisuuksiin kuuluu, että se pystyy pyydettäessä suorittamaan sille kuuluvat luokassa määritellyt toiminnot eli metodit. Lisäksi olio pystyy tallentamaan tietoa olion tietokenttiin eli attribuutteihin. Olion attribuutteja ja metodeita kutsutaan yhteises-

ti *olion piirteiksi*. Luokka siis määrittelee itseasiassa olion piirteet. Kuvassa 2.1 on luotu *Henkilö*-luokasta kaksi oliota. Luokka on staattinen, mutta olio on dynaaminen, ajoaikana luotava rakenne.

Jokaisella oliolla on yleensä yksikäsitteinen luokka, jonka mukaan olio on luotu. Luokka on siis eräänlainen muotti, jonka avulla olioita luodaan. Luokka kertoo, mitä attribuutteja ja metodeita sen olioilla on. Piirteet ovat samat jokaisella luokan oliolla, vain attribuuttien arvot voivat vaihdella. Usein puhutaankin luokan attribuuteista ja metodeista, vaikka tarkoitetaan olion attribuutteja ja metodeita. [McS01]



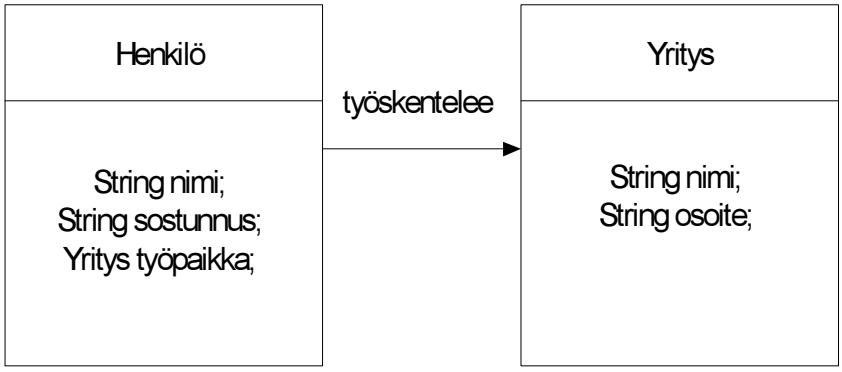
Kuva 2.1: *Henkilö*-luokasta on luotu kaksi eri oliota, *Matti* ja *Maija*.

Olion *tilaksi* kutsutaan attribuuttien arvojen yhdistelmiä. Kutsumalla ja suorittamalla eri metodeja voidaan muuttaa olion tilaa. Olion attribuutteihin päästään käsiksi vain metodien kautta. Ohjelman suoritus koostuu olioiden välisestä vuorovaikutuksesta, kun oliot kutsuvat toistensa metodeita.

Jokaisella oliolla on sen yksikäsitteinen tunniste eli viite. Tämän avulla olio voidaan tunnistaa ja siihen voidaan viitata. Olion attribuutin arvona voi olla viite toiseen olioon. Olio on myös suojattu kokonaisuus. Olio pystyy vapaasti käyttämään omia attribuutteja ja metodeita, mutta toisen olion piirteisiin pääsee käsiksi vain toisen olion suojauksen ehdoilla. Suojausta on käsitelty kappaleessa 2.4.

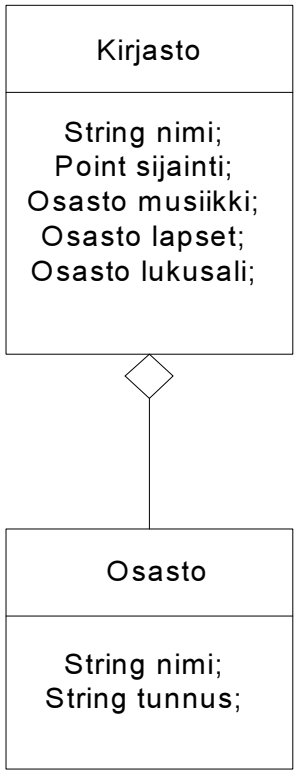
2.2 Olioiden väliset suhteet

Olio-ohjelman toiminta perustuu olioiden keskinäiseen vuorovaikutukseen. On siis luonnollista, että olioiden välillä on erilaisia suhteita. Oliot voivat olla toistensa kanssa *assosiaatio-*, *kooste-* tai *periytyvyyssuhteessa*. Assosiaatio on hyvin yleinen suhde olioiden välillä. Se kuvaa, miten oliot ovat toistensa kanssa tekemisissä. Esimerkki *Henkilö*-olion ja *Yritys*-olion välillä on assosiaatio-suhde, koska henkilö työskentelee yrityksessä. Assosiaatio edustaa jotain sellaista luokkien välistä suhdetta, jolla on tietty pysyvyys. Hetkelliset suhteet, jotka kestävät vain tietyn operaation suoritusajan, eivät yleensä ole edustettuna assosiaatiolla [Kos98]. Yhteyssuhde toteutetaan viittausten avulla tai erillisellä yhteyssuhdeluokalla. Kuvassa 2.2 on esitetty *Henkilö*-olion ja *Yritys*-olion assosiaatiosuhde. *Henkilö*-oliolla on yhtenä attribuuttina viittaus *Yritys*-olioon.



Kuva 2.2: *Henkilö*-olion ja *Yritys*-olion assosiaatiosuhde

Kooste-suhde on oikeastaan eräs assosiaatiolaji. Se on kuitenkin niin merkittävä erityistapaus, että sitä käsitellään omana suhteena. Kooste esittää suhteen ”on-osa” tai ”kuuluu” olioiden välillä [Kos98]. Kooste-suhde tarkoittaa siis, että olio koostuu muista olioista. Tällaisella oliolla on attribuutteinaan oliot, joista se koostuu. Esimerkiksi *Kirjasto*-olio koostuu *Osasto*-olioista. Kooste-suhteen ero assosiaatio-suhteeseen on, että kooste-suhteessa olio tarvitsee tiukasti toista oliota. Assosiaatio-suhde on löysempi. Kooste-suhdetta on olemassa kaksi eri tyyppiä. Löyhemmässä kooste-suhteessa osaoliot voivat olla olemassa myös itsenäisinä. Tiukemmassa kooste-suhteessa osaoliot kuuluvat tiukasti koko elinkaarensa ajan olioon, jonka osia ne ovat. Tällöin osa-oliot luodaan siinä oliossa, joka koostuu näistä osa-olioista. Jos olio tuhotaan, tuhoutuvat myös sen osaoliot [FoS00]. Kuvassa 2.3 on esimerkki löyhemmästä kooste-suhteesta. *Kirjasto*-olio koostuu *Osasto*-olioista. *Kirjasto*-olion attribuuteissa on viittaus eri *Osasto*-olioihin.



Kuva 2.3: *Kirjasto*-olio koostuu *Osasto*-olioista.

Jotkut luokat voivat koostua kokonaan toisten luokkien olioista. Tällaiset luokat ovat *kokoelmaluokkia.* Näiden luokkien ilmentymät ovat oliokokoelmia. Kokoelmaluokat sisältävät viittauksia toisiin olioihin sekä voivat luoda ja tuhota sisältämiään toisia olioita. Tyypillisiä esimerkkejä kokoelmaluokista ovat esimerkiksi listat, pinot, jonot ja taulukot. Kaikki nämä luokat sisältävät toisten luokkien olioita. Tällaiset luokat edesauttavat ohjelmistojen uudelleenkäytettävyyttä ja niitä käytetäänkin luokkakirjastoissa. [MeK98]

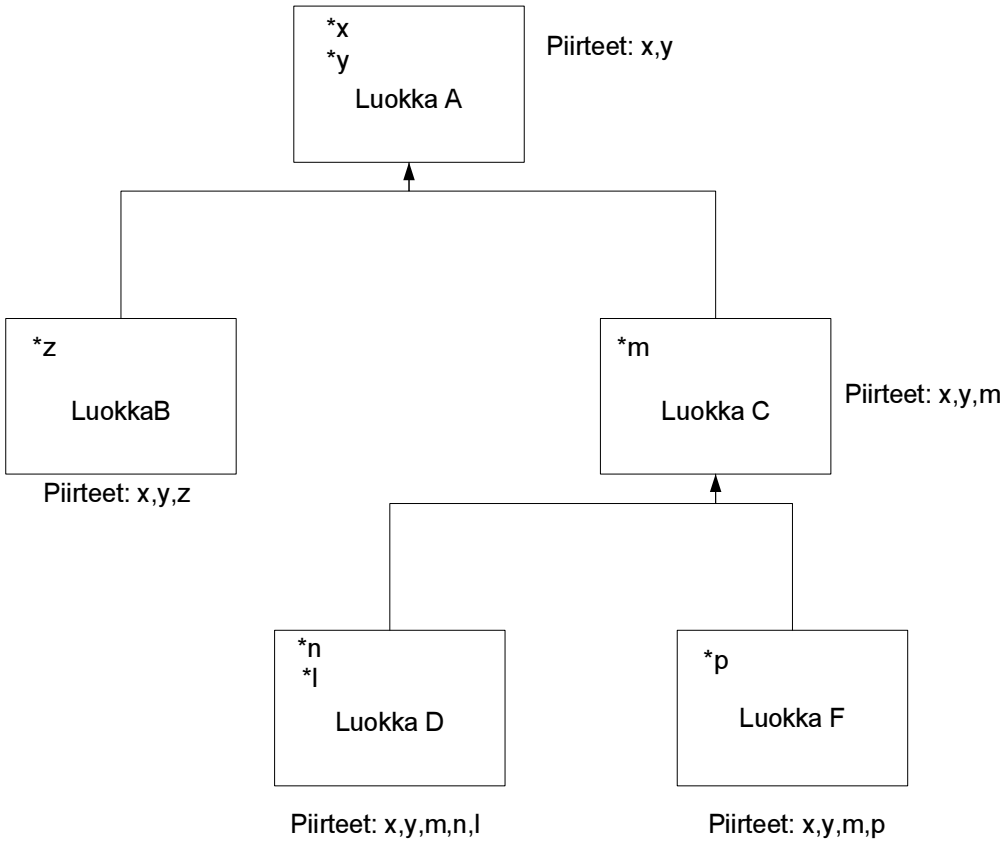
Periytymissuhde tarkoittaa, että luokan piirre siirtyy toisen luokan käyttöön. Periytyminen kuuluu olio-ohjelmien erityispiirteisiin, siksi sitä on selitetty tarkemmin seuraavassa kappaleessa.

2.3 Olio-ohjelmoinnin kolme erityispiirrettä

Olio-ohjelmoinnilla on kolme erityispiirrettä, jotka tekevät siitä olio-ohjelmoinnin. Nämä ovat *kapselointi*, *periytyminen* ja *polyformismi*. Kapselointi tarkoittaa tiedon ja toimintojen yhdistämistä yhdeksi kokonaisuudeksi. Olio-ohjelmoinnissa tämä piirre on oliolla. Olio on ohjelman perusyksikkö, joka ei ole puhtaasti toiminnallinen eikä tietoa säilyttävä, vaan sisältää nämä molemmat ominaisuudet. Perinteisessä proseduraalisessa ohjelmoinnissa on puhtaasti toiminnallinen yksikkö, alioh-

jelma ja tietoa säilyttävät yksiköt, tietue ja tietokanta. Ainut tapa muuttaa olion tilaa eli attribuuttien arvoja on käyttää sen metodeita. [BaS94]

Periytyminen liittyy luokkien välisiin suhteisiin. Se tarkoittaa luokan piirteiden siirtymistä jonkun toisen luokan käyttöön. Luokan *yläluokaksi* kutsutaan luokkaa, jolta luokka perii piirteitä. Vastaavasti luokan *alaluokka* on luokka, joka saa käyttöönsä luokan piirteitä. Näin ohjelmiston luokista muodostuu puumainen hierarkia. Mitä alemmas hierarkiassa mennään, sitä enemmän luokilla on piirteitä, koska luokallahan on aina omat piirteet ja lisäksi se perii yläluokkansa piirteet. Kuvassa 2.4 on havainnollistettu periytymistä. [McS01]



Kuva 2.4: Periytymishierarkia

Jos jokaisella luokalla on korkeintaan yksi yläluokka, on kyseessä *lineaarinenperiytyminen*. *Moniperitytymisessä* luokalla voi olla kaksi tai useampia yläluokkia. Moniperiytymistä ja sen testausta ei tarkastella tässä tutkielmassa, vaan keskitytään lineaariseen periytymiseen, joka on yleisimmin käytössä teollisuudessa tänä päivänä. Luokka on ilman alaluokkiaankin käännettävissä oleva itsenäinen ohjelmanosa, koska luokka tunnistaa vain yläluokkansa, ei alaluokkiaan.

Periytyminen edesauttaa uudelleenkäytettävyyttä. Luokkaan ei tarvitse ohjelmoida piirteitä, joita on sen yläluokalla, koska se voi periä tarvittavat piirteet. Voidaan tehdä yleiskäyttöisiä luokkia,

joissa on monien luokkien tarvitsemia piirteitä. Tällöin muut luokat vain perivät tämän yleiskäyttöisen luokan.

Polymorfismi eli monimuotoisuus on myös ominaisuus, joka on tyypillinen olio-ohjelmoinnille. Yleisesti monimuotoisuudella tarkoitetaan sitä, että ohjelmassa esiintyvä nimetty kohde voi tarkoittaa ajonaikana useita erilaisia asioita. Saman niminen metodi tai attribuutti voi edustaa useiden eri luokkien ilmentymiä. Polymorfismi on yleensä rajoitettu periytymiseen, jolloin näiden eri luokkien täytyy kuulua samaan luokkahierarkiaan [BaS94]. Käytännössä polymorfismi tarkoittaa sitä, että esimerkiksi yläluokan tietty metodi määritellään uudelleen alaluokissa, mutta alaluokkien olioita voidaan asettaa yläluokan olioiden arvoiksi [SoF98]. Esimerkiksi *Asiakas*-olion arvoksi voidaan luoda uusi *Tiliasiakas*-olio. Tällöin polymorfisen metodin kutsu kohdistuu *Tiliasikas*-luokan metodiin. Seuraava esimerkki selkeyttää asiaa:

Määritellään luokka *Asiakas* ja sille alaluokat *Tiliasiakas* ja *Käteisasiakas*. Kaikissa luokissa on *ota_maksu*-metodi, mutta metodin toteutus on erilainen eri luokissa.

```
class Asiakas{
   public void ota_maksu(){    //oletustoimi on laskutus
        pyyda_maksu();
        laskuta_maksu();
   }
   protected void pyyda_maksu()
      {/*toteutus*/}
   private void laskuta_maksu()
      {/*toteutus*/}
}

class Tiliasiakas extends Asiakas{
   public void ota_maksu(){         //erilainen toteutus
        pyyda_maksu();
        kirjaa_tilille();
   }
   private void kirjaa_tilille()
     {/*toteutus*/}
}

class Käteisasiakas extends Asiakas{
   public void ota_maksu(){        //taas omanlaisensa toteutus
       pyyda_maksu();
```

```
        laita_kassaan();
    }
    private void laita_kassaan()
      {/*toteutus*/}
}
```

Nyt voidaan luoda kaikkien luokkien oliot pelkän *Asiakas*-viitteen avulla ja kutsua metodia *ota_maksu*, esimerkiksi:

```
Asiakas asiakas = new TiliAsiakas();
asiakas.ota_maksu();
```

Kaikilta asiakkailta otetaan maksu jollakin tavalla. Voidaan siis tehdä metodin kutsu riippumatta siitä, millainen asiakas on kyseessä. Esimerkin tapauksessa metodin kutsu kutsuisi *Tiliasiakas*-luokan *ota_maksu*-metodia. Sama voitaisiin tehdä yhtä hyvin *Käteisasiakas*-luokalle.

2.4 Näkyvyysmääreet

Olio pystyy suojaamaan piirteensä muilta olioilta. Silloin muut oliot eivät pääse käsiksi olion attribuutteihin tai metodeihin. Tällöin olio antaa ulkopuolisille niin sanotun abstraktin kuvan, joka kertoo, mitkä asiat olio pystyy tekemään, mutta ei sitä, miten nämä asiat tehdään. Suojauksen ansiosta oliot kommunikoivat keskenään käyttäen toistensa ulospäin näkyviä piirteitä. Olion piirteet voidaan suojata monella eri tavalla, mutta usein kaikki attribuutit on suojattu ja julkiset metodit ovat kaikkien saatavilla. [Kos98]

Suojaus tehdään yleensä luokkakohtaisesti. Jokaisen luokan yhteydessä ilmoitetaan, mitkä piirteet ovat käytettävissä luokan ulkopuolelta. Esimerkiksi Java-ohjelmointikielessä näkyvyysmääreitä on oletus-määreen lisäksi kolme: *julkinen (public)*, *suojattu (protected)* ja *yksityinen (private)*. Näillä avainsanoilla luokan piirteiden edessä ilmoitetaan piirteen suojauksesta. Jos määrettä ei ole annettu, suojaus on oletusnäkyvyys, mikä tarkoittaa, että piirteeseen voidaan viitata kyseisen luokan sisältävässä pakkauksessa. Julkiseksi määrättyyn piirteeseen viittaamisella ei ole mitään rajoituksia, vaan kaikki voivat viitata tähän piirteeseen. Vastaavasti määreen ollessa yksityinen piirteeseen voidaan viitata vain kyseisen luokan sisällä. Suojattu-määre on vähän kahden edellisen määreen väliltä. Tällä määreellä piirteet annetaan luokan ja sen jälkeläisluokkien käyttöön, mutta ei muiden luokkien.

Varsinaista testausta ei tarvita näkyvyysmääreiden testaukseen. Kääntäjä ilmoittaa, jos ollaan viittaamassa suojattuun piirteeseen. Koodin tarkastus vaiheessa on kuitenkin hyvä käydä suojaukset läpi, että varmasti oikeat piirteet ovat suojattuja.

3 Erot olio-ohjelmien ja proseduraalisten ohjelmien testauksessa

Testaus on aina jätetty ohjelmistotuotannossa muita vaiheita, analyysi, suunnittelu, ohjelmointi, vähemmälle huomiolle. Vasta viime vuosina testaukseen on alettu kiinnittää huomiota. Erityisesti oliopohjaisten ohjelmien testaus on ollut vähäistä. Monesti kuvitellaan väärin, että olio-ohjelmien testaus on lähes turhaa. Luotetaan vain olio-ohjelmien hyvin suunniteltuun rakenteeseen. Oliokeskeinen suunnittelu saattaa johtaa parempaan arkkitehtuuriin ja kurinalaisempaan ohjelmointiin, mutta se ei itsessään takaa virheetöntä ohjelmaa.

Toisaalta voidaan ajatella, että olio-ohjelmien testaus on hyvin helppoa tai että olio-ohjelmien testaukseen sopivat samat tavat kuin proseduraalisten ohjelmien testaukseen. Myös nämä ovat vääriä kuvitelmia. Olio-ohjelmien rakenteeseen kuuluvat piirteet kapselointi, periytyminen ja polyformismi tuovat testaukseen haastetta ja välttämättömyyden käyttää uusia testausmenetelmiä. Tavallisten, proseduraalisten ohjelmien testausmenetelmät eivät enää sovellu sellaisenaan. Vanhojakin menetelmiä voidaan kuitenkin käyttää hyväksi, ainakin hieman soveltaen [Bin99b].

Oliopohjaisten ohjelmien testaukseen täytyy keskittyä huolella. Uusi ohjelman rakenne tuo monia uusia asioita myös testaukseen. Itseasiassa olio-ohjelmien testaus on jopa tärkeämpää kuin proseduraalisten ohjelmien, koska oliopohjaisilla ohjelmistokomponenteilla on korkea uudelleenkäytettävyys [Bin99b]. On arvioitu, että uudelleenkäytettävä komponentti vaatii kahdesta neljään kertaan enemmän testausta kuin ei-uudeelleenkäytettävä komponentti [Bir92].

3.1 Oliopohjaisten ja proseduraalisten ohjelmien rakenne-erot

Proseduraalisessa ohjelmoinnissa ohjelman rakenne koostuu pää- ja aliohjelmista. Aliohjelma on pienin ohjelman yksikkö. Siitä isompi yksikkö on vain aliohjelmien joukko. Pää- ja aliohjelmat kommunikoivat keskenään välittämällä parametrejä tai käyttämällä globaaleja muuttujia.

Oliopohjainen ohjelma rakentuu olioista ja luokista. Oliolla on attribuutteja tiedon säilytykseen sekä metodeita, joita se voi suorittaa. Attribuutit ja metodit on kuvattu luokassa, jonka ilmentymä olio on. Ohjelman suoritus etenee, kun oliot kommunikoivat keskenään. Monet olion piirteet ovat yleensä suojattuja, joten kommunikointi eri olioiden välillä tapahtuu käyttämällä olioiden julkisia piirteitä.

Olio-ohjelmien rakenne on siis hyvin erilainen verrattuna proseduraalisten ohjelmien rakenteeseen. Proseduraalisissa ohjelmissa pienin itsenäinen testattava yksikkö on aliohjelma. Olio-ohjelmissa pienintä itsenäisesti testattavaa yksikköä on vaikeampi rajata. Luokka itsestään ei ole testattava yksikkö. Se täytyy aina testata ilmentymiensä kautta. Metodeita ei ole järkevää testata

luokasta irrallaan, vaikka joissakin tapauksissa näin voidaan tehdä sopivien apuohjelmien avulla. Jokainen metodin suoritus saattaa kuitenkin muuttaa olion tilaa, mikä pitää ottaa huomioon. Oliokeskeisten ohjelmien testauksessa täytyykin muun muassa keskittyä olioiden tiloihin ja tarkastaa, että tilat muuttuvat oikein metodien suoritusten jälkeen [BaS94].

3.2 Olio-ohjelmien testauksen ongelmat

Olio-ohjelmointi tuo rakenteensa, mutta myös käyttäytymisensä vuoksi ongelmia testaukseen. Kappaleessa 2.3 esitellyt olio-ohjelmoinnin erityispiirteet, kapselointi, periytyminen ja polymorfismi, tuovat testaukseen paljon uutta ratkaistavaa. Nämä asiat huomioon ottaen testaukseen pitää saada uusia menetelmiä.

Kapselointi aiheuttaa testaukseen havainnoimisongelman. Ainoa keino tarkkailla olion tilaa on olion metodien kautta. Olion rakenteeseen ei pääse käsiksi muuten kuin suorittamalla sen metodeja. Lisäksi jotkut olion metodeista ovat suojattuja, joten niitä ei voi suorittaa ulkopuolelta. Ongelmia tulee, jos oliolla on attribuutteja, joita ei voida saavuttaa minkään metodin kautta. Testauksessa joudutaan tällöin lisäämään luokkaan testimetodeita, jotka helpottavat olion tilan muutosten tarkkailua [Lab97]. Näillä metodeilla saadaan esiin olion piilotettuja piirteitä, mutta ne ovat samalla häiritseviä, koska metodit ovat ylimääräisiä osia juuri testattavassa luokassa. Hieman parempi tapa on määritellä testimetodit testattavan luokan alaluokkiin ja sieltä käsin tarkkailla testattavan luokan olion tiloja. Tämä tapa taas ei toimi täydellisesti esimerkiksi Java-kielessä, koska alaluokat eivät näe yläluokan yksityisiä private-määreellä määriteltyjä piirteitä [BaS94].

Periytymishierarkiassa luokka perii yläluokaltaan piirteitä, mutta voi myös muokata yläluokalta perittyjä piirteitä ja lisätä uusia piirteitä. Perittyjen piirteiden testauksessa voidaan ajatella, että piirteet on jo testattu yläluokassa, joten niitä ei tarvitse enää testata. Näin ei kuitenkaan aina voida menetellä. Testauksessa täytyy selvittää, mitkä piirteet voidaan jättää testaamatta yläluokan testaukseen luottaen. Tämä on erityisen tärkeää. Suoraan perityt piirteet, joita ei muuteta alaluokassa mitenkään, ovat niitä joiden testausta voidaan vähentää. Monesti kuitenkin nämäkin täytyy testata uudestaan alaluokassa, koska ympäristö on uusi. Alaluokalla voi olla esimerkiksi eri attribuutteja kuin yläluokalla, mikä vaikuttaa metodien toimintaan. Alaluokka voi myös uudelleenmääritellä perittyjä attribuutteja. Alaluokkaan lisätyt uudet piirteet tuovat oman vaikutuksensa luokkaan. Usein joudutaan testaamaan myös yläluokkaa uudestaan, koska alaluokka rikkoo yläluokkansa kapseloinnin ja pystyy muuttamaan yläluokkansa olioiden tilaa. Perityt piirteet, joita muutetaan alaluokassa, täytyy ilman muuta testata alaluokassa uudelleen Tällöin saatetaan joutua kuitenkin testaamaan myös piir-

teitä, jotka eivät ole muuttuneet, mutta ovat vuorovaikutuksessa muutetun piirteen kanssa [BaS94]. Periytymisen testausta on käsitelty kappaleessa 6.

Polymorfismi tuo päättämättömyys- ja tietämättömyys -ongelman testaukseen. Polymorfinen piirre voi edustaa montaa eri luokkaa. Kun kutsutaan polymorfista metodia, ei voida tietää, ennen kuin ajoaikana, mitä metodia ollaan oikeastaan kutsuttu. Onko kyseessä metodin alkuperäinen toteutus vai uudelleenmääritelty toteutus. Tätä kutsutaan dynaamiseksi sidonnaksi. Testauksessa täytyy tehdä oletuksia metodin toteutuksesta pohjautuen metodin parametreihin. Voidaan miettiä kaikki mahdolliset vaihtoehdot ja valita todennäköisimmät vaihtoehdot [BaS94]. Polymorfisten piirteiden testausta käsitellään kappaleessa 7.6.

Metodien ja attribuuttien liittyminen tiukasti luokkiin, metodien välinen viestinvälitys, olioiden välinen vuorovaikutus ja luokkien periytyminen tekevät olio-ohjelmasta yhden kokonaisuuden, jota on vaikea pilkkoa pienemmiksi testattaviksi osiksi. Tämä aiheuttaa myös ohjelmaan monimutkaisuutta, joka vaikeuttaa entisestään testausta. Ennen varsinaista testausta ohjelmakoodia olisikin hyvä käydä läpi tarkastelumenettelyn tapaisessa tilaisuudessa. Siinä ohjelma tulisi tutuksi testaajille ja näkyvimmät virheet, jotka liittyvät koodin rakenteeseen saataisiin poistettua jo tässä vaiheessa.

4 Metoditestaus

Olio-pohjaisten ohjelmien varsinainen testaus aloitetaan ohjelman pienimmistä osista eli yksittäisistä metodeista. Luonnollisesti metodit kuuluvat aina johonkin luokkaan ja lähettävät viestejä muiden metodien kanssa. Tämän takia yksittäistä metodia on hankala testata ilman osia, joiden kanssa metodi on vuorovaikutuksessa. Yksittäiset metodit pitäisi kuitenkin testata. Tarvittaessa täytyy tehdä pieniä apuohjelmia testattavan metodin kanssa vuorovaikutuksessa olevien metodien ja attribuuttien tilalle. Muutoin metodin testaukseen käyvät samantapaiset testausmenetelmät kuin proseduraalisten ohjelmien testaukseen. Yksittäisessä metodissahan olio-ohjelmien erikoispiirteet eivät tule esille.

4.1 Perusmenetelmät

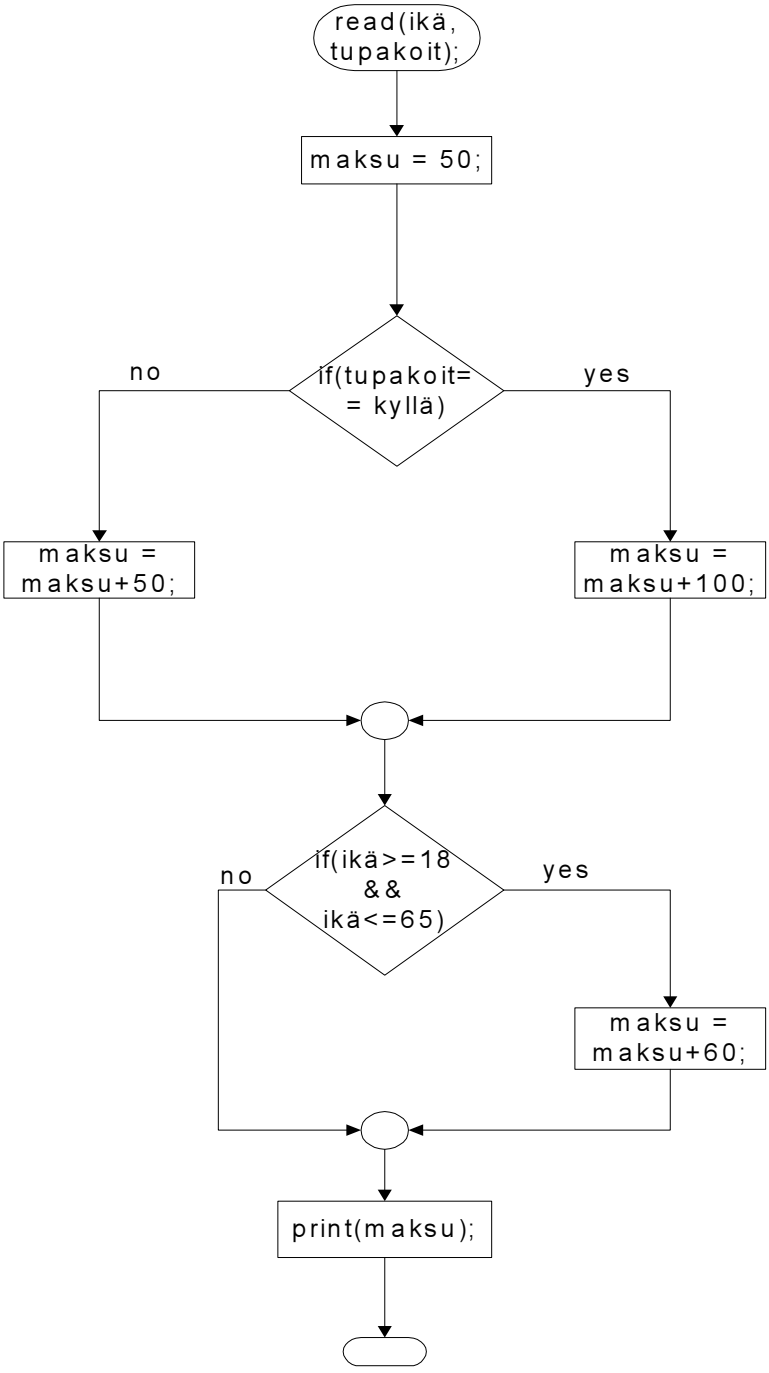
Lasilaatikkotestauksen menetelmät ovat hyviä yksittäisten metodien testaukseen. Lasilaatikkotestauksessa testausmenetelmät pohjautuvat joko kontrollivirtaan tai tietovirtaan. Metodin kontrollivirtaa tutkimalla saadaan selville metodin rakenne ja suorituksen eteneminen metodissa. Tämän perusteella voidaan tehdä testitapauksia pohjautuen eri kattavuuskriteereihin. *Lause- ja päätöskattavuuskriteereiden* mukaan tehdään testitapauksia, jotka testaavat lauseiden suorittamista metodin eri polkuja

pitkin. *Ehto- ja moniehtokattavuuskriteereiden* mukaan tehdyt testitapaukset taas testaavat metodin ehtolauseiden suorittamista ehtojen arvojen eri kombinaatioilla. Testitapaukset, jotka tehdään *polkukattavuuskriteerin* mukaan huolehtivat siitä, että kaikki metodin suorituspolut testataan. Kontrollivirtaan perustavat testitapaukset käyvät huolella läpi metodin suorituksen etenemistä kaikkia mahdollisia metodin suorituspolkuja pitkin [Paa00].

Seuraava metodi laskee vakuutusmaksua. Maksun määrä riippuu vakuutuksen saajan iästä sekä tupakanpoltosta. Metodille on myös piirretty kontrollivuokaavio, joka on esitetty kuvassa 4.1.

```
public int vakuutusmaksu(String tupakoit, int ikä){
  int maksu = 50;

  if(tupakoit == ”kyllä”){
    maksu = maksu + 100;
  }
  else{
    maksu = maksu + 50;
  }
  if(ikä>=18 && ikä<=65){
    maksu = maksu + 60;
  }
  print(maksu);
}
```

Lausekattavuuskriteerin mukaan tehdään metodille sellaiset testitapaukset, että metodin jokainen lause tulee suoritettua. Testitapauksia on helppo valita kontrollivuokaavion avulla. Siitä huomataan, mitkä lauseet tulevat milläkin arvoilla suoritettua. Muuttujan *tupakoit* täytyy lausekattavuuskriteerin mukaan saada kummatkin arvonsa; kyllä ja ei. Näin molemman if-lauseen haarat tulevat suoritettua. Jälkimmäisessä if-lauseen else-osassa ei ole lausetta ollenkaan, joten välttämättä tätä haaraa ei tarvitse lausekattavuuskriteerin mukaan käydä. Muuttujalle *ikä* riittää siis testiarvoiksi jokin väliltä 18-65. Päätöskattavuuskriteerin mukaan tehdyt testitapaukset eroavat lausekattavuuskriteeristä niin, että nyt testitapausten täytyy suorittaa kaikki metodista piirretyt kontrollivuokaavion haarat. Nyt siis *ikä*-muuttujalle pitää antaa myös pienempiä arvoja kuin 18 ja suurempi arvoja kuin 65. Näin metodin jälkimmäisen if-lauseen else-haara suoritetaan myös, vaikka siinä ei varsinaisesti tapahdu mitään toimintoa. Lause- ja päätöskattavuuskriteerien mukaiset testitapaukset ovat esitetty taulukossa 4.1. Kummatkin kriteerit voidaan täyttää kahdella eri testitapauksella.



Kuva 4.1: Kontrollivuokaavio *vakuutusmaksu*-metodille.

	tupakoit	ikä
lausekattavuus	kyllä	36
	ei	52
päätöskattavuus	kyllä	40
	ei	15

Taulukko 4.1: Lause- ja päätöskattavuuskriteerien mukaiset testitapaukset.

Ehtokattavuuskriteerin mukaan tehdään testitapauksia ehtojen perusteella. Jokainen metodin ehto käsitellään itsenäisesti. Metodille tehdään testitapaukset niin, että jokainen ehto saa molemmat arvonsa sekä true että false. Vakuutusmaksua laskevassa metodissa ehtoja on kolme: `tupakoit == "kyllä"`, `ikä >= 18 ja ikä <= 65`. Ehtokattavuuskriteeri täytetään kahdella eri testitapauksella. Esimerkit mahdollisista testitapauksista on esitelty taulukossa 4.2. Moniehtokattavuuskriteerin mukaan tehdyt testitapaukset ovat kaikista kattavimmat verrattuna edellä kuvattuihin. Siinä tehdään testitapauksia myös ehtojen perusteella, mutta nyt yhteen ehtokokonaisuuteen kuuluvat kaikki yhdessä ehtolauseessa määritellyt ehtovaihtoehdot. Vakuutusmaksu-metodissa on kaksi ehtokokonaisuutta, joista jälkimmäinen sisältää kaksi ehtolausetta. Kriteerin mukaan jokaisen ehtokokonaisuuden pitää saada kaikki ehtojen eri arvot. Siis, jos ehtokokonaisuudessa on kaksi ehtoa, testitapauksia tulee neljä. Nämä ovat:

- ehto1 = true ja ehto2 = true
- ehto1 = true ja ehto2 = false
- ehto1 = false ja ehto2 = true
- ehto1 = false ja ehto2 = false.

Välttämättä kaikkia vaihtoehtoja ei voida saavuttaa. Esimerkiksi vakuutusmaku-metodin jälkimmäisessä ehtokokonaisuudessa ei voi tulla tilannetta, että molemmat ehdot saisivat arvon false. Tämä ei ole mahdollista, koska ikä ei voi olla yhtä aikaa pienempi kuin 18 ja suurempi kuin 65. Ehto- ja moniehtokattavuuskriteerin mukaiset testitapaukset on lueteltu taulukossa 4.2. Taulukossa on myös kerrottu, toteutuuko kyseinen ehto (true) vai eikö toteudu (false).

Testitapaukset	tupakoit	ikä		tupakoit == kyllä	ikä>18	ikä<65
ehtokattavuus	kyllä	72		TRUE	TRUE	FALSE
	ei	12		FALSE	FALSE	TRUE
moniehtokattavuus	kyllä	69		TRUE	TRUE	FALSE
	ei	15		FALSE	FALSE	TRUE
	kyllä	29		TRUE	TRUE	TRUE

Taulukko 4.2: Ehto- ja moniehtokattavuuden testitapaukset.

Tietovirtatestaus puolestaan seuraa metodin muuttujia ohjelman ajon aikana. Keskeisiä tapahtumia, joita testataan, ovat muuttujien määrittely ja alustus sekä muuttujien käyttö. Tietovirtatestauksessa muodostetaan kattavuuskriteereiden mukaan testitapauksia, jotka testaavat metodin eri polkuja muuttujien määrittelyn ja käytön välillä. Metodiin ei näin ollen jää virheitä, jotka johtuisivat muuttujien määrittelyn puuttumisesta, muuttujien virheellisestä määrittelystä tai muuttujien määrittelystä väärässä kohtaa [Paa00].

Lasilaatikkotestauksen menetelmiin kuuluu myös *silmukkatestaus*. Silmukat ovat metodien ydinrakenteita ja niiden ohjelmoinnissa tulee helposti virheitä. Tämän takia silmukoiden testaus täytyy tehdä huolella. Lasilaatikkotestauksen kattavuuskriteerit eivät riitä silmukoiden testaukseen, vaan silmukoille on oma testausmenetelmä. Tässä menetelmässä testataan silmukoita eri kierrosten lukumäärällä. Erityisesti paneudutaan silmukoiden minimi ja maksimi kierrosten määriin, koska nämä ovat kaikkein virhealttiimpia. Lisäksi testataan, että silmukan suoritus pysähtyy oikeassa kohtaa [Paa00].

4.2 Metoditestauksen apuohjelmat

Yksittäisen metodin testaukseen tarvitaan usein avuksi pieniä apuohjelmia. Testattava metodi voi olla vuorovaikutuksessa monen sellaisen metodin ja attribuutin kanssa, jotka on rajattu pois testauksesta. Tällöin puuttuvien metodien ja attribuuttien toiminta täytyy keinotekoisesti simuloida. Jos testattava metodi esimerkiksi lähettää viestin jollekin toiselle oliolle ja odottaa tältä vastausta, ei testattavan metodin suoritus voi jatkua ennen kuin se on saanut vastauksen. Toinen tilanne, jossa tarvitaan apuohjelmaa, on silloin kun, halutaan testata, vastaako testattava metodi jonkun toisen olion kutsuun oikein. Tällöin kutsuvan olion toiminta täytyy simuloida.

Apuohjelmia, joita tehdään vastaamaan testattavan metodin kutsuun, sanotaan *tyngiksi*. Metodia varten tehty tynkä simuloi kutsuttavaa olion metodia. Se vastaa testattavan metodin kutsuun ja palauttaa oikean tyyppisen arvon testattavalle metodille. Tärkeintä siis on, että tynkä osaa ottaa vastaan testattavan metodin kutsussa olevat parametrit ja palauttaa takaisin oikean tyyppisen arvon. Jos testattava metodi viittaa johonkin attribuuttiin, joka on testauksen ulkopuolella, tehdään puuttuvalle attribuutille vastaavalla tavalla tynkä. Seuraavassa esimerkissä havainnollistetaan tynkämetodin toimintaa.

Potilas-luokan olio kutsuu metodia *lähiomaistenlkm()*, joka palauttaa potilaan lähiomaisten lukumäärän. Parametrina viestillä potilaan id-numero, joka yksilöi potilaan. Metodi *lähiomaistenlkm()* on kuitenkin rajattu testauksesta pois ja sille täytyy tehdä tynkämetodi. Tynkämetodi ottaa vastaan viestin parametrin ja palauttaa vaaditun tyyppisen arvon. Palautusarvon ei tarvitse olla oikea.

```
lähiomaistenlkmTynkä(int id){
  return 1;
}
```

Apuohjelmat, jotka kutsuvat testattavaa metodia ovat puolestaan *ajureita*. Ajuri kutsuu testattavaa metodia ja ottaa vastaan tämän palauttaman arvon. Ajurin täytyy kutsussaan myös välittää testattavalle metodille oikean tyyppiset parametrit ja ottaa vastaan metodin palauttama arvo. Tästä arvosta

voidaan tarkistaa, että testattava metodi palauttaa oikean arvon. Yleisesti ajurit ovat helpompia ohjelmoida kuin tyngät. Seuraavassa esimerkissä on tehty ajuri:

Potilasrekisteri-luokka sisältää tietoja potilaista. Luokalla on metodi nimeltä *tulosta()*, joka tulostaa listaan kaikki potilaat rekisteristä. Halutaan testata, miten *tulosta*-metodi vastaa kutsuun, mutta olio, joka kutsuisi tätä metodia on rajattu testauksesta pois. Täytyy siis tehdä ajuri simuloimaan tätä kutsua.

```
Lista potilaslista = rekisteri.tulosta();
```

Ajuri kutsuu *tulosta()*-metodia. Metodilla ei ole parametreja, joten parametreja ei tarvitse välittää. Ajuri ottaa vastaan metodin palauttaman arvon. [Paa00]

5 Luokkatestaus

Metodien testaaminen yksitellen erikseen luokasta ei ole välttämättä käytännöllistä eikä järkevää, koska luokan kokonaisvaikutusta ei tällöin testata. Metodit kuuluvat aina johonkin luokkaan, joten on luonnollista testata niitä luokkien vaikutuksen alaisina. Seuraava vaihe metoditestauksen jälkeen onkin testata luokat. Luokat ovat toisten luokkien kanssa yhteydessä, joten jälleen yksittäisten luokkien testauksessa saattaa tulla ongelmia testauksen ulkopuolelle jäävien luokkien takia. Aluksi olisi kuitenkin hyvä testata kukin luokka irrallisena. Vaikka tässä puhutaankin luokan testauksesta, pitää kuitenkin muistaa, että luokkaa ei voida testata sellaisenaan, vaan se testataan aina luokan olion kautta.

Yksittäisten luokkien testauksessa testataan luokassa määriteltyjen metodien ja attribuuttien toimintaa yhdessä. Metodien yhteisvaikutus on suuri virheiden aiheuttaja. Virheellinen metodi voi toimia oikein esimerkiksi toisen metodin syöttämällä virheellisellä arvolla tai jonkin virheellisen sivuvaikutuksen alaisena. Väärässä järjestyksessä kutsutut metodit voivat myös aiheuttaa virheen. Tämän takia metodin sallittujen järjestyksien testaaminen onkin erityisen tärkeää [Bin99].

5.1 Attribuuttitestaus

Luokan sisälle kuuluvat metodien lisäksi attribuutit. Ne ovat luokan tietokenttiä, joissa säilytetään luokalle ominaista tietoa. Attribuuttien testaus saattaa helposti unohtua metodien testauksen rinnalla, mutta luonnollisesti attribuutitkin tulee testata.

Pelkkien attribuuttien testaus ei ole kuitenkaan niin vaativaa, kuin metodien testaus. Ohjelman ajonaikana kääntäjä tarkistaa esimerkiksi, että attribuuttien tyypit ovat oikeita ja että attribuuteilla

on alustusarvot. Monesti attribuuttien testaukseen saattaakin riittää tarkastusmenettelykin. Attribuuttien tarkastuksessa pitää kiinnittää ainakin seuraaviin asioihin huomiota:

- attribuutin tyyppi on oikea
- attribuutti on alustettu
- attribuutin alustusarvo on oikea
- ylimääräisiä attribuutteja ei ole
- kaikki tarvittavat attribuutit on olemassa.

Lisäksi täytyy tarkistaa olioiden assosiaatio- ja koostesuhteissa, että olion viittaus toiseen olioon on toimiva. Tämä viittaushan on toteutettu attribuuttien välityksellä. Attribuutin tyyppinä on olion tyyppi, johon attribuutti viittaa. Tarkistetaan, että tällainen olio on todellakin olemassa, johon attribuutti viittaa.

Attribuuttien testauksessa ei riitä, että tarkastetaan attribuutin alustusarvo. Tietenkin täytyy myös testata, muuttuuko attribuutin arvo oikeissa tilanteissa ja onko muutos oikea. Attribuutit ovat kuitenkin suojattuja ulkopuolisilta. Olion kapselointi huolehtii siitä, että attribuutteihin pääsee käsiksi vain olion metodien kautta. Ainoastaan olion omat metodit voivat muuttaa attribuuttien arvoja. Tämän vuoksi attribuuttien muutokset tulevat testattua samalla kuin testataan olion metodeja. Metodien toimivuus todetaan attribuuttien arvojen muutoksista. Tällöin siis testataan myös sitä, onko attribuuttien arvot muuttuneet oikein ja oikeissa tilanteissa. Myös olion tilan testauksessa testataan samalla attribuutteja. Olion tilojen testausta on käsitelty kappaleessa 7.4.

5.2 Viestinvälitys

Luokassa määriteltyjen metodien pitää toimia oikein yhdessä. Metodit lähettävät viestejä toisilleen ja ottavat vastaan toistensa palauttamia tuloksia. Yleensä luokan metodit välittävät viestejä myös toisen luokan olioille. Mutta testattaessa vain yhden luokan toimintaa, luokan ulkopuoliset viestinvälitykset täytyy korvata kappaleessa 4.2 esiteltyjen tapaisilla apuohjelmilla, tyngillä ja ajureilla.

Kun kyseessä on vain yksi luokka, piirteiden suojaus ja kapselointi ei ole haitaksi. Oliohan näkee kaikki omat piirteensä ja pystyy suorittamaan kaikkia metodeitaan. Metodien viestinvälityksen testauksessa tarkoituksena on tarkkailla, välittävätkö metodit parametrejä oikein ja osaavatko ne ottaa parametrit oikein vastaan. Tietenkin testataan myös metodien palauttama tulos sekä attribuuttien arvojen muutokset. Testauksen avuksi tehdään ylimääräisiä metodeita ja tulostuslauseita, joiden avulla tarkkaillaan attribuuttien ja parametrien arvoja kussakin tilanteessa. Tässä tarkastellaan siis jo hieman olioiden tiloja, joiden testaaminen onkin tärkeintä olio-ohjelmien testauksessa. Tilatestausta on käsitelty kappaleessa 7.4. Pelkkä metodien viestinvälityksen testaaminen ei riitä, koska viestin

lähetysjärjestykselläkin on merkitystä. Viestin lähetysjärjestyksen testausta on käsitelty seuraavassa kappaleessa.

5.3 Modaalisuus

Luokan modaalisuuden määritteleminen helpottaa testitapausten luomista ja estää monilta virheiltä. Modaalisuus nimittäin määrittelee luokan käyttäytymisen olioiden kulloisellekin tilalle ja viestien järjestykselle. Se kertoo olioiden tilojen ja viestijärjestyksen rajoitukset. Luokan käyttäytyminen määritellään tietylle olion tilalle tai viestin järjestykselle tai molemmille. Tilakaavion avulla voidaan määrittää olioiden tilojen järjestys, joka auttaa luomaan testausjärjestyksen. [Bin99]

Testauksen kannalta kiinnostavia ovat neljä seuraavaa modaalisuutta: *Ei-modaalinen* luokka ei aseta minkäänlaisia rajoituksia olioiden tiloille tai viestien järjestykselle. Luokat, jotka toteuttavat perustietotyyppejä ovat usein ei-modaalisia. Esimerkiksi päivämäärän määrittelevä luokka voisi sallia viestejä, missä järjestyksessä tahansa. On sama, asetetaanko ensin vuosi vai päivämäärä.

Yksimodaalinen luokka asettaa viestien järjestykselle rajoituksen välittämättä olion tilasta. Tällaisia luokkia ovat usein sellaiset, jotka tuottavat sovellukseen kontrollirakenteita. Esimerkiksi liikennevalot määrittelevä luokka voisi rajoittaa viestien järjestystä. Punaisen valon asettamiselle tuleva viesti ei voi tulla ennen keltaisen valon asetusviestiä.

Quasimodaalinen luokka asettaa rajoitukset puolestaan olioiden tiloille. Viestien, jotka muuttavat olioiden tilaa, järjestys on rajoitettu. Monet kokoelmaluokat (collection classes) ovat quasimodaalisia. Esimerkiksi pinon määrittelevässä luokassa alkion lisäysviestiä ei hyväksytä pinon ollessa täysi. Luonnollisesti muulloin lisäysviesti on hyväksyttävä.

Modaalisessa luokassa on rajoitettu sekä viestien järjestys että olioiden tilojen muutos. Esimerkiksi pankkitilin toteuttava luokka ei hyväksy nostoviestiä, jos pankkitilin saldo on nolla, ja tilin jäädytysviesti ei mene läpi, jos tilin sulkemisesta on jo lähetetty viesti.

Luokan modaalisuuden määritteleminen auttaa siis keskittymään testauksessa kaikista todennäköisimpiin virheisiin. Esimerkiksi ei-modaalisessa luokassa ei voi tulla virheitä viestien järjestyksessä, koska kaikki järjestykset ovat sallittuja. Tilakaavion avulla voi määrittää viestien eri järjestykset testausta varten. Ei-modaalisen luokan kohdalla siitä ei ole apua, koska tilakaaviosta tulee helposti vain määritettyä kaikki mahdolliset viestien järjestykset. Tämä taas ei ole useinkaan kovin tehokasta testaamista [Bin99]. Seuraavaksi esitellään invarianttitestaus sekä ei-modaalisen, quasimodaalisen ja modaalisen luokan testausta. Yksimodaalisen luokan testaus seurailee soveltuvin osin modaalisen luokan testausta. Invarianttitestaus on apuna muissa testauksissa.

5.3.1 Invarianttitestaus

Invariantti on Boolean-tyyppinen looginen ilmaisu, joka määrittää vaaditut rajat tai tilat muuttujille. Invarianttien testausta tarvitaan useassa testauksessa apuna, erityisesti aluetestauksessa, jossa testataan muuttujien tai olioiden rajoja tai tiloja. Invarianttien testaus on tarkoitettu luokille ja olioille, jotka koostuvat perustietotyypeistä, kuten kokonaisluvuista ja merkkijonoista, tai vähän monimutkaisemmista tietotyypeistä. Tämä testaus on hyvin hyödyllinen muiden testausten yhteydessä. [Bin99]

Tarkoituksena on aikaansaada luokan muuttujien invarianteille mahdollisimman tehokkaat ja kattavat kelvot ja epäkelvot arvot, joilla testata. Epätavalliset, mutta mahdolliset muuttujien arvoyhdistelmät ovat yleinen ongelmien lähtökohta. Nämä muuttujien invariantti yhdistelmät voidaan esittää *luokkainvariantteina*. Luokkainvarianttiin on siis vain koottu luokan muuttujien invariantit. Perus aluetestausmalli on hyödyllinen numeeristen tyyppien luokitteluun, mutta sen laajennuksella voidaan käsitellä myös olio-pohjaisia ohjelmia. Invarianttien testausta voidaan käyttää metodeille tai luokille missä tahansa testauksessa, missä vain invariantteja voidaan määritellä. [Bin99]

Testauksessa testataan kaikkien muuttujien invariantit. Jos virhe löytyy joltain rajalta, täytyy tehdä testitapaus, jolla paljastetaan virheellinen raja ohjelmasta. Esimerkiksi yleinen virhe ohjelmoitaessa rajoja on laittaa rajaksi $x > 1$, vaikka pitäisi olla $x > 0$. Aluetestauksella kyllä löydetään helposti tällaiset virheet, kunhan vain oikeat rajat ovat tiedossa testaajalla.

Testistrategia koostuu kolmesta askeleesta. Ensimmäiseksi tunnistetaan luokkainvariantit. Kehitetään muuttujien arvoyhdistelmistä sellaisia ehtoja, joita halutaan testata. Tämän jälkeen luokkainvariantin jokaiselle ehdolle määritetään on- ja off-pisteet. On-piste tarkoittaa arvoa, joka on juuri ja juuri määritetyllä rajalla. Off-piste on puolestaan arvo, joka on juuri ylittänyt määritetyn rajan. Lopuksi voidaan määrittää vielä in-pisteitä, jotka ovat arvoja rajojen sisältä. Nämä pisteet ovat testitapauksia invarianttien testaukseen. [Bin99]

Seuraavaksi on esitelty *Kello*-luokan metodit ja muodostettu luokasta luokkainvariantti. Lopuksi on vielä kehitetty on-, off- ja in-pisteet testausta varten.

```
class Kello{
  void Kello(int sek, int min, int tunti) {/*toteutus*/}
  void asetaSek(int sek)  {/*toteutus*/}
  void asetaMin(int min) {/*toteutus*/}
  void asetaTunti(int tunti) {/*toteutus*/}
  int palautaSek(int sek)  {/*toteutus*/}
  int palautaMin(int min) {/*toteutus*/}
  int palautaTunti(int tunti) {/*toteutus*/}
```

```
  void kelloOn() {/*toteutus*/}
  void kelloOff() {/*toteutus*/}
  boolean käyköKello(Kello x) {/*toteutus*/}
}
```

Kello-luokan luokkainvariantti on esimerkiksi:

```
((sek>=0 && sek <=59) &&
(min>=0 && min <=59) &&
(tunti>=0 && tunti <=23) &&
(käyköKello(kello1)==true || käyköKello(kello1)==false)
```

Luokkainvariantti määrää, että kellonajan sekuntit ovat välillä 0-59, samoin minuutit. Tunnit puolestaan ovat väliltä 0-23. Lisäksi kellon täytyy olla joko päällä tai pois päältä. Luokkainvariantin perusteella määritetään on-, off- ja in-pisteet. Ne on määritetty taulukossa 5.1.

Ehto		on-piste	off-piste	in-piste
sek >= 0		0	-1	32
sek <= 59		59	60	27
min >= 0		0	-1	41
min <= 59		59	60	15
tunti >= 0		0	-1	8
tunti <= 23		23	24	19
käyköKello(kello1)		TRUE	FALSE	TRUE
käyköKello(kello1)		FALSE	TRUE	FALSE

Taulukko 5.1: On-, off- ja in-pisteet *Kello*-luokan luokkainvariantille

Tätä testaustapaa käytetään siis yhdessä jonkin muun mallin kanssa, esimerkiksi ei-modaalisen ja quasi-modaalisen luokan testausmallien kanssa. Mallin edellytyksenä on, että invariantit ovat saatavilla tai kehitettävissä. Monesti voikin olla, että muuttujien rajoja tai tiloja ei edes tarkkaan tiedetä. Tällöin invarianttien kehittäminen on aikaa tuhlaavaa. Testaus on tehty kunnolla loppuun saakka, kun kaikkien invarianttien on- ja off-pisteet on kehitetty ja kattava aluetestaus on suoritettu näillä arvoilla. Oikein tehtynä testaus paljastaa hyvin tehokkaasti virheet raja-alueilta. Lisäksi testitapaukset pystytään kehittämään hyvin nopeasti, kun invariantit ovat tiedossa. [Bin99]

5.3.2 Ei-modaalisen luokan testaus

Ei-modaaliset luokat ovat luokkia, jotka eivät aseta rajoituksia viestien järjestykseen. Konstruktoria, joka luo uuden olion, lukuunottamatta, mikä tahansa metodi voi seurata jotain toista metodia.

Kun viestijärjestystä ei rajoiteta ja harvat olioiden tilatkin ovat kiellettyjä, ongelmana on löytää testitapauksia, jotka paljastaisivat virheitä. Kuitenkin jotkut viestijärjestykseen liittyvät virheet voivat olla mahdollisia. Esimerkiksi ohjelma ei hyväksy laillista viestijärjestystä tai se tuottaa tuloksena virheellisen arvon. Metodi saattaa myös aiheuttaa jonkin virheellisen sivuvaikutuksen, joka muuttaa olion tilaa tai virheellinen laskutoimitus saattaa vahingoittaa metodien tulosta. [Bin99]

Testaus aloitetaan kehittämällä luokkainvariantit. Seuraavaksi luokkaa testataan suorittamalla metodeja jossain järjestyksessä. Ei-modaalisten luokkien kohdalla viestien lähetysjärjestyksellä ei ole väliä, mutta muutama hyväksi havaittu viestien testausjärjestys on olemassa: *Määrittely-käyttö –järjestys* (Define-use sequence) tarkoittaa, että metodi, jossa määritellään muuttuja edeltää kaikkia metodeja, joissa tätä samaa muuttujaa käytetään. *Satunnaisessa järjestyksessä* (random sequence) lähetetään metodien kutsuja aivan satunnaisesti. Jos jokin viestien järjestys on jostain syystä epäilyksenalainen, sitä kannattaa testata. Tätä viestien lähetysjärjestystä kutsutaan *epäilyttäväksi* (suspect sequence). Tämän jälkeen testataan jokainen luokan olio invarianttitestauksen testitapauksilla ja tietyllä viestijärjestyksellä, mikä on valittu. Jos ollaan testaamassa off-pisteen arvolla oliota, pitäisi testauksen päättyä poikkeukseen, joka ei kuitenkaan muuta eikä vahingoita olion tilaa. On-pisteen testauksessa puolestaan olion pitäisi muuttua odotettuun tilaansa. Lopuksi vielä testataan kaikki aksessori-metodien, jotka palauttavat olion attribuuttien arvoja, lähettämät viestit, jotta niiden tuottamat arvot eivät ole ristiriidassa määrittelyjen mukaisten arvojen kanssa. [Bin99]

Kappaleessa 5.3.1 on määritelty testitapauksia *Kello*-luokan invarianteille. *Kello*-luokka on ei-modaalinen, joten sen olioita testattaessa valitaan metodien viestien lähetysjärjestys, joita testataan invarianttitestauksen testitapauksilla. Määritelmä-käyttö –järjestyksen mukaan esimerkiksi sekunteja testattaessa kutsuttaisiin ensin AsetaSek-metodia ja sen jälkeen heti PalautaSek-metodia. Tällöin havaitaan heti, onko sekuntit asetettu oikein. Sama tehdään minuuteille ja tunneillekin. Testattaessa luokan olioita satunnaisella viestien lähetysjärjestyksellä kutsutaan metodeita luonnollisesti satunnaisessa järjestyksessä. Lopuksi kuitenkin pitää muistaa tarkistaa, että olion tila on muuttunut oikeaksi testin aikana. Tämä tehdään aksessori-metodien kautta. Kello-luokan eri metodien järjestyksistä ei saada sellaisia, jotka olisivat kovin epäilyttäviä. Voidaan kuitenkin testata aksessori-metodien kautta, että kelloOff-metodin kutsun jälkeen kello ei todellakaan käy.

5.3.3 Quasi-modaalisen luokan testaus

Quasi-modaalisissa luokissa viestin lähetys on rajoitettu tietyissä tiloissa. Jos edellinen viesti muuttaa ratkaisevasti olion tilaa, seuraava viesti saattaa olla rajoitettu. Esimerkiksi pino-oliolla ei ole missään tilassa väliä viestien järjestyksellä, ennen kuin joku viesti muuttaa olion tilaa ratkaisevasti. Pino voi tulla täyteen, jolloin pinon lisäysviestiä ei enää hyväksytä tai pinon ollessa tyhjä siitä ei voida enää poistaa mitään. Tehokkaan testauksen täytyy ottaa huomioon nämä erikoistilanteet, mutta tietenkin myös tavalliset tilanteet. Testauksessa ollaan enemmän kiinnostuneita näistä erikoistilanteista kuin esimerkiksi satunnaisesti valittujen alkioiden lisäämisestä pinoon. [Bin99]

Esimerkkinä quasi-modaalisista luokista on seuraavaksi esitetty Set-luokan metodien esittelyrivit:

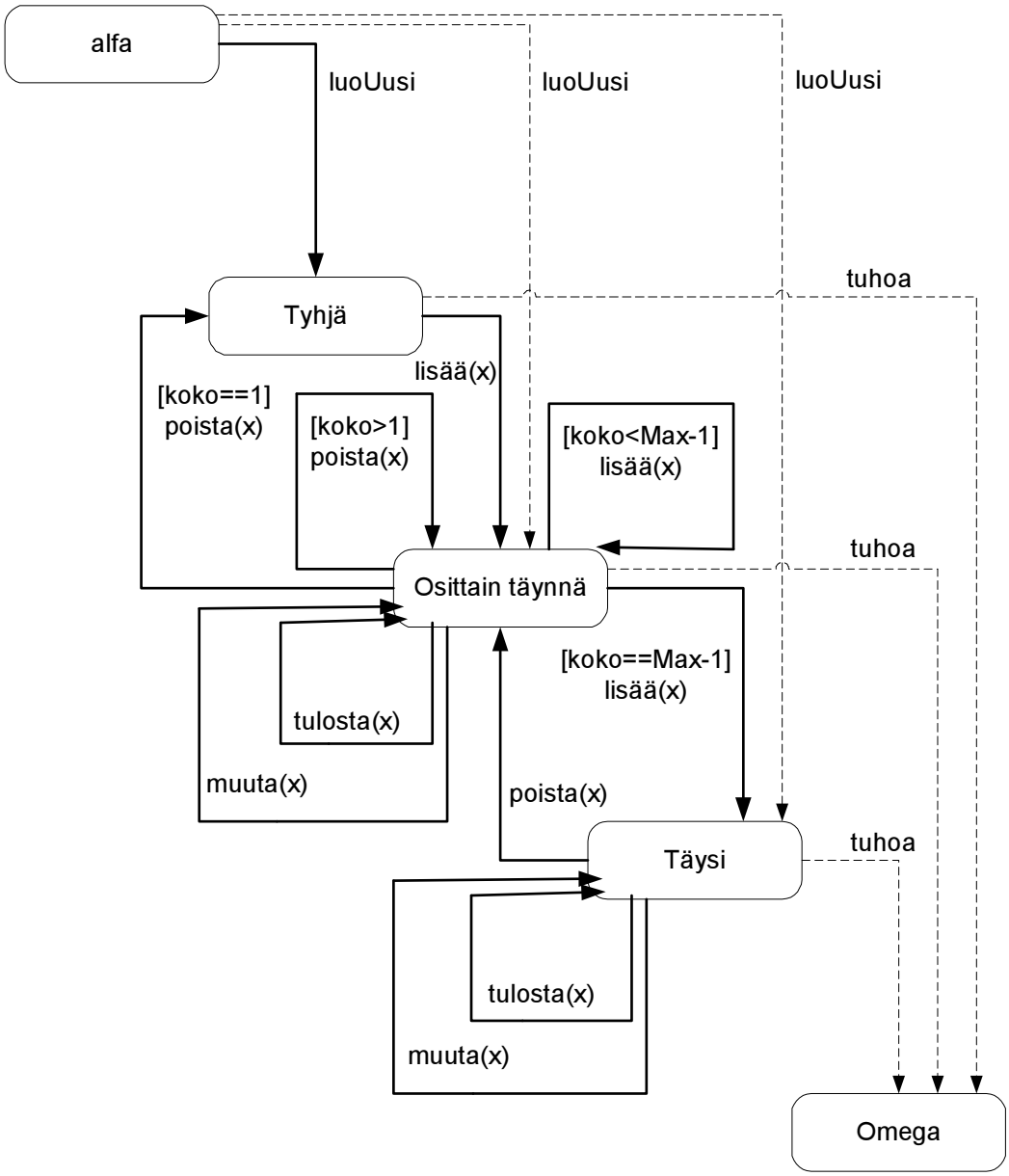
```
class Set{
  public Set() {}
  public boolean add(int i) {}
  public boolean remove(int i) {}
  public boolean clear(Set s) {}
  public boolean isMember(int I) {}
  public boolean isEmpty(Set s) {}
 }
```

Virheitä tällaisissa luokissa syntyy, kun metodeihin liittyviä rajoituksia ei oteta huomioon, eikä testata viestijärjestyksiä huolellisesti eri tiloissa. Oletetaan, että edelliseen Set-joukkoon ei voi lisätä kahta samaa alkiota. Esiehtona siis add(int i) –viestille on, että alkio i ei jo ole joukossa mukana. Testattaessa duplikaattivirhettä täytyy viesti add(5) lähettää kahteen kertaan. Ensimmäinen viesti pitäisi hyväksyä ja toinen hylätä. Oletetaan, että toisen viestin lähettämisestä seuraa virheilmoitus, mutta siitä huolimatta alkio lisätään virheellisesti joukkoon. Tätä virhettä ei huomata ennen kuin lähetetään viesti remove(5) ja tämän jälkeen viestin isMember(5) tulos olisi silti true. Tällaisen virheen paljastamiseen vaadittaisiin siis seuraavanlainen viestiketju:

add(5)
add(5)
remove(5)
isMember(5) = true

Quasi-modaalisen luokan testausmalli jakaantuu kolmeen osaan. Ensimmäiseksi piirretään tilakaavio luokasta tai käytetään jotain yleistä mallia kokoelmaluokista. Malli kuvaa luokan käyttäytymistä. Kuvassa 5.1 on yleinen tilakaavio kokoelmaluokille. Siinä on neliöissä esitetty eri tilat ja nuolilla eri tilasiirtymät. Erikoistilat alpha ja omega mallintavat alkutilaa ja tuhoamisen jälkeistä

tilaa. Katkoviivoilla on kuvattu piirteet, jotka ovat mahdollisia joissakin kokoelmissa. Yleisin tapa on luoda ensin tyhjä kokoelma, mutta joissain on mahdollista luoda kokoelmaan valmiiksi myös alkioita. Samoin vanhanaikaisemmat kokoelmat eivät salli tuhota kuin täysin tyhjän kokoelman, kun taas uudemmissa täydenkin kokoelman tuhoaminen on sallittua. [Bin99]



Kuva 5.1: Yleinen tilakaavio kokoelmaluokille. [Bin99]

Seuraavaksi käytetään invarianttitestausta määrittämään testitapauksia parametreille ja parametrien eri suhteille, jotka määräävät luokan käyttäytymistä. Testitapausten määrittelemisessä käytetään hyväksi tilakaavioita, joka on piirretty luokasta. Tämän avulla tunnistetaan tarvittavat parametrit, jotka pitää testata. Pyritään siis testaamaan kaavion tilasiirtymiä. Esimerkiksi Set-luokan tapaukses-

sa kiinnostavia parametrejä olisi joukon maksimaalinen koko ja tämän hetkinen alkioiden lukumäärä. Testataan siis luokan oliota vaihtelemalla joukon maksimaalista kokoa ja alkioiden määrää joukossa. Invarianttitestauksen mukaan saadaan kummallekin parametrille testitapauksiksi minimi-1, minimi, minimi+1, satunnaisesti valittu arvo, maksimi-1, maksimi ja maksimi+1. Joukon maksimaalisen koon ja alkioiden lukumäärän minimi- ja maksimi-arvot vaihtelevat tietenkin kulloistenkin määrittelyjen mukaan. Yleensä alkioiden lukumäärä voi olla pienimmillään nolla ja suurimmillaan joukon maksimaalisen koon verran. Joukon maksimaalinen koko taas vaihtelee yleensä yhdestä ylöspäin. Testattaessa Set-luokan oliota alkioiden lukumääräksi syötetään kaikki edellä määritellyt testitapaukset: alkioiden lukumäärän minimi, minimi-1, minimi+1, satunnaisesti valittu alkioiden lukumäärä sekä lukumäärän maksimi, maksimi-1 ja maksimi+1. Samalla vaihdellaan joukon maksimaalista kokoa vastaavilla joukon koon minimi-, maksimi- ja satunnaisesti valituilla arvoilla. Näin jatketaan kunnes kaikki kombinaatiot kahden parametrin testitapauksista on testattu. [Bin99]

Lopuksi tehdään tärkein osuus. Luokan erikoistilanteista muodostetaan testitapauksia. Tehdään siis luokalle erilaisia viestiketjuja, joita kannattaa testata. Quasi-modaalisilla luokilla on yhteisiä piirteitä, joten yhtä mallia voi käyttää monen eri luokan testaukseen. Kuitenkin erojakin löytyy, esimerkiksi joukkoon ei voi lisätä kahta kertaa samaa arvoa, mutta pinoon saman arvon voi lisätä lukemattomia kertoja. Seuraavaksi on lueteltu joitain yleisiä kokoelmia ja erityisesti niille tarkoitettuja testausjärjestyksiä.

Peräkkäiset eli sekventiaaliset kokoelmat (sequential collections) sisältävät vaihtelevan määrän alkioita. Alkioita voi lisätä ainoastaan loppuun ja niihin pääsee käsiksi vain alkioiden lisäysjärjestyksessä. Esimerkiksi merkkijonot ja erilaiset puskurit ovat tällaisia. Kiinnostava testauskohde sekventiaalisisssa kokoelmissa on kerralla lisättävien tai haettavien alkioiden määrä. Näitä kannattaakin testata vaihtelevasti eri tiloissa oleviin kokoelmiin. Erikoistilanteita ovat luonnollisesti ääripäät eli yksi lisättävä elementti tai liian monta kerralla lisättävää elementtiä.

Järjestetyissä kokoelmissa elementit haetaan, lisätään ja poistetaan käyttäen jotain järjestysmallia, kuten pino, jono, puu tai lista. Näissä testaus kannattaa kohdistaa etenkin siihen, mihin kohtaan alkiota ollaan lisäämässä tai mistä sitä ollaan hakemassa tai poistamassa. Erikoistilanteita ovat tietenkin ensimmäinen ja viimeinen positio.

Joissakin kokoelmissa on yksiselitteinen tunnistin jokaiselle alkiolle. Näitä kutsutaan avain kokoelmiksi. Yksi esimerkki avain kokoelmasta on hajautetut taulukot. Avain kokoelmia täytyy testata hyvin laajasti monella eri testausketjulla. Taulukossa 5.2 on esitetty kaksi esimerkkiä testausketjusta. Siinä on testattu samaa operaatiota aina kaksi kertaa peräkkäin järjestyksessä lisäys, haku ja tuhoaminen.

	Operaatiot	x:n arvo	y:n arvo	Odotettu tulos
Testiketju 1	lisää(x), lisää(y)	mikä tahansa	sama kuin x:n arvo	y:n lisäys hylätään
	hae(x), hae(y)	mikä tahansa	sama kuin x:n arvo	hyväksytty
	tuhoa(x), tuhoa(y)	mikä tahansa	sama kuin x:n arvo	y:n lisäys hylätään
Testiketju 2	lisää(x), lisää(y)	mikä tahansa	eri kuin x:n arvo	hyväksytty
	hae(x), hae(y)	mikä tahansa	eri kuin x:n arvo	hyväksytty
	tuhoa(x), tuhoa(y)	mikä tahansa	eri kuin x:n arvo	hyväksytty

Taulukko 5.2: Testiketjut.

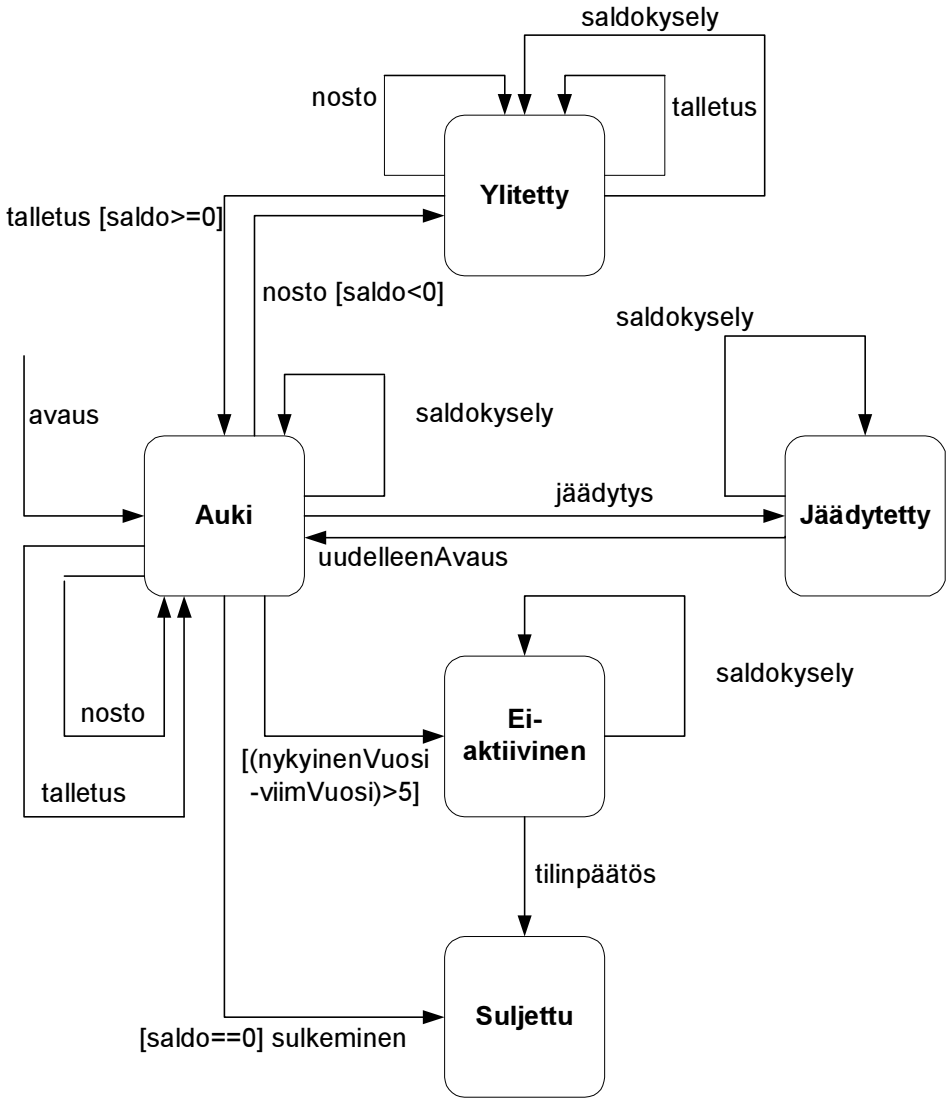
Quasi-modaalisessa luokkatestauksessa täytyisi saavuttaa ainakin päätöskattavuus. Tarkoituksena on testata luokasta piirretyn tilakaavion kiinnostavia tilasiirtymiä sekä lisäksi erilaisia viestiketjuja erikoistilanteista. Tämä malli paljastaa etenkin virheet puuttuvista tilasiirtymistä, puuttuvista tai vääristä toiminnoista ja vääristä tai laittomista tulostiloista.

5.3.4 Modaalisen luokan testaus

Modaalisissa luokissa metodien viestien järjestys on rajoitettu sekä viestin sisällöstä että olion tilasta riippuen. Vuorovaikutus viestien järjestyksellä ja olion tiloilla on monimutkainen ja tämän takia virhealtis. Testaajan täytyy testata kaikissa mahdollisissa tiloissa, että

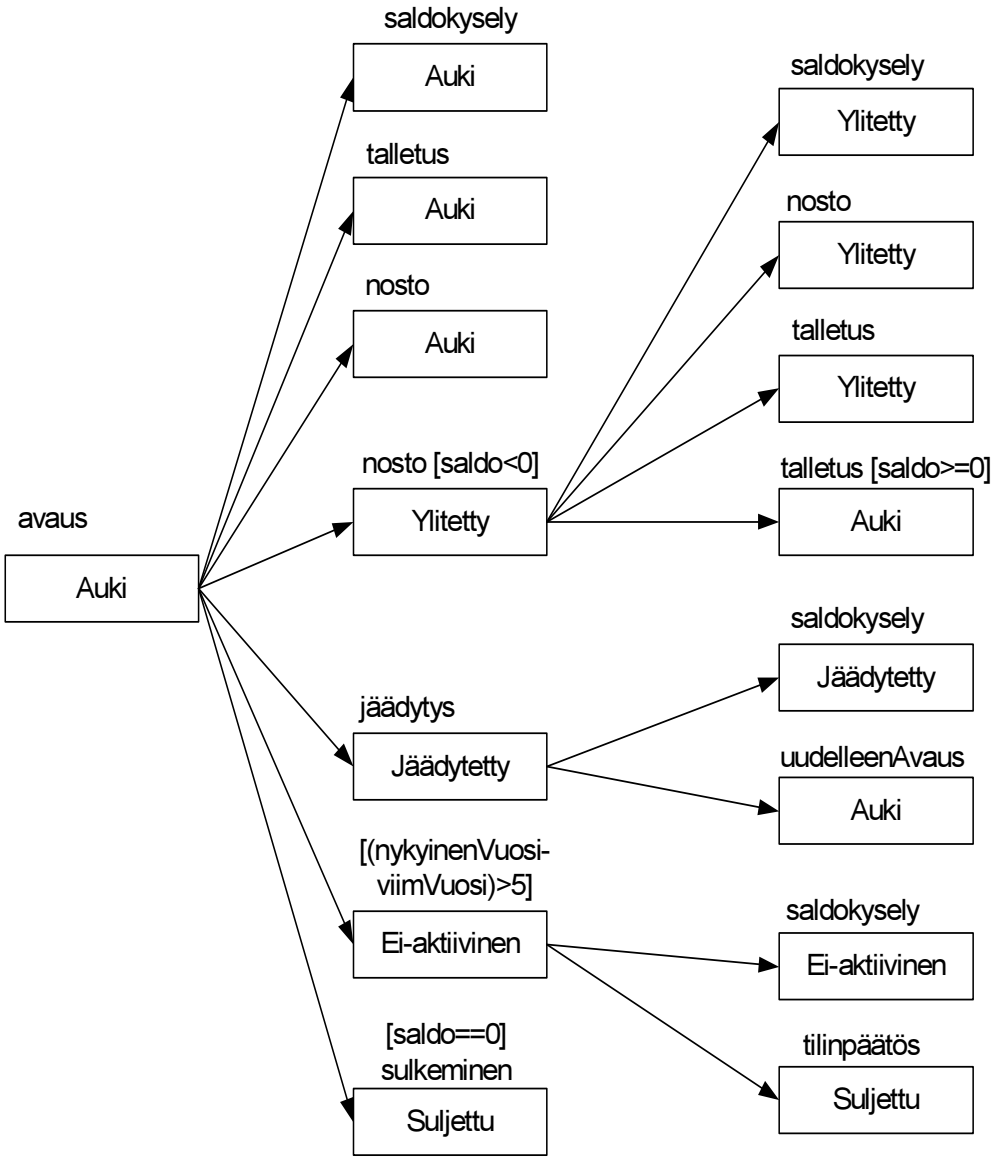
- viestit, jotka ovat hyväksyttäviä tietyssä tilassa ovat todellakin hyväksyttäviä
- viestit, jotka ovat laittomia tietyissä tiloissa, hylätään
- tulostila hyväksyttyjen ja hylättyjen viestien jälkeen on oikea
- jokaisen testattavan viestin tulos on oikea [Bin99].

Testitapausten suunnittelun ensimmäinen askel on piirtää tilakaavio. Sitä varten täytyy määrittää luokan olioiden eri tilat ja toiminnot, joilla päästään tilasta toiseen. Kuvassa 5.2 on esitetty *Tili*-luokan tilakaavio. Kuvassa luokan metodit on esitetty tapahtumina tilojen välillä eli ne kuvaavat, miten siirrytään tilasta toiseen. Tilat ovat *Auki*, *Ylitetty*, *Jäädytetty*, *Ei-aktiivinen* ja *Suljettu*. Nuolet tilojen välillä kuvaavat tilasiirtymiä. Nuolet on nimetty metodien nimillä, jotka kertovat, minkä tapahtuman johdosta siirrytään tilasta toiseen tai pysytään samassa tilassa. Esimerkiksi, kun *Auki*-tilassa suoritetaan nosto-metodi niin, että saldo muuttuu negatiiviseksi, siirrytään *Ylitetty*-tilaan.



Kuva 5.2: *Tili*-luokan tilakaavio. [Bin99]

Seuraavaksi tilakaaviosta kehitetään *siirtymäpuu* (transition tree). Puusta tilasiirtymät ovat nähtävissä vielä helpommin kuin tilakaaviosta. Tilasiirtymät näkyvät nuolina kuvassa. Puun juurisolmuksi tulee luonnollisesti alkutila. *Tili*-luokan tapauksessa alkutilana on *Auki*. Jokaiselle alkutilasta lähtevälle tapahtumalle piirretään oma viiva solmuun, joka kuvaa juuri sen toiminnon lopputilaa. Näin jatketaan kaikille tapahtumille, kunnes kaikki lopputilat on saatu puuhun. Mikäli tilasiirtymään liittyvä toiminto ei muuta tilaa, lopputilana on sama tila, kuin toiminnon alussa. Näin siirtymäpuussa voi esiintyä sama tila monta kertaa. *Tili*-luokan siirtymäpuu on esitetty kuvassa 5.3. [Bin99]



Kuva 5.3: *Tili*-luokan siirtymäpuu. [Bin99]

Tämän jälkeen testitapauksia muodostetaan siirtymäpuun avulla. Jokaisesta tapahtumapolusta alkutilasta lopputilaan tulee yksi testitapaus. Lisäksi jokaisesta yksittäisestä polusta yhdestä tilasta toiseen tulee testitapaus. Esimerkiksi *Tili*-luokan siirtymäpuusta tulisi mm. testitapaus, jossa *Auki*-tilassa olevalle oliolle lähetettäisiin *nosto*-metodin viesti eli tililtä otettaisiin raha, niin että saldo jäisi negatiiviseksi. Nyt tilan pitäisi vaihtua *Ylitetty*-tilaksi. Tämän jälkeen oliolle lähetettäisiin *talletus*-metodin viesti, niin että saldo nousisi positiiviseksi. Tällöin tilan pitäisi vaihtua takaisin *Auki*-tilaksi.

Siirtymäpuuta kannattaa vielä tarkistaa. Täytyy varmistua, että kaikki mahdolliset tilasiirtymät on otettu huomioon. Edellisissä testitapauksissa on aina jokin ehto, joka täytyy täyttyä. Esimerkiksi jotta tila muuttuisi *Auki*-tilasta *Ylitetty*-tilaksi, saldon pitää muuttua negatiiviseksi. Tapaukset voi-

daan koota totuustauluun. Taulukossa 5.3 on esimerkinomaisesti totuustaulu muutamista testitapauksista. Testauksessa täytyy ottaa huomioon tapaukset, joissa ehto toteutuu ja joissa ehto ei toteudu. Taulut helpottavat hahmottamaan kaikkia tarvittavia testitapauksia. Jo tehdyille testitapauksille saattaa siis tulla vielä lisätapauksia. Nämä testitapaukset lisätään jo piirrettyyn siirtymäpuuhun.

Tila	Viesti	Jälkiehto	Odotettu tila
Ylitetty	talletus	saldo < 0	Ylitetty
Ylitetty	talletus	saldo > 0	Auki
Auki	nosto	saldo < 0	Ylitetty
Auki	nosto	saldo > 0	Auki
Auki	---	nykyinenVuosi-viimVuosi > 5	Ei-aktiivinen

Taulukko 5.3: Totuustaulu testitapauksista.

Alkuperäinen siirtymäpuu saattaa jo sisältää kaikki mahdolliset testitapaukset liittyen muuttujien ehtoihin. Siinä tapauksessa tietysti lisätestitapauksia ei tarvita. Esimerkiksi *Ylitetty*-tilassa *talletus*-metodin viestin jälkeen on kaksi eri toimintamahdollisuutta. Joko tili pysyy ylitettynä tai avautuu, jolloin tilin saldo tulee positiiviseksi. Nämä kummatkin tapaukset löytyvät jo alkuperäisestä siirtymäpuusta, joten lisäystä ei tarvitse tehdä. Kun taas *nosto*-metodin viesti *ylitetty*-tilassa olevalle oliolle, pitää estää. Jos tili on ylitetty, ainoastaan muutama nosto on silloin sallittu. Tämä tapaus puolestaan ei löydy alkuperäisestä siirtymäpuusta ja se on siis lisättävä.

Jos ehdossa olevat muuttujat määrittävät jonkin rajan, myös rajaehdot täytyy testata. Esimerkiksi *Tili*-luokan tilanteessa olio menee *Ei-aktiivinen* -tilaan, jos mitään toimintoa ei ole tapahtunut viiteen vuoteen. Tässä tapauksessa muuttuja määrittää rajan, joten testataan esimerkiksi, mitä tapahtuu, kun toimintoa ei ole ollut yli viitteen vuoteen ja tasan viiteen vuoteen. [Bin99].

Siirtymäpuun avulla voidaan helposti todeta, että testattava luokka todellakin toteuttaa sille määrätyt toiminnot. Puun avulla voidaan kuitenkin myös testata, että luokka ei tee mitään ylimääräisiä tai laittomia toimintoja. *Laittomasta siirtymästä* (illegal transition) puhutaan, kun testattava luokka jossain laillisessa tilassa hyväksyy viestin, jota se ei saisi hyväksyä siinä tilassa. *Laiton viesti* (illegal message) puolestaan on ihan laillinen viesti, jota ei vain saisi hyväksyä jossain tietyssä tilassa. Jos viesti hyväksytään tällaisessa tilassa, siitä seuraa laiton siirtymä. *Käärmepolku* (sneak path) on virhe, joka sallii laittoman viestin ja tuottaa laittoman siirtymän. Tällainen käärmepolku on mahdollinen jokaiselle viestille, jota ei saisi hyväksyä jossain tietyssä tilassa. Käärmepolkuja testataan luonnollisesti lähettämällä laittomia viestejä. Esimerkiksi *Tili*-luokassa *tilinpäätös*-metodin viesti voidaan hyväksyä vain *Ei-aktiivinen*-tilassa. Olisi väärin hyväksyä se *ylitetty*-tilassa, koska silloin tili suljettaisiin, vaikka sen saldo olisi negatiivinen. Siis yksi käärmepolku testitapaus tulisi, kun

testattaisiin *tilinpäätös*-metodin viestin lähettämistä *Ylitetty*-tilassa. Taulukossa 5.4 on esitetty kaikkien mahdollisten käärmepolkujen paikat *Tili*-luokassa [Bin99].

Jokainen mahdollinen käärmepolku täytyy siis testata. Oliot asetetaan eri tiloihin ja yritetään lähettää kaikki mahdolliset laittomat viestit. Jotta testi menisi läpi, viesti pitäisi hylätä ja olion tila ei saisi muuttua. Testatessa riittää testata vain jokaisen tilan omia käärmepolkuja. Laittoman viestin lähettämisestä heti toisen laittoman viestin lähettämisen jälkeen ei tarvitse huolehtia aikaisempien testien jälkeen. Näin kaikki mahdolliset käärme polut tulevat paljastettua. [Bin99].

Tapahtuma	**Tila**				
	Auki	Ylitetty	Jäädytetty	Ei-aktiivinen	Suljettu
avaus	*	*	*	*	*
talletus			*	*	*
nosto			*	*	*
saldokysely					*
jäädytys		*	*	*	*
uudelleenAvaus	*			*	*
tilinpäätös	*	*	*		*
sulkeminen		*	*	*	*

* = mahdollinen käärmepolku

Taulukko 5.4: Mahdolliset käärmepolkujen paikat *Tili*-luokassa.[Bin99]

Mallissa siis kehitetään testitapauksia kahdella eri tavalla; suoraan siirtymäpuusta tulevat tapaukset sekä käärmepoluista tulevat tapaukset. Lisäksi totuustaulun tekeminen testitapauksista saattaa auttaa paljastamaan vielä lisää testitapauksia. Modaalisen luokan testausmallin on quasimodaalisen luokan testausmallin tavoin todistettu paljastavan luokasta puuttuvat siirtymät, puuttuvat tai väärät toiminnot ja väärät tai laittomat lopputilat.

6 Periytymisen testaus

Yksi monista eduista olio-ohjelmoinnissa on periytyvyys. Olio-pohjaisissa kielissä voidaan määritellä luokkia ja niille alaluokkia. Alaluokat automaattisesti perivät yläluokkansa. Yläluokan piirteet voivat suoraan periytyä alaluokalle tai alaluokka voi määritellä uudelleen jonkin yläluokan metodin toteutuksen. Ainoastaan yksityiset private-määreellä määritellyt piirteet eivät periydy, vaan ovat vain oman luokkansa käytössä. Nämä alaluokkien ja yläluokkien suhteet tuovat paljon lisävaatimuksia testaukseen. Joudutaan miettimään, mitä luokkia, luokkien välisiä suhteita, metodeita ja attribuutteja pitää testata ja kuinka paljon.

Luonnollisesti metodit, jotka määritellään vasta alaluokassa, sekä näiden viestinvälitys muiden metodien kanssa täytyy testata, mutta miten menetellään perittyjen piirteiden kanssa. Pitääkö perityt metodit testata uudestaan vai voidaanko luottaa yläluokassa tehtyyn testaukseen? Tätä asia täytyy tarkastella testattaessa luokkahierarkioita. Monesti voidaan vähentää testitapausten määrää, kun selvitetään periytymissuhteet tarkasti. Sitä, mitä ollaan jo testattu yläluokassa, ei välttämättä tarvitse testata enää uudelleen alaluokassa. Tähän ei saa kuitenkaan aina luottaa. Usein perityt piirteetkin tarvitsevat uudelleentestausta uudessa ympäristössä.

6.1 Periytymissuhteissa esiintyvät ongelmat

Luokkahierarkioiden ohjelmointi voi olla hyvin vaikeaa ja monimutkaista. Tämän takia se on myös virhealtista. Testauksen lähtökohtana on luonnollisesti testata asioita, joissa on todettu tulevan usein virheitä. Periytymissuhteissa tällaisia kohtia on paljon. Ala- ja yläluokan välissä voi olla jotain ongelmia. Alaluokka ei hyväksy kaikkia viestejä, jotka yläluokka hyväksyy tai alaluokka jättää olion sellaiseen tilaan, joka on laiton yläluokassa. Alaluokka saattaa laskea jonkin arvon, joka on ristiriidassa yläluokan muuttujien kanssa. Alaluokka voi myös luoda muuttujille uudet rajat, jotka eivät ole yhteneviä yläluokassa määriteltyjen alueiden kanssa. Alaluokka ei siis saisi millään tavoin särkeä yläluokan tila-avaruutta. [Bin99]

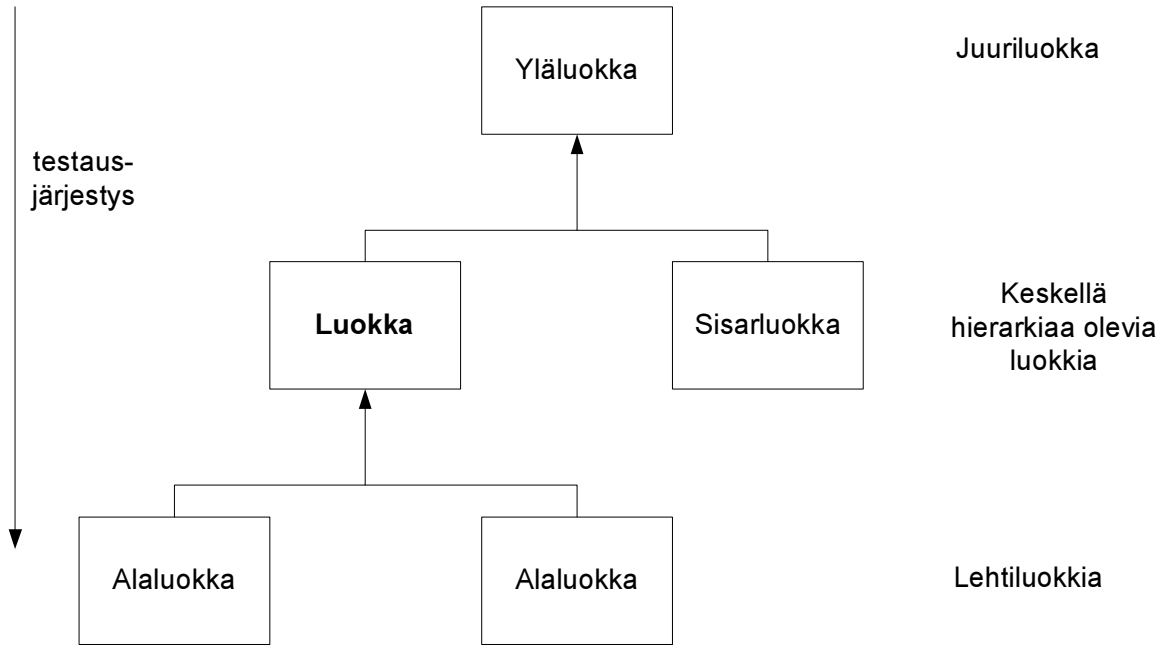
Ongelmia voi myös esiintyä, jos yläluokan toteutusta muutetaan huolimattomasti. Yläluokan muuttujathan ovat alaluokan käytettävissä, jos näin on määritelty. Alaluokan metodit voivat päivittää näitä muuttujia suoraan. Jos yläluokan toteutusta muutetaan, se saattaa aiheuttaa alaluokan toimintaan virheitä. Nämä virheet heijastuvat taas yläluokan metodeihin. Alaluokkaa ei voida edes toteuttaa ilman yläluokan toteutuksen tutkimista. Esimerkiksi, jos ollaan uudelleen määrittelemässä jotain yläluokan metodia, täytyy ensin tutkia käyttääkö yläluokan metodi mitään muita metodeja. Jos käyttää, täytyy miettiä, pitäisikö nämä käytetytkin metodit määritellä uudelleen. Tätä kutsutaan särkyvän yliluokan ongelmaksi [FrF01]. Testauksen kannalta tämä täytyy ottaa huomioon lisäämällä testausta. Jos yläluokkaa on muutettu, täytyy testata sekä ylä- että alaluokka. Jos ollaan taas muutettu alaluokkaa, täytyy testata ylä- ja alaluokan piirteet käytettynä alaluokasta. [Bin99]

Lisäksi itse periytymishierarkian rakentaminen voi tuottaa hankaluuksia. Monesti alaluokka on saatettu sijoittaa hierarkiaan väärin. On ajateltu luokan olevan tietyn luokan alaluokka, vaikka itse asiassa luokkien kuuluisi olla esimerkiksi samalla tasolla hierarkiassa. Myös liian syvien hierarkioiden rankentaminen on virhealtista. Tällöin toteutuksesta tulee liian monimutkainen. Periytymissuhteita ei saa rakentaa väärin perustein. Esimerkiksi jos luokka tarvitsisi metodia, joka löytyisi toisesta luokasta, ei luokkaa kannata sijoittaa toisen luokan alaluokaksi, jos luokilla ei ole mitään muuta

yhteistä. Tällöin luokat sijoitetaan eri hierarkioihin ja luokka kutsuu tarvitsemaansa metodia toisesta luokasta. [Bin99]. Luokan attribuuttien ja metodien suunnitteluun kannattaa kiinnittää huomiota. Periytymishierarkian suunnittelussa pitää muistaa periytymisessä käytössä oleva luokkien *is-kind-of –suhde*. Sen mukaan jokainen alaluokan olio on eräänlainen yläluokan olio. Esimerkiksi jos yläluokkana on *Potilas*, voisi sen alaluokkana olla *Syöpäpotilas*. Syöpäpotilas on tietynlainen potilas.

6.2 Periytymishierarkian perustestaus

Periytymishierarkiassa voidaan ajatella olevan kolmenlaisia luokkia. Hierarkian ylimmällä tasolla on kaikkein ylin luokka, juuriluokka (superclass), joka ei periydy mistään toisesta luokasta. Keskellä hierarkiaa on luokkia, joilla on yläluokka sekä alaluokkia. Ihan hierarkian alimmat luokat, lehtiluokat, voidaan erottaa myös omaksi ryhmäkseen, koska niillä on olemassa yläluokka, mutta ei alaluokkia. Jokainen luokka tunnistaa yläluokkansa, mutta ei alaluokkiaan. Tämä jaottelu auttaa testauksen suunnittelussa. Jokaiselle ryhmälle on oma malli testata luokkia. Kuvassa 6.1 on nimetty muut luokat yhden luokan suhteen.



Kuva 6.1: Luokkien nimityksiä.

Periytymishierarkian testaus aloitetaan hierarkian ylimmästä luokasta. Tämä luokka ei periydy mistään, joten periytymissuhteita ei tarvitse testata. Juuriluokan testaaminen on siis helpompaa verrattuna muihin luokkiin. Toisaalta sen testaaminen on tärkeintä, koska kaikki juuriluokassa olevat

virheet periytyvät muihin luokkiin. Juuriluokalle suoritetaan yksittäisten metodien testaus, jonka jälkeen yksittäisen luokan testaus. Jos luokan attribuutit ja metodit ovat yhteydessä luokan ulkopuolelle, tehdään tarvittaville attribuuteille ja metodeille apuohjelmia.

Ylimmän luokan testauksen jälkeen testaus etenee hierarkiassa alaspäin. Seuraavaksi testataan siis juuriluokan alaluokat. Näiden luokkien testauksessa täytyy ottaa huomioon yläluokasta perityt piirteet. Hierarkian keskellä olevien luokkien testaus aloitetaan testaamalla uudet metodit, jotka on määritelty testattavassa luokassa. Näille metodeille tehdään yksittäisten metodien testaus. Jos nämä uudet metodit ovat yhteydessä perittyihin piirteisiin, tehdään vielä testauksen tässä vaiheessa avuksi perityt piirteet korvaavat apuohjelmat. Seuraavaksi testataan metodit, jotka on peritty yläluokasta, mutta ylikirjoitettu alaluokassa. Näiden metodien toteutus on siis muuttunut. Metodeille tehdään yksittäisten metodien testaus. Testitapausten suunnittelussa voidaan käyttää apuna kyseisen metodin yläluokassa tehtyjä testitapauksia. Jos metodin testaus ei ole muuttunut ratkaisevasti, voidaan hyödyntää suoraan jo joitakin valmiita yläluokassa tehtyjä testitapauksia. Suoraan yläluokasta periytyville metodeille ei tarvitse tehdä yksittäisten metodien testausta enää alaluokassa, koska ne on jo testattu yläluokassa. Yläluokasta perityt metodit ovat kuitenkin yhteydessä alaluokassa määriteltyihin uusiin metodeihin ja attribuutteihin. Näiden metodien välisiä viestinvälityksiä ei ole vielä testattu, joten uusien metodien ja perittyjen metodien yhteentoimivuus täytyy testata. Hierarkian keskellä oleville luokillekin tehdään siis yksittäisen luokan testaus, jossa ei käytetä enää apuohjelmia perityille piirteille. Luokkatestauksessa pyritään testaamaan perittyjen piirteiden toimivuus alaluokan ympäristössä ja alaluokan metodien viestinvälitykset sekä luonnollisesti attribuutit.

Hierarkian alimpien luokkien testaus ei paljon eroa keskellä olevien luokkien testauksesta. Alimmilla luokilla ei ole enää alaluokkia, mutta luokka tiedostaakin vain yläluokkansa ei alaluokkia. Näiden luokkien testaus etenee siis hierarkian keskellä olevien luokkien tapaisesti. Ainoa asia, mikä täytyy huomioida alimpien luokkien testauksessa, on abstraktit piirteet. Alimmissa luokissa ei ole järkevää määritellä abstrakteja metodeja, koska ei ole enää alaluokkia, joissa määriteltäisiin metodien toiminta.

6.3 Testauksen riittävyys

Testauksen riittävyyteen on kehitetty useita sääntöjä, joita kutsutaan aksioomiksi. *Laajentamattomuus* (antiextensionality), *osiin jakamattomuus* (antidecomposition) ja *yhdistämättömyys* (anticomposition) -aksioomat pitävät paikkansa myös periytymissuhteiden testauksessa. Laajentamattomuusaksiooman mukaan testitapaus, joka kattaa jollakin tietyllä tavalla toteutetun ohjelman osan, ei välttämättä kata eri tavalla toteutettua saman ohjelman osaa. Siis vaikka on kyse samojen määritelmien

mukaan tehdyistä kahdesta eri osasta, jotka on toteutettu eritavoin, täytyy luonnollisesti kummallekin tehdä omat testitapaukset.

Osiin jakamattomuus -aksiooma taas sanoo, että testit, jotka kattavat tietyn moduulin testauksen, eivät välttämättä kata niiden moduulien testausta, joita on testattavan moduulin osana. Toisin sanoen jonkin ohjelman riittävä testaus ei välttämättä takaa sitä, että kyseisen ohjelman komponentit olisi riittävästi testattu. Jokaisen erillinen osa, jota saatetaan tulla käyttämään jossain muussa ympäristössä, pitäisi testata erikseen.

Yhdistämättömyys-aksiooma puolestaan kertoo, että riittävä testaus jokaiselle ohjelman komponentille, ei takaa sitä, että koko ohjelma olisi riittävästi testattu. Jokaisen komponentin yksikkötestauksen jälkeen pitää siis vielä testata vuorovaikutus muiden komponenttien kanssa. Oliopohjaisissa ohjelmissa ei siis voida luottaa alaluokkaan, joka on läpäissyt vain alaluokan testauksen, vaikka yläluokkakin olisi testattu jo erikseen. Tämän vuoksi yhdistämättömyys-aksiooma on tärkein aksiooma. [HaM92]

Näiden sääntöjen toteutuminen on helposti nähtävissä periytymissuhteiden testauksessa. Testaus erityisesti muutosten jälkeen on tärkeää. Jos muutetaan joitakin metodeita yläluokasta, täytyy luonnollisesti testata muuttuneet metodit ja niiden vuorovaikutukset muiden luokan metodien kanssa. Luokan testausta alaluokkansa kanssa ei silti saa unohtaa, vaikka alaluokkaan ei olisikaan tullut muutoksia. Yhdistämättömyys-aksiooman mukaan muuttuneet yläluokan metodit täytyy vielä testata myös alaluokan kanssa. Sama pätee, kun tehdään muutoksia tai lisäyksiä alaluokkaan yläluokan säilyessä muuttumattomana. Yhdistämättömyys ja osiin jakamattomuus -aksioomien mukaan täytyisi testata uudestaan yläluokan metodit, jotka alaluokka perii, vaikka nämä kyseiset metodit eivät olisikaan muuttuneet. Lisäksi uusi metodi pitäisi testata kaikkien luokkahierarkiassa olevien metodien kanssa mielenkiintoisissa eri järjestyksissä. Esimerkiksi kehitetään luokka *Tili* ja testataan se. Myöhemmin tehdään luokalle alaluokka *Osaketili*.

```
class Tili {
   double saldo;
   void asetaSaldo(double s) {
     saldo = s;
   }
}
class Osaketili {
   double osakearvo;
   void asetaOsakeSaldo(int osakemäärä) {
     saldo = osakemäärä*osakearvo;
   }
}
```

Epähuomiossa jää huomaamatta, että molemmat metodit *asetaSaldo* ja *asetaOsakeSaldo* muuttavat samaa yläluokan muuttujaa. Tätä virhettä ei tulla löytämään ennen kuin testataan molemmat metodit *asetaOsakeSaldo* ja jo aikaisemmin testattu *asetaSaldo* alaluokasta käsin. [Bin99]

Myös ylikirjoitettujen metodien testauksessa tulee olla huolellinen. Oletetaan, että lisätään alaluokkaan metodi, joka ylikirjoittaa yläluokan vastaavan metodin. Luonnollisesti uusi metodi täytyy testata, mutta voidaanko käyttää uudestaan samoja testitapauksia, joita ollaan käytetty testattaessa yläluokan vastaavaa metodia. Laajentamattomuus-aksiooman mukaan, vaikka kahden samaan tarkoitukseen tehdyn metodin toteutukset olisivat vain vähänkin erilaiset, niille eivät käy samat testitapaukset. Tässä tapauksessa testitapausten uudelleenkäyttö ei siis onnistu, vaan alaluokan metodi täytyy testata ihan omilla testitapauksilla. Alaluokassa olevat ylikirjoitetut metodit voivat helposti aiheuttaa sivuvaikutuksia yläluokasta perityille metodeille, vaikka perityt metodit olisivat muuttumattomia. [Bin99]

Oletetaan, että on olemassa yläluokka *Tili* ja sille alaluokka *Osaketili* ja kummatkin luokat on testattu kattavasti.

```
class Tili {
   laskeKorko(){
    // käyttää metodia saldo
   }
   saldo() {//…}
}
class Osaketili {
   saldo(){
    //ylikirjoitettu Tili-luokasta
   }
}
```

Luokan *Tili* metodi *laskeKorko* käyttää saman luokan metodia *saldo*. Myöhemmin metodi *saldo* ylikirjoitetaan ja sijoitetaan luokkaan *Osaketili*. Nyt kuitenkin luokasta *Tili* peritty metodi *laskeKorko* käyttää tätä uutta ylikirjoitettua metodia *saldo*. Ylikirjoitetussa metodissa saattaa olla esimerkiksi parametrien määrä eri alkuperäiseen verrattuna. Vaikka luokan *Tili* metodia *laskeKorko* ei ole muutettu, se silti käyttäytyy nyt eritavalla. Testauksessa tämä täytyy ottaa huomioon siten, että ei voida olettaa metodin *laskeKorko* käyttäytyvän oikein alaluokassa, vaikka se toimii yläluokassa hyvin ja mitään muutoksia tähän metodiin ei ole tehty. Uusi ylikirjoitettu metodi on siis testattava yhdessä kaikkien perittyjen metodien kanssa. Myös tässä asia pätee toisinkin päin. Abstraktin yläluokan ja alaluokassa olevien useiden ylikirjoitettujen metodien ollessa kyseessä, alaluokan metodien testausta ei saa unohtaa, jos yläluokka muuttuu. Vaikka yläluokka on abstrakti ja alaluokan me-

todeita ei olla muutettu, täytyy alaluokan metodit testata silti uudestaan yläluokan muuttuessa. [Bin99]

6.4 Inkrementaalinen testausmenetelmä

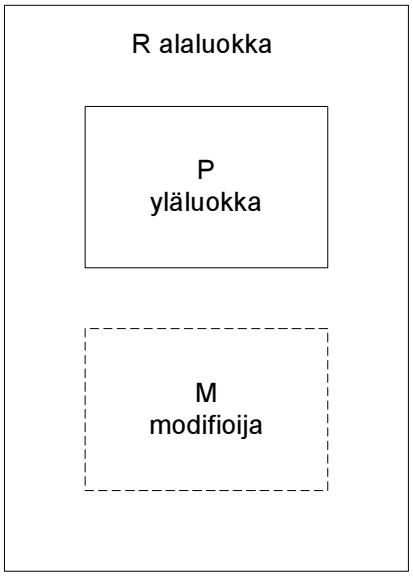
Mary Jean Harrold ja John D. McGregor ovat tutkineet ala- ja yläluokkien testaamista ja ovat kehittäneet yhden tavan testata periytymistä. He hyödyntävät yläluokkien testitapauksia uudelleen testattaessa alaluokkia. Kun alaluokkia testataan, käytetään siis yläluokan testitapaukset, joihin lisätään alaluokan vaatimia uusia testitapauksia. Uusien testitapausten määrä ja vanhojen uudelleenkäytettävyys riippuu alaluokan piirteiden tyypistä. Monet alaluokan perityt attribuutit täytyy silti testata uudelleen alaluokan omassa ympäristössä. [HaM92]

Menetelmä on hierarkkinen, koska sen testausjärjestystä ohjaa luokkien periytyvyyssuhteet. Menetelmä on myös inkrementaalinen, koska se käyttää hierarkian yhdeltä tasolta saatuja testituloksia hyväkseen myöhemmillä tasoilla vähentäen testausta alemmilla tasoilla. Menetelmä on tehty C++-kielelle, mutta se on hyvin joustava ja sopii myös kielille, joissa on samanlaisia piirteitä kuin C++:ssa. Kielen täytyy olla vahvasti tyypitetty ja tukea polymorfismia. Lisäksi kielessä olisi hyvä olla kolme tasoa piirteiden näkyvyydelle, esimerkiksi yksityinen, suojattu ja julkinen. Menetelmässä on myös oletettu, että luokalla voi olla vain yksi yläluokka. Seuraavassa piirteellä tarkoitetaan luokan muuttujaa tai metodia. [HaM92]

6.4.1 Määritelmät

Menetelmässä luokasta ajatellaan saavutettavan alaluokka modifioijan (modifier) avulla. Modifioija on abstrakti käsite, joka sisältää piirteet, jotka ovat tulossa alaluokkaan yläluokasta perittyjen piirteiden lisäksi. Modifioija määrittelee siis piirteet, jotka ovat erilaisia yläluokan piirteisiin verrattuna. Modifioijan ja yläluokan piirteet yhdistettynä määrittelevät alaluokan, kuva 6.2.

Vaikka modifioija onkin mukana alaluokan muodostamisessa, se ei kokonaan itsekseen määrittele alaluokkaa. Piirteiden periytymisessä täytyy ottaa huomioon myös piirteiden näkyvyysmääreet. Esimerkiksi Java- ja C++-kielissä avainsanat public (julkinen), protected (suojattu) ja private (yksityinen) määräävät piirteiden näkyvyyden alaluokassa.



Kuva 6.2: Alaluokka muodostuu modifioijasta ja yläluokasta.

Modifioija voi sisältää erityyppisiä piirteitä, jotka täytyy selvittää testausta varten. Seuraavaksi on lueteltu piirteiden eri tyyppejä:

uusi piirre: 1) A on piirre, joka on määritelty alaluokassa, mutta ei yläluokassa.

peritty piirre: A on määritelty yläluokassa, mutta ei alaluokassa. A viittaa paikallisesti yläluokassa määriteltyyn piirteeseen ja on käytettävissä alaluokasta.

uudelleenmääritelty piirre: A on määritelty sekä ylä- että alaluokassa. A:n toteutusta on kuitenkin muutettu alaluokassa.

virtuaalinen uusi piirre: 1) A on määritelty alaluokassa, mutta sen toteutus saattaa olla osittain määrittelemättä (virtuaalinen).

virtuaalinen peritty piirre: A on yläluokassa virtuaalinen ja A:ta ei ole määritelty alaluokassa. Tällöin A on sidottu yläluokassa paikallisesti määriteltyyn piirteeseen ja on käytettävissä alaluokasta.

virtuaalinen uudelleenmääritelty piirre: A on yläluokassa virtuaalinen ja A on määritelty myös alaluokassa samalla argumenttilistauksella kuin yläluokassa.

Taulukossa 6.1 havainnollistetaan erilaisten piirteiden periytymistä yläluokasta alaluokkaan. Vasemmalla luokka P kuvaa yläluokkaa, keskiosassa on mallinnettu modifioijaa ja oikealla kuvassa on alaluokan R piirteet. Luokalla P on kaksi attribuuttia i ja j sekä kolme metodia A, B ja C. Näistä metodeista B on virtuaalinen eli se voi ajonaikana korvautua alaluokkansa vastaavalla metodilla. Modifioijalla on yksi uusi attribuutti i, jota ei ole määritelty luokassa P. Lisäksi modifioijalla on metodit A ja C sekä virtuaalinen metodi B. A on uusi metodi, koska sen parametrit ovat erilaiset

verrattuna luokan P metodiin A. Virtuaalinen metodi B ja metodi C ovat uudelleenmääritelty modifioijassa. Tämän perusteella R luokan piirteiksi tulevat yksi attribuutti i sekä neljä metodia. Metodista A luokalla on kaksi toteutusta sekä luokasta P peritty että modifioijassa määritelty. Metodi B on virtuaalinen uudelleenmääritelty, koska se on määritelty uudelleen modifioijassa. Samoin metodi C on uudelleenmääritelty. Attribuutit i ja j ovat tavallaan perittyjä luokasta P, mutta kuitenkin piilotettuja luokassa R. Attribuutit ovat suojattu private-määreellä. Ne eivät siis näy alaluokalla. Niihin ei pääse käsiksi modifioijassa määritellyillä metodeilla, mutta niitä voi käyttää suoraan perityllä metodilla A. [HaM92]

Class P {
  private:
  int i;
  int j;
  public:
  P() {}
  void A(int a,int b)
  {I=a; j=a+2*b;}
 	
  virtual int B()
  {return i;}
  int C()
  {return j;}
};
	Modifioija
	  private:
	  float I;
	
	  public:
	  R() {}
	
	  void A(int a)
	  {return 5*a;}
	  virtual int B()
	  {return 3*i;}
	  int C()
	  {return 2*I}
	};
		class R {
		  private:
		  float i; //uusi
		
		  public:
		  R() {}
		  void A(int a,int b) //peritty
		  {i=a; j=a+2*b;}
		  void A(int a) //uusi
		  {return 5*a;}
		  virtual int B() //virtuaalinen-
		  {return 3*i;} //uudelleenmääritelty
		  int C() //uudelleenmääritelty
		  {return 2*i}
		};
		piilotetut:
		  Int i;
		  Int j;

Taulukko 6.1: Piirteiden periytyminen yläluokasta alaluokkaan.

6.4.2 Yläluokan testaus

Luokkien testaus aloitetaan yläluokista, joilla ei ole omia yläluokkia. Näiden luokkien metodit testataan ensin yksitellen ja sitten yhdessä muiden metodien kanssa. Yksittäiset metodit testataan käyttäen perinteisiä metodien testausmenetelmiä. Jokaista metodia pitää testata riittävästi, koska ne voivat periytyä alaluokkiin ja toimia siellä uudessa ympäristössä. Metodien testitapauksiin lukeutuukin sekä mustalaatikko- että lasilaatikkotestausmenetelmän mukaisia testitapauksia. Kaikki testitapaukset, ajonaikaiset tiedot ja metodien välillä olevat riippuvuudet tallennetaan testihistoriaan. Tämä testihistoria voidaan esittää seuraavasti: {mi,(TSi, test?), (TPi, test?)}, missä mi on testattava metodi, TSi on mustalaatikkomenetelmän mukaiset testitapaukset, TPi on lasilaatikkomenetelmän mu-

kaiset testitapaukset ja test? ilmaisee, ajetaanko testit vai ei. Yläluokan tapauksessa lähes kaikki suunnitellut testit luonnollisesti ajetaan, mutta alaluokka voi periä joitain testejä yläluokalta, jolloin test?-muuttujalla ilmaistaan, että testejä ei tarvitse ajaa uudelleen. Muuttuja test? voi saada kolmenlaisia arvoja: testit ajetaan kokonaan (Y), testit ajetaan osittain uudelleen (P) ja testejä ei ajeta ollenkaan (N).

Vaikka yksittäiset metodit olisi testattu huolellisesti, se ei takaa sitä, että koko luokkaa on riittävästi testattu. Yksikkötestauksen lisäksi täytyy testata metodien vuorovaikutusta muiden metodien kanssa. Täytyy siis tehdä integraatiotestaus. Tätä varten muodostetaan graafi, jossa solmut kuvaavat luokan metodeja ja kaaret metodikutsuja. Graafia käytetään apuna testitapausten luomisessa. Myös nyt tehdään sekä mustalaatikko- että lasilaatikkotestausmenetelmän mukaisia testitapauksia. Tämäkin testaus tallennetaan testihistoriaan, joka esitetään seuraavasti: {mi,(TISi, test?), (TIPi, test?)}, missä mi on muodostetun graafin yksi solmu eli metodi, joka kutsuu muita metodeja, TISi on mustalaatikkopohjaiset integrointitestitapaukset, TIPi on lasilaatikkopohjaiset integrointitestitapaukset ja test? ilmaisee, ajetaanko testit kokonaan (Y) tai osittain (P) uudelleen vai ei ollenkaan (N). [HaM92]

Seuraavassa esimerkissä on havainnollistettu testausmenetelmää luokan *Kuvio* avulla. Luokka on tehty eri kuvioiden piirtämistä varten. Alla on esitelty luokan attribuutti ja metodit, mutta toteutukset on jätetty pois. Jokaisella kuviolla on aloituspiste, josta piirtäminen aloitetaan. Luokan metodilla *asetaPiste* aloituspiste voidaan asettaa haluttuun kohtaan. Metodilla *siirrä* piirrettyä kuviota voidaan siirtää sen aloituspisteen avulla. Metodi kutsuu *poista*-metodia ja sitten *piirrä*-metodia. *Poista*-metodi poistaa kuvion kutsumalla *piirrä*-metodia, joka poistaa kuvion piirtämällä sen päälle. Metodi *piirrä* on virtuaalinen, joten se voi korvautua ajonaikana luokan alaluokkien vastaavalla metodilla. Metodin toteutusta ei ole tässä luokassa edes määritelty.

```
class Kuvio {
  Private:
    Point piste;
  Public:
    Kuvio();
    void asetaPiste(Point p);
    void siirrä(Point p);
    void poista();
    void piirrä();
}
```

Taulukossa 6.2 on esitetty *Kuvio*-luokan testaushistoria. Taulukon merkintätapa on edellä selitetyn menetelmän mukainen merkintä, eli TS tarkoittaa mustalaatikkotestauksen ja TP lasilaatikkotestauksen mukaisia testejä sekä Y tarkoittaa, että testitapaukset ajetaan ja N, että testitapauksia ei ajeta. Testaushistoria on hyvin yksinkertainen tässä vaiheessa, koska on kyse juuriluokasta. On siis testattava kaikki mahdollinen ja yhtään valmista testiä ei ole saatavilla. Kaikille metodeille luodaan mustalaatikko- ja lasilaatikkotestauksen mukaiset testitapaukset. *Piirrä*-metodin toteutusta ei ole vielä saatavilla, joten sille ei voida luoda lasilaatikkotestauksen testitapauksia. Mustalaatikkotestauksen testitapaukset voidaan puolestaan luoda, mutta niitä ei voida ajaa ilman toteutusta. Nämä kannattaa kuitenkin jo luoda, koska niitä voi käyttää sitten alaluokkien testauksessa hyväksi. Taulukossa 6.2 on myös nähtävillä metodien väliseen integraatiotestaukseen vaaditut testit. Metodit *siirrä* ja *poista* kutsuvat toisia metodeita, joten niille täytyy tehdä lisäksi integraatiotestauksen testitapauksia.

Piirre	Mustalaatikkotestauksen testitapaukset	Lasilaatikkotestauksen testitapaukset
Yksittäiset metodit		
Kuvio	(TS1, Y)	(TP1, Y)
asetaPiste	(TS2, Y)	(TP2, Y)
siirrä	(TS3, Y)	(TP3, Y)
poista	(TS4, Y)	(TP4, Y)
piirrä	(TS5, N)	(---)
Metodien välinen integraatio		
siirrä	(TIS6, Y)	(TIP6, Y)
poista	(TIS7, Y)	(TIP7, Y)

Taulukko 6.2: *Kuvio*-luokan testihistoria.

6.4.3 Alaluokkien testaus

Alaluokkien testaamisessa käytetään hyväksi yläluokkien testaamisessa tullutta testihistoriaa. Tämän testihistorian ja lisäksi modifioijan perusteella päätellään, mitkä piirteet pitää testata uudelleen alaluokassa ja mitä yläluokan testitapauksia voidaan käyttää uudelleen. Menetelmään on kehitetty algoritmi, joka muuntaa yläluokan testihistorian alaluokan testihistoriaksi. Algoritmi saa syötteenään yläluokan testihistorian, yläluokan integraatiotestausvaiheessa muodostetun graafin ja modifioijan. Tuloksena algoritmi antaa alaluokkaa varten päivitetyn testihistorian, jonka mukaan alaluokka testataan. Aikaisemmin kappaleessa 6.4.1 esitellyt kuusi piirretyyppiä, uusi, peritty, uudelleenmääritelty sekä virtuaaliset piirteet, vaikuttavat testihistorian syntyyn. Jokainen erityyppinen piirre täytyy testata eri tavalla ja yläluokan testihistoriaa voidaan käyttää erityyppisiin piirteisiin eritavalla hyödyksi. Algoritmin toiminta alkaa alaluokan testihistorian alustamisella yläluokan testi-

historialla. Alaluokan testihistoriassa kaikkiin testitapauksiin muutetaan vain test?-kentän arvoksi, että ei testata uudelleen. Alussa siis oletetaan, että alaluokan piirteitä ei tarvitse enää testata uudelleen, kun ne ovat jo testattu yläluokassa. Tämän jälkeen algoritmi tutkii jokaisen alaluokan piirteen erikseen ja päivittää testihistoriaa sen mukaan, tarvitseeko piirteille suorittaa uudelleentestausta alaluokassa ja tuleeko alaluokan piirteille kokonaan joitain uusia testejä. [HaM92]

Jos on kyseessä uusi tai virtuaalinen uusi piirre, täytyy se testata huolellisesti kokonaan, koska sitä ei ole määritelty yläluokassa eikä näin ollen vielä testattu ollenkaan. Piirre täytyy testata itsenäisesti sekä yhdessä muiden alaluokkien piirteiden kanssa. Uudelle piirteelle tehdään sekä mustalaatikko- että lasilaatikkopohjaiset testitapaukset. Nämä uudet testitapaukset lisätään alaluokan testihistoriaan ja niiden test?-kentän arvoksi tulee Y, eli piirteet tulee testata. Piirre yhdistetään lisäksi luokkagraafiin ja sille tehdään integraatiotestitapaukset. Jos piirre on virtuaalinen uusi –tyyppiä, generoidaan sille ainakin mustalaatikkomenetelmän mukaiset testitapaukset. Lasilaatikkopohjaiset testitapaukset tehdään, jos voidaan. Tällöin piirteen toteutuksen täytyy olla määritelty. Myös integraatiotestitapaukset voidaan tehdä vain, jos piirteen toteutus on määritelty. [HaM92]

Jos taas piirteen tyyppi on uudelleenmääritelty tai virtuaalinen uudelleenmääritelty, täytyy se testata huolellisesti uudestaan. Tässä tapauksessa pystytään kuitenkin käyttämään monia jo olemassa olevia, yläluokasta perittyjä, mustalaatikkopohjaisia testitapauksia osaksi uudestaan, koska vain piirteen toteutus on muuttunut. Testihistoriaan päivitetäänkin mustalaatikkopohjaisten testitapausten kohdalle, että testataan uudelleen. Yläluokassa tehdyt testitapaukset testataan siis uudelleen alaluokan ympäristössä. Lasilaatikkopohjaisia testejä ei voida käyttää uudestaan, koska testattavan toteutus on muuttunut. Uudet lasilaatikkomenetelmän mukaiset testitapaukset täytyy generoida ja lisätä testihistoriaan. [HaM92]

Peritty tai virtuaalinen peritty piirre ei puolestaan tarvitse kovinkaan paljon uudelleentestausta. Piirre peritään suoraan yläluokasta ja se on jo siellä testattu, eikä sen määritelmä ja toteutus ole muuttunut. Tässä tapauksessa ei siis tarvitse tehdä muutoksia testihistoriaan. Voidaan käyttää samoja testejä kuin yläluokallekin. Piirre voi kuitenkin olla vuorovaikutuksessa alaluokan uusien tai uudelleenmääriteltyjen piirteiden kanssa. Tällöin täytyy piirre testata sen uudessa ympäristössä eli alaluokassa. Piirteelle tehdään siis integraatiotestaus uusien ja uudelleenmääriteltyjen piirteiden kanssa, kun ne ovat liitetty alaluokkaan. [HaM92]

Luokan muuttujien näkyvyysmääreet voivat muuttua periytymisen aikana. Jos muuttujan näkyvyysmäärе muuttuu rajoittuneemmaksi, se ei voi olla vuorovaikutuksessa uusien tai uudelleenmääriteltyjen piirteiden kanssa. Tällöin muuttujaa ei tarvitse testata uudelleen. Jos muuttuja on näkyvä jollekin metodille, joka on määritelty alaluokassa, täytyy uusille metodeille ja vanhoille metodeille,

jotka vaikuttavat tähän muuttujaan, tehdä integrointitestaus. Tämä testaus suoritetaan, kun uudet metodit liitetään testausgraafiin. [HaM92]

Alaluokan testauksen havainnollistamiseksi on esitelty seuraavaksi kappaleessa 6.4.2 esitellyn *Kuvio*-luokan alaluokka *Kolmio*, joka on tehty kolmioiden piirtämistä varten. *Kolmio*-luokasta on esitetty metodien esittelyrivit, mutta toteutukset on jätetty pois. Luokassa määritellään kolmioiden kärkipisteet attribuutteina. Kärkipisteet voidaan asettaa metodien avulla. Metodi *asetaKärkipiste1* kutsuu yläluokan metodia *asetaPiste*, jonka avulla kolmion alkupisteelle saadaan koordinaatit. Luokassa määritellään myös yläluokasta peritty *piirrä*-metodi uudestaan. Muut toiminnot luokka perii suoraan yläluokaltaan.

```
Class Kolmio {
Private:
  Point kärkipiste1;
  Point kärkipiste2;
  Point kärkipiste3;
Public:
  Kolmio();           //uusi
  void asetaKärkipiste1(Point p);  //uusi
  void asetaKärkipiste2(Point p);  //uusi
  void asetaKärkipiste3(Point p);  //uusi
  void piirrä();                  //virtuaalinen uudelleenmääritelty
}
```

Taulukossa 6.3 on esitelty *Kolmio*-luokan testihistoria. Taulukon merkintätapa on edellä selitetyn menetelmän mukainen merkintä, eli TS tarkoittaa mustalaatikkotestauksen ja TP lasilaatikkotestauksen mukaisia testejä sekä Y tarkoittaa, että testitapaukset ajetaan, P tarkoittaa, että testitapaukset ajetaan osittain ja N, että testitapauksia ei tarvitse ajaa uudestaan. *-merkityt testitapaukset ovat uusia testitapauksia, joita siis ei ole ollut yläluokan testihistoriassa ja **-merkityt ovat osaksi uusia ja osaksi testitapauksia, jotka on peritty yläluokan testauksesta. Taulukosta nähdään, että suoraan perittyjen metodien *asetaPiste*, *siirrä* ja *poista* testitapauksia ei tarvitse ajaa uudelleen, koska niihin ei ole tullut mitään uudelleenmäärityksiä. Nämä metodit on jo testattu yläluokassa. Testit ovat merkitty N:llä, joka ilmaisee sen, että testejä ei tarvitse ajaa uudelleen. Virtuaalinen uudelleenmääritelty metodi *piirrä* on testattava uudelleen, koska sen toteutus on muuttunut. Kuitenkin ainoastaan lasilaatikkotestauksen testitapaukset on kehitettävä uudestaan, koska mustalaatikkotestauksen testit voidaan periä yläluokan testihistoriasta ja vain ajaa uudestaan. Kaikki kolme *Kolmio*-luokan kärkipisteen asetusmetodia täytyy tietenkin testata, koska ne ovat uusia piirteitä. Yläluokan testihistorias-

sa ei luonnollisesti näiden metodien testejä voi olla, joten näiden testit lisätään alaluokan testihistoriaan.

Metodien välisiin integraatiotesteihin täytyy lisätä metodi *asetaKärkipiste1*, joka kutsuu yläluokan *asetaPiste*-metodia. Metodit *siirrä* ja *poista* kutsuvat nyt *Kolmio*-luokan *piirrä*-metodia, joten uusia integraatiotestitapauksia näille metodeille täytyy luoda ja suorittaa. Yläluokan testihistoriassa saattaa kuitenkin olla testitapauksia, joita voidaan uudelleenkäyttää. P:llä merkityt testitapaukset tarkoittaa, että yläluokan testitapauksia voidaan ajaa osittain uudelleen. Integraatiotestausta on tarkemmin käsitelty kappaleessa 7.

Piirre	Mustalaatikkotestauksen testitapaukset	Lasilaatikkotestauksen testitapaukset
Yksittäiset metodit		
Kuvio	(TS1, N)	(TP1, N)
asetaPiste	(TS2, N)	(TP2, N)
siirrä	(TS3, N)	(TP3, N)
poista	(TS4, N)	(TP4, N)
piirrä	(TS5, Y)	*(TP5, Y)
Kolmio	*(TS6, Y)	*(TP6, Y)
asetaKärkipiste1	*(TS7, Y)	*(TP7, Y)
asetaKärkipiste2	*(TS8, Y)	*(TP8, Y)
asetaKärkipiste3	*(TS9, Y)	*(TP9, Y)
Metodien välinen integraatio		
siirrä	**(TIS10, P)	**(TIP10, P)
poista	**(TIS11, P)	**(TIP11, P)
asetaKärkipiste1	*(TIS12, Y)	*(TIP12, Y)

Taulukko 6.3: *Kolmio*-luokan testihistoria

Suurin hyöty tässä menetelmässä on, että alaluokan täydellinen testaus vältetään käyttämällä yläluokan testihistoriaa hyödyksi. Ainoastaan uudet ja uudelleenmääritellyt piirteet täytyy testata. Lisäksi yläluokan testitapaukset voidaan joissain tapauksissa ajaa uudelleen. Aina ei kuitenkaan edes tätä tarvitse tehdä, vaan voidaan luottaa testaukseen, joka on jo tehty yläluokalle. Menetelmä säästää aikaa uusien testitapausten suunnittelussa ja itse testien suorittamisessa. Parhaiten menetelmä soveltuu luokkahierarkioille, joissa ylimmässä luokassa on eniten toiminnallisuutta ja alaluokissa ainoastaan muutoksia ja lisäyksiä.

7 Integrointitestaus luokkien välillä

Kun yksittäiset metodit ja luokat sekä luokkien periytymishierarkiat on testattu, seuraava vaihe testauksessa on yhdistellä erillisiä luokkia ja testata niiden toimivuutta yhdessä. Vaikka yhdistettävät luokat on jo testattu erikseen, ne eivät välttämättä toimi oikein yhdessä. Luokkien välisessä testauksessa (inter-class testing) testataan vuorovaikutusta kahden tai useamman luokan välillä. Itse asiassa vuorovaikutuksessa keskenään ovat luokkien ilmentymät eli oliot. Ohjelman toimivuus riippuu siitä, miten oliot toimivat keskenään. Olioiden keskinäiset vuorovaikutukset ovat siis tärkeä testauskohde. Vuorovaikutusta testataan tarkastelemalla olioiden toisilleen lähettämiä viestejä ajonaikana. Vuorovaikutuksia testaamalla saadaan siis selvyyttä luokkien yhteisestä toiminnasta ja voidaan todentaa, että yksittäiset luokat toimivat hyvin myös yhdessä. [McS01]

7.1 Integraatiotestauksen apuohjelmat

Kappaleessa 4.2 on esitelty apuohjelmia, joita tarvitaan metoditestauksessa. Näitä apuohjelmia voidaan soveltaa myös luokkatestaukseen. Integraatiotestauksessa apuohjelmia täytyy kuitenkin hieman muuttaa verrattuna metoditestaukseen. Metoditestauksessa apuohjelmissa simuloitiin testauksen ulkopuolelle jääneitä metodeja. Integraatiotestauksessa tarkoituksena on yhdistellä useita luokkia, joten nyt testauksen ulkopuolelle jää kokonaisia luokkia. Näiden luokkien simuloimista varten tarvitaan apuohjelmia. [Paa00]

Periaate apuohjelmien tekemisessä on kuitenkin sama kuin metoditestauksen kohdallakin. Luokkia simuloimaan tarvitaan sekä tynkäluokkia, että ajureita. Apuohjelmista yritetään tehdä mahdollisimman yksinkertaisia. Tynkäluokkaa tarvitaan silloin, kun joku luokka on rajattu pois testauksesta, mutta sitä kuitenkin kutsutaan jostain testattavasta luokasta. Tällöin kutsuttavaa luokkaa varten täytyy tehdä tynkäluokka, joka simuloi alkuperäistä luokkaa. Nyt täytyy tosiaankin ottaa huomioon koko luokka, eikä vain yhtä metodia.

Esimerkkinä voisi olla *Potilas*-luokalle tehty tynkäluokka. *Potilas*-luokka sisältää paljon tietoa potilaasta. Luokassa on siis paljon erilaisia attribuutteja potilaan nimestä lähiomaisten tietoihin asti. Tynkäluokkaan tarvitaan ottaa näistä attribuuteista vain tarvittavat. Sellaiset attribuutit, joita tarvitaan välttämättömästi jossain luokan metodeista. *Potilas*-luokassa voisi olla esimerkiksi metodit: *annaPotilaanTiedot()*, joka palauttaa potilaan tiedot, *palautaLähiomaisenNimi()*, joka palauttaa potilaan lähiomaisen nimen ja *palautaOsastonlkm(int osastoNro)*, joka palauttaa osaston henkilöiden lukumäärän sille syötetyn potilaan osastonumeron perusteella. Kaikki oikeassa luokassa olevat metodit pitää olla myös tynkäluokassa. Metodien toteutusten ei tarvitse olla mitään monimutkaisia

tai edes oikeita. Yleensä tynkämetodin toteutus on vain pelkkä palautuslause. Pääasia, että metodi palauttaa oikean tyyppisen arvon. Näin testauksessa olevien luokkien testaus voi jatkua, kun tynkäluokka huolehtii testauksesta pois rajatun luokan toiminnoista. Alla on esitetty Potilas-luokalle tehty tynkäluokka.

```
class PotilasTynkä {

  int osastoNro;


  PotilasTynkä(int nro){                                  // Konstruktori
   osastoNro = nro;
  }


  String annaPotilaanTiedot(){
    return "Tila: Vakaa, Lääkitys: Vahva";
  }                                              // Palautetaan aina sama vastaus.


  String palautaLähiomaisenNimi(){               // Palautetaan aina sama vastaus.
    return "äiti";
  }

  int palautaOsastonlkm(int osastoNro){          // Palautetaan aina sama vastaus.
    return 12;
  }
}
```

Ajuria tarvitaan koko luokalle silloin, kun halutaan kutsua testattavaa luokkaa. Ajuri kutsuu testattavan luokan metodeja ja ottaa vastaan metodien palauttamia arvoja. Ajurilla testataan, miten testattava luokka ja sen metodit vastaavat niitä kutsuttaessa. Välttämättä tätä varten ei tarvita yhtä isoa ajuria, vaan testaus voidaan toteuttaa myös yksinkertaisilla ajureilla. Tällöin jokaista testattavan luokan metodia varten on tehty oma ajuri kappaleessa 4.2 kuvatun mukaisesti. Alla on esitetty *Potilasrekisteri*-luokkaa varten tehty ajuri. *Potilasrekisteri*-luokka säilyttää tietoja potilaista. Rekisteriin voi lisätä potilaan ja tulostaa kaikki potilaat. Luokalle tehty ajuri luo testattavan luokan. Sen jälkeen ajuri kutsuu *Potilasrekisteri*-luokan *lisää*-metodia, jonka jälkeen *tulosta*-metodia. Näin saadaan selville, onko *Potilasrekisteri*-luokan luonti onnistunut ja onko sinne lisätty potilas.

```
void potilasrekisteriAjuri{

   Potilas p;

    // Luodaan testattava luokka.
  Potilasrekisteri rekisteri = new Potilasrekisteri();

    //Kutsuu testattavan luokan lisää-metodia
  rekisteri.lisää(p);
```

```
    //Kutsuu testattavan luokan tulosta-metodia.
  Lista potilaslista = rekisteri.tulosta();
    return potilaslista;
}
```

7.2 Olioiden välinen vuorovaikutus

Olioiden vuorovaikutus on yksinkertaisesti viestien lähettämistä oliolta toiselle. Tämän viestin vaikutuksesta olioiden tilat voivat pysyä ennallaan tai muuttua jotenkin. Toiminto voi olla myös uuden olion luonti tai vanhan tuhoaminen. Kahden luokan välistä vuorovaikutusta testataan siis lähettämällä viestejä luokan ajonaikaisesta ilmentymästä toiselle. Olioiden tilojen tarkastaminen on tärkeä osa tätä testaamista. Oliot ovat aina jossain tilassa ja viestit niiden välillä voivat sisältää parametreina myös olioita, joilla on jokin tila. Viestien ja eri tilojen yhdistelmät saattavat aiheuttaa odottamattomia seurauksia. [McS01]

Jokaisen olion attribuuttien ja metodien omat näkyvyysmääreet saattavat aiheuttaa olion tilan testaamisessa hankaluuksia. Ainoastaan julkisia public-määreellä määrättyjä piirteitä pystytään tarkastelemaan olion ulkopuolelta. Monesti ajatellaan, että tämä riittääkin, koska yksittäiset luokat on kuitenkin jo testattu erikseen. Aina on kuitenkin parempi vielä tässäkin vaiheessa tarkastaa piilotettujenkin piirteiden tilat [McS01]. Oliopohjaisissa ohjelmissa luokkien väliseen testaukseen tuo omat hankaluutensa olioiden tilojen muutosten lisäksi polymorfiset metodit ja dynaaminen sidonta.

Oliopohjaisten ohjelmien integrointitestaus voidaan jakaa luokan sisäiseksi ja luokkien väliseksi testaukseksi. Luokan sisäisessä integrointitestauksessa testataan metodien viestinvälitystä luokan sisällä. Tätä on käsitelty tarkemmin kappaleessa 5. Luokkien välisessä testauksessa testataan luokkien välistä vuorovaikutusta viestinvälityksen kautta. Kuten huomataan, oliopohjaisten ohjelmien testauksessa ei voida asettaa selvää rajaa sille, milloin yksittäisten luokkien testaus loppuu ja milloin integrointitestaus alkaa. Yksittäisten luokkien testaaminen on jo periaatteessa integrointitestausta, koska siinähän testataan metodien toimimista yhdessä. Luokan metodit voivat myös lähettää viestejä luokan ulkopuolelle jolloin yksittäisen luokan testauksessakin tarvittaisiin toisia luokkia. Käyttämällä kappaleessa 4.2 sekä 7.1 mainittuja apuohjelmia voidaan kuitenkin luokan sisäinen testaus tehdä ennen kuin aloitetaan luokkien välinen testaaminen.

Luokkien välisessä testauksessa testataan viestin lähettävän metodin ja viestin saavan metodin välisiä rajapintoja, viestin toiminnallisuutta, olioiden välistä vuorovaikutusta ja tietenkin koko yhdistelmän toimivuutta yleensä. Testaus tapahtuu viesti kerrallaan. Olio-ohjelmissa viestin lähettämismahdollisuuksia on tuhansia, joten integrointitestaus on laaja vaihe olio-ohjelmien testauksessa. [McM94]

Tutkijat Leung ja White ovat luokitelleet virheet, joita ilmenee proseduraalisten ohjelmien integraatiotestauksessa, kolmeen luokkaan. Tämä sama luokittelu pätee myös oliopohjaisissa ohjelmissa. Virheluokat ovat ylimääräinen metodi, puuttuva metodi ja rajapintavirheet. Ylimääräinen metodi -virhe ilmenee, kun luokassa A on jotain ylimääräistä toimintaa, jota ei vaadita luokassa B. Kun A:sta lähetetään jokin tietty viesti tietyillä parametreillä B:lle, tämä ylimääräinen metodi B:ssä aktivoituu ja aiheuttaa virheitä luokkaan A. Puuttuva metodi –virhe ilmenee puolestaan esimerkiksi, kun luokasta A lähetetään viesti luokalle B parametreillä, joita ei ole B:n määritelmässä. Rajapintavirheitä tulee milloin tahansa, kun rajapinnat kahden luokan välillä eivät ole yhteensopivia. Esimerkiksi silloin, kun viestien parametrit eivät ole oikeassa järjestyksessä tai eivät ole oikeaa tyyppiä. [LeW90]

Näitä edellä mainittuja virheitä pyritään löytämään kahdenlaisilla testeillä, rajapintatestillä ja funktionaalisella testillä. Integrointitestauksen päämääränä voidaan pitää edellä luokiteltujen virheiden etsimistä.

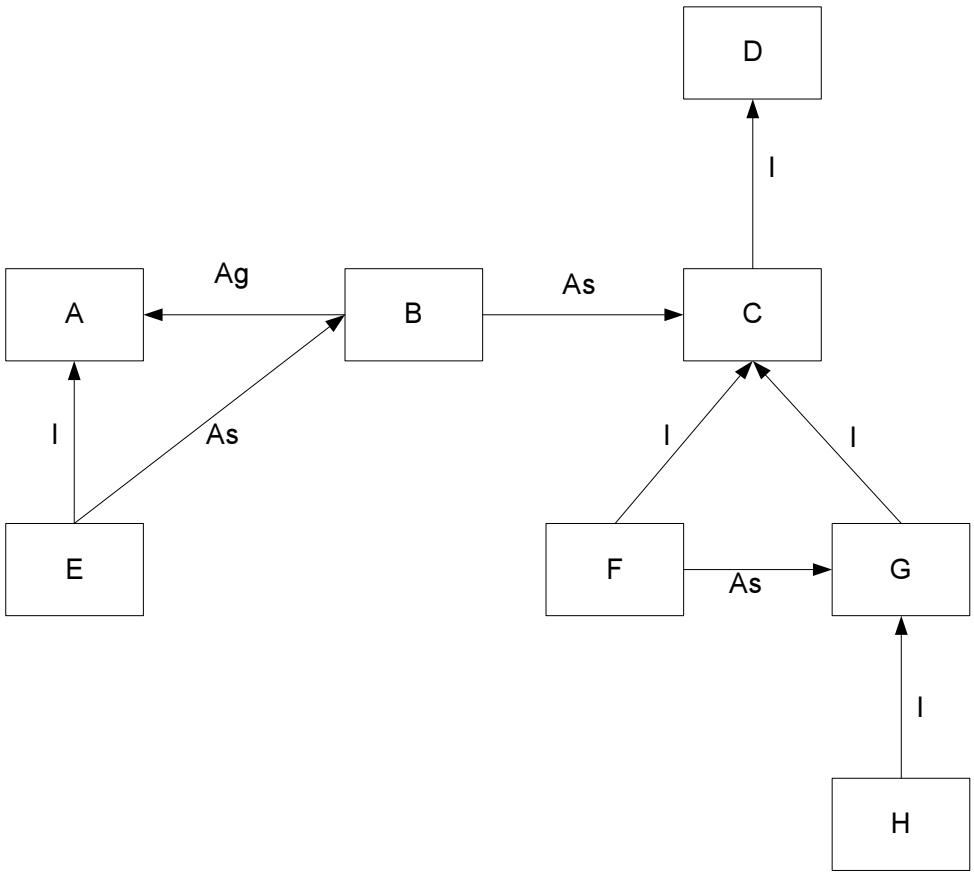
7.3 Testaustasot olio-ohjelmien integrointitestauksessa

Olio-ohjelmien luokkien integroinnissa vaikeutena on ohjelman monimutkaiset riippuvuudet. Luokat voivat olla periytymis- assosiaatio- tai kooste-suhteessa toisiinsa. Assosiaatiota ja koostetta on selitetty kappaleessa 2.2. Kun luokkien välillä on paljon riippuvuuksia, vaikeudeksi tulee valita oikea integrointijärjestys. Kokemuksesta tiedetään, että ei ole hyvä yhdistää kaikkia luokkia kerralla. Tällöin virheen paikantaminen on hankalaa. Kun taas yhdistetään luokkia toisiinsa vähän kerrallaan, ongelmaksi tulee luokkien riippuvuudet toisiin luokkiin, jotka on rajattu testauksen ulkopuolelle. Jälkimmäinen tapa on kuitenkin parempi. Jos integrointitestauksen jossain vaiheessa tulee virhe, tiedetään, että se johtuu juuri lisättyjen luokkien integroinnista. Luokkien, joita tarvittaisiin testauksessa, mutta jotka on täytynyt rajata testauksen ulkopuolelle, tilalle tehdään apuohjelmia, jotka simuloivat luokan toimintaa. Näitä tynkiä ja ajureita on selitetty kappaleessa 4.2.

Luokkien yhdistämisjärjestys täytyy suunnitella huolella. Täytyy miettiä, mistä luokasta testaus kannattaa aloittaa ja montako luokkaa lisätään kerralla. Erilaiset luokkakaaviot ja –mallit helpottavat tätä suunnittelua. Hyvänä päämääränä luokkien yhdistämisjärjestyksen suunnittelussa on yrittää minimoida tynkä-luokkien käyttöä. Tynkä-luokkien toteutus ei ole aina helppoa. Niidenkin suunnitteluun ja tekemiseen menee aikaa. Tynkä-luokkien toteutuksen kokonainen automatisointikaan ei ole mahdollista, koska se vaatii luokkien ohjelmoinnin merkityksen ymmärtämistä. Lisäksi myös tynkä-luokissa saattaa esiintyä virheitä. Mahdollisimman vähällä tynkä-luokkien käytöllä, säästetään huomattavasti testauksen kustannuksissa. [LaT00]

Olio-ohjelmille on kehitetty erilaisia olioiden riippuvuusgraafeja, joissa olioiden tai luokkien riippuvuudet on esitetty. Käyttämällä integrointitestauksessa *oliosuhdekaaviota* (ORD, Object Relation Diagram) voidaan helpottaa luokkien integroinnin suunnittelua tynkä-luokkia minimoimalla. ORD-kaaviossa luokkien väliset periytymis-, assosiaatio- ja koostesuhteet tulevat esille. Kuvassa 7.1 on esitetty erään ohjelman ORD-kaavio. Luokille X ja Y on voimassa seuraavat säännöt:

- I:llä merkitty kaari luokasta X luokkaan Y tarkoittaa, että luokka X periytyy luokasta Y.
- Ag:lla merkitty kaari luokasta X luokkaan Y tarkoittaa, että luokkien välillä on koostesuhde. Luokka X sisältää yhden tai useampia luokan Y olioita.
- As:llä merkitty kaari luokasta X luokkaan Y tarkoittaa, että luokkien välillä assosiaatiosuhde siten, että luokka X assosioituu luokan Y kanssa. [LaT00]



Kuva 7.1: ORD-kaavio

Tämän jälkeen, kun ORD-kaavio on luotu, voidaan alkaa suunnitella testausjärjestystä kaavion avulla. Tutkija Kung on kehittänyt testausjärjestyksen suunnitteluun avuksi luokan palomuuri –menetelmän (class firewall). Luokan X palomuurilla tarkoitetaan luokkien joukkoa, joita koskee luokan X muutokset ja jotka täytyy testata uudelleen, kun luokka X on muuttunut. Tätä joukkoa luokalle X merkitään CFW(X). Jotta testaus olisi riittävä CFW(X):n täytyy sisältää luokan X jälke-

läisluokat sekä luokan X kanssa assosiaatio- tai koostesuhteessa olevat luokat. Siis kaikki nämä luokat vaikuttavat luokan X testaukseen. [KuG95]

Alunperin Kung on kehittänyt luokan palomuuri menetelmän testausjärjestyksen suunnitteluun sen jälkeen, kun luokkien koodiin on tehty muutoksia. Tämä sama menetelmä sopii kuitenkin myös testausjärjestyksen suunnitteluun integrointitestauksessa, kun tarkoituksena on minimoida tynkäluokkien käyttöä. Pääideana menetelmässä on testata ensin itsenäiset luokat, joilla ei ole riippuvuuksia muihin luokkiin. Tämän jälkeen testaukseen lisätään vähitellen luokkia, joilla on riippuvuuksia ottaen huomioon kuitenkin luokkien suhteet. Esimerkiksi vanhempi luokka testataan ennen lapsi-luokkaa ja koostesuhteessa olevat luokat, joiden oliot ovat osana toista luokkaa testataan ennen kuin tämä olioita sisältävä luokka. Tällä tavoin voidaan käyttää lopullisia luokkia tynkäluokkien sijaan. [LaT00]

Luokista tehty ORD-kaavio voi olla syklinen tai ei-syklinen. Jos luokkien riippuvuudet muodostavat syklejä, ne luonnollisesti kuvataan ORD-kaaviossa sykleinä. Tällöin puhutaan syklisestä kaaviosta. Ei-syklinen kaavion on puolestaan silloin, kun luokkien riippuvuuksista tulee puumainen kaavio ilman syklejä. Kaavion ollessa ei-syklinen testausjärjestyksen luominen ilman tynkäluokkien käyttöä on helpompaa. Silloin testausjärjestys muodostuu siten, että luokka testataan ennen luokkia, jotka on esitetty tämän luokan palomuuri-listassa. Tällöin tynkä-luokkia ei tarvita ollenkaan. [LaT00]

Taulukossa 7.1 on esitetty kuvan 7.1 ORD-kaavion jokaiselle luokalle sen palomuuri-lista CFW(X). Listaan on siis koottu kaikki luokan X jälkeläisluokat sekä luokan X kanssa assosiaatio- tai koostesuhteessa olevat luokat. Tässä listassa olevat luokat on siis testattava kyseisen luokan jälkeen. Esimerkiksi luokka A on testattava ennen luokkia B ja E, koska B on koostesuhteessa luokkaan A ja E periytyy luokasta A. Luokka D on testattava ennen lähes kaikkia muita kaaviossa olevia luokkia. Vain luokka A puuttuu luokan D palomuuri-listasta. Lähes kaikki luokat riippuvat siis luokasta D. Luokkien E, F ja H testausjärjestys saadaan vapaasti valita. Muilla luokilla ei ole niihin riippuvuuksia.

Luokka X	CFW(X)
A	B, E
B	E
C	B, E, F, G, H
D	B, C, E, F, G, H
E	----
F	----
G	F, H
H	----

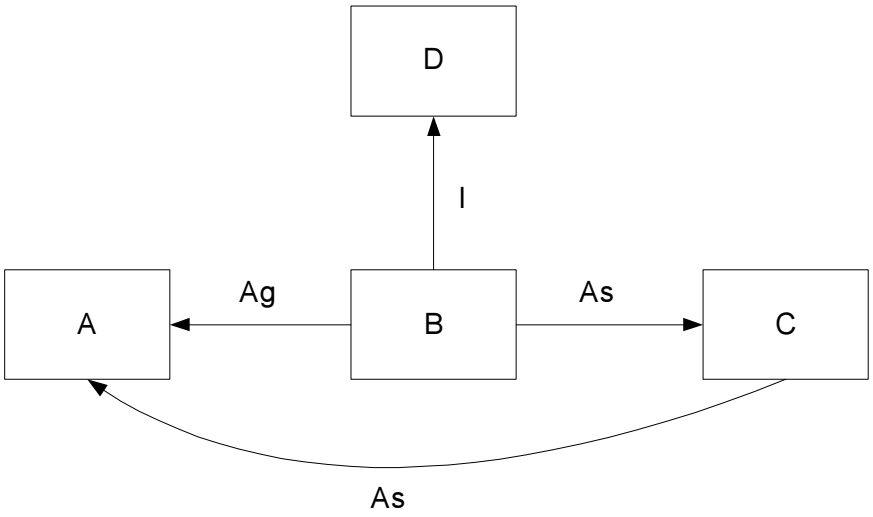
Taulukko 7.1: Luokkien palomuuri-listat

Taulukossa 7.2 on puolestaan esitetty kuvan 7.1 ORD-kaavion luokkien testausjärjestys ottaen huomioon luokkien palomuurilistat. Luokat on esitetty testaustasoittain. Ensimmäisellä testaustasolla olevat luokat testataan ensin. Tässä tapauksessa ensin testataan siis luokat A ja D. Samalla testaustasolla olevien luokkien testausjärjestyksellä ei ole väliä. Ne voidaan testata missä järjestyksessä vain. On siis aivan sama, testataanko luokka A vai luokka D ensin. Kun luokat A ja D on testattu, lisätään testaukseen toisen testaustason luokat. Esimerkissä viimeisimpänä testataan luokat E, F ja H. Näistä luokistahan ei riippunutkaan mikään muu luokka, joten on luonnollista, että nämä lisätään testaukseen viimeisenä.

Testaustaso	Luokat
1	A, D
2	C
3	B, G
4	E, F, H

Taulukko 7.2: Testausjärjestys

Jos ORD-kaavio on syklinen eli luokkien oliot ovat vaikutuksessa toisiinsa muodostaen syklejä kuten kuvassa 7.2, testausjärjestyksen suunnittelu on hankalampaa. Sitä ei voida suoraan tehdä edellä kuvatulla tavalla, vaan syklisestä luokkarakenteesta on ensin poistettava syklit. Kungin menetelmän mukaan syklit poistetaan väliaikaisesti poistamalla yksi assosiaatiosuhde. Assosiaatiosuhde on helpoin poistaa, koska se on heikoin suhde kahden luokan välillä. Menetelmä on aina mahdollinen, koska jokaisessa syklissä on ainakin yksi assosiaatiosuhde [KuG95]. Kun syklit on poistettu luokkien väliltä, voidaan testausjärjestys suunnitella ei-syklisten rakenteiden tavoin. [KuG95]

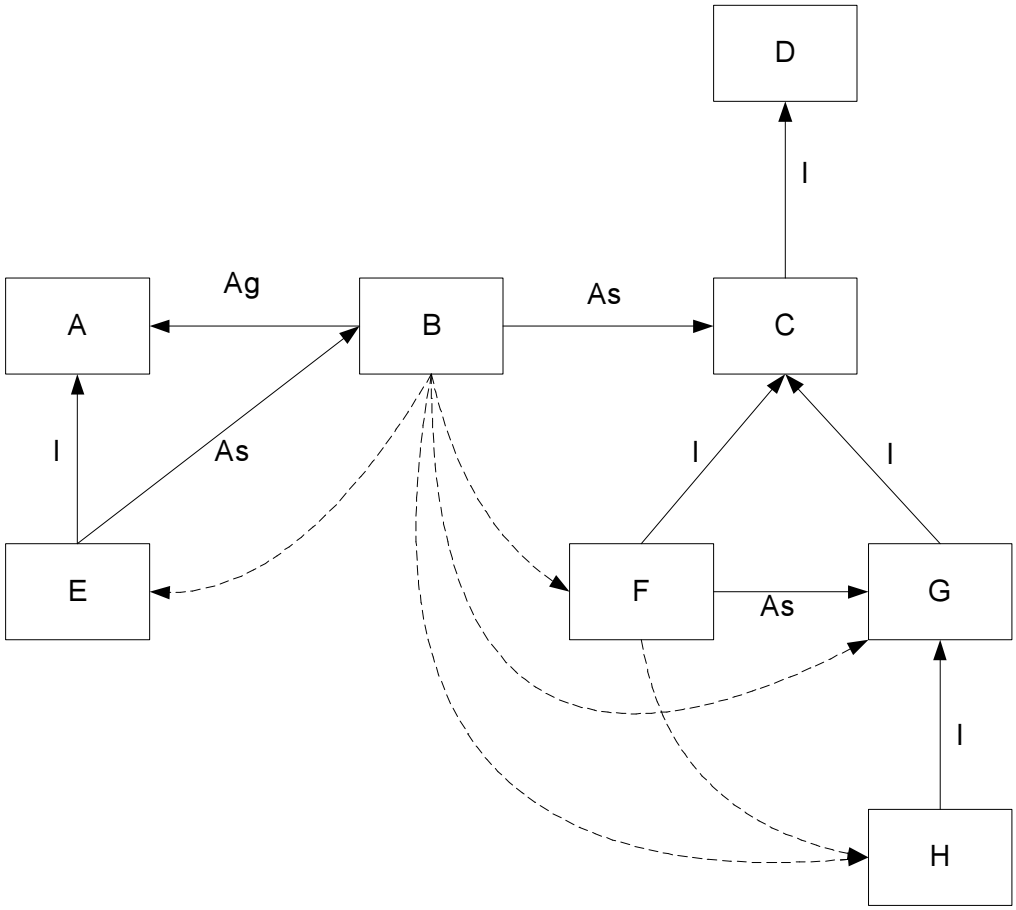


Kuva 7.2: Syklinen ORD-kaavio

Luokkien integroinnissa on kuitenkin vielä joitakin asioita, joita ei ole edellä kuvatussa menetelmässä otettu huomioon. Kaikki edellisessä esimerkissä olevat luokkien väliset suhteet olivat staattisia eli käännösaikana esiin tulevia suhteita. Olio-ohjelmissa on kuitenkin myös dynaamisia eli ajonaikana esiin tulevia suhteita luokkien välillä. Esimerkiksi kuvan 7.1 esimerkissä luokka F on assosiaatiosuhteessa luokan G kanssa. Luokka H periytyy luokasta G. Siten polymorfisten korvautuvuuksien ansiosta luokka F saattaakin ajonaikana assosioitua luokan H kanssa luokan G sijaan. Jos näin käy, luokka F pitäisi testata luokan H jälkeen. Esimerkissä luokat F ja H testataan kuitenkin samalla testaustasolla. Polymorfisista korvautuvuuksista integroinnin yhteydessä on selitetty enemmän kappaleessa 7.5.

Olio-ohjelmissa voi olla myös abstrakteja luokkia eli luokkia, joissa toteutusta ei ole loppuun asti määritelty vaan toteutus määritellään luokan alaluokissa. Abstraktit luokat aiheuttavat myös testausjärjestykseen muutoksia. Jos oletetaan, että esimerkin luokka A olisi abstraktinen, silloin sen testaaminen ei olisi kovin toteutuskelpoinen ensimmäisellä testaustasolla. Mutta huomioon otettava asia on, että luokka B vaatii nyt luokan A lapsiluokan E:n toteutusta. Tämän takia E pitäisi testata ennen B:tä. Mutta luokka E on kuitenkin suunniteltu testattavaksi luokan B jälkeen.

Kuvassa 7.3 on kuvan 7.1 ORD-kaavio piirretty uudestaan. Nyt kaaviossa on näkyvissä myös dynaamiset riippuvuudet katkoviivoilla staattisten riippuvuuksien lisäksi. Kaavioon tulee myös syklejä, kun dynaamiset riippuvuudet tulevat staattisten riippuvuuksien lisäksi. Nyt kun luokka B on assosiaatiosuhteessa luokan C kanssa, B voikin muodostaa assosiaatiosuhteen ajonaikana luokan F, G tai H:n kanssa, koska kyseiset luokat ovat luokan C jälkeläisluokkia. Integrointitestauksen järjestystä suunniteltaessa täytyy miettiä siis, mistä luokista jokainen luokka riippuu staattisesti ja mistä luokista dynaamisesti. Taulukossa 7.3 on lueteltu kaavion jokaisen luokan staattiset ja dynaamiset riippuvuudet. Luokan X staattiset riippuvuudet on merkitty S(X) ja dynaamiset riippuvuudet D(X). S(X) ilmoittaa luokat, jotka vähintään vaaditaan luokan X testaukseen ja D(X) ilmoittaa kaikki luokat, mitkä maksimissaan vaaditaan luokan X testaukseen. Luokkahan on riippuvainen kaikkiin luokkiin, joihin se on jonkinlaisessa suhteessa, mutta myös näiden luokkien yläluokkiin. [LaT00]



Kuva 7.3: ORD-kaavio, jossa otettu huomioon dynaamiset riippuvuudet

Luokka X	S(X)	D(X)
A	----	----
B	A, C, D	A, B, C, D, E, F, G, H
C	D	D
D	----	----
E	A, B, C, D	A, B, C, D, E, F, G, H
F	C, D, G	C, D, G, H
G	C, D	C, D
H	C, D, G	C, D, G

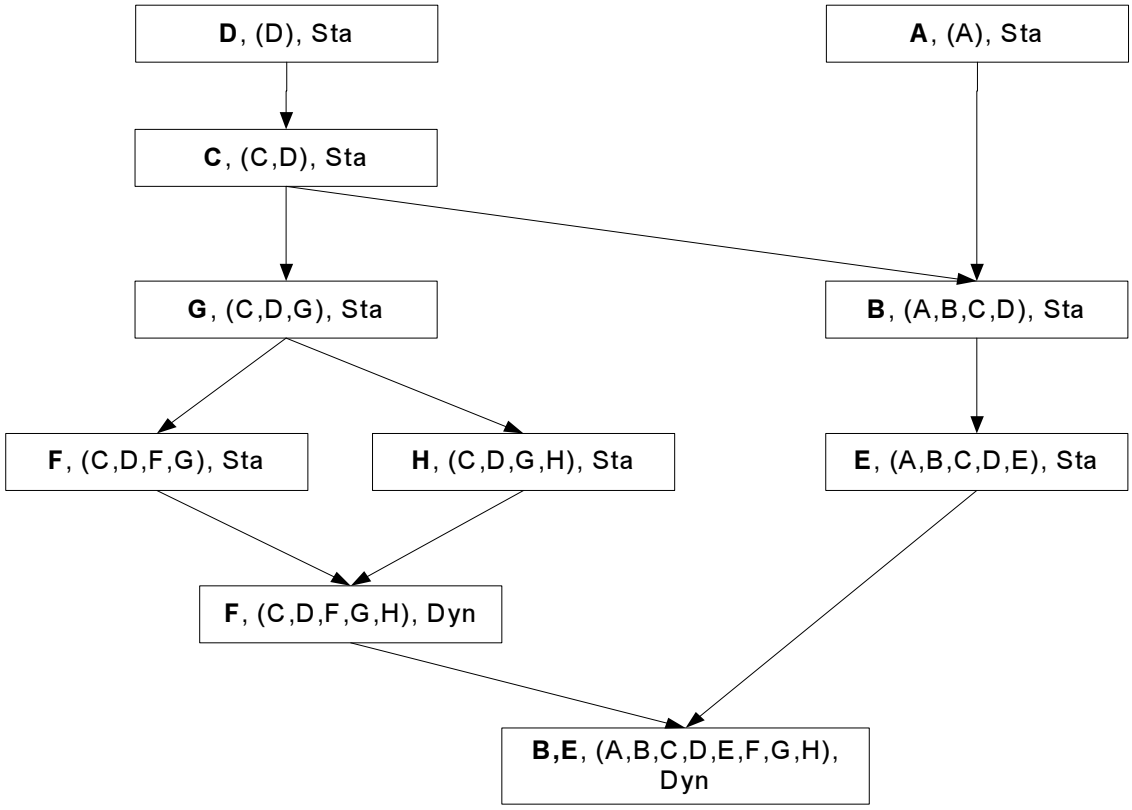
Taulukko 7.3: Luokkien staattiset ja dynaamiset riippuvuudet

Taulukossa 7.3 on esitetty kuvan 7.3 kaavion testaustasot ottaen huomioon staattisten riippuvuuksien jälkeen myös dynaamiset riippuvuudet. Kohde-sarakkeessa on luokka, jota ollaan juuri testaamassa. Tarve-sarakkeessa on lueteltu testattava luokka ja kaikki luokat, joihin se on riippuvuus suhteessa. Tyyppi-sarakkeessa on ilmoitettu riippuvuuden tyyppi, staattinen tai dynaaminen. Dynaamisia riippuvuuksia testattaessa luokat B ja E voidaan yhdistää yhdeksi kokonaisuudeksi, koska ne muodostavat syklin [LaT00].

Kohde	Tarve	Tyyppi
A	A	Sta
B	A, B, C, D	Sta
C	C, D	Sta
D	D	Sta
E	A, B, C, D, E	Sta
F	C, D, F, G	Sta
G	C, D, G	Sta
H	C, D, G, H	Sta
B, E	A, B, C, D, E, F, G, H	Dyn
F	C, D, F, G, H	Dyn

Taulukko 7.4: Testaustasot dynaamiset riippuvuudet huomioon ottaen

Testausjärjestystä mietittäessä on edelleen voimassa sääntö, että luokka on testattava kaikkien niiden luokkien jälkeen, joista sen on riippuvainen. Tämä sääntö vähentää tynkä-luokkien käyttöä. Lisäksi ensimmäisenä testausjärjestyksessä ovat staattiset riippuvuudet ja vasta näiden jälkeen dynaamiset riippuvuudet. Kuvassa 7.4 on esitetty kaavio luokkarakenteen testausjärjestyksestä integrointitestauksessa. Kaavion muodostamisessa on käytetty apuna testaustasoja. Kaaviota luetaan ylhäältä alaspäin eli ylimmät luokat testataan ensin ja niiden kanssa yhdistetään kaavion mukaan seuraavat luokat. [LaT00]



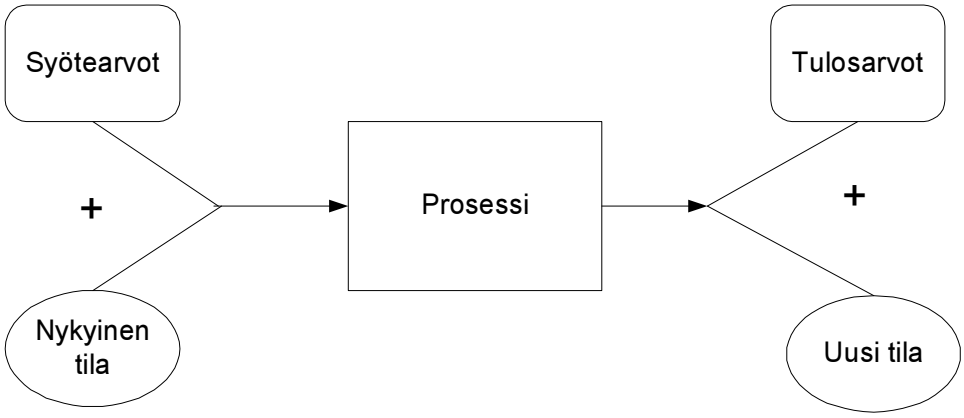
Kuva 7.4: Testausjärjestys dynaamiset riippuvuudet huomioon ottaen

7.4 Tilojen testaus integraatiotestauksessa

Kun luokkien testausjärjestys integraatiotestauksessa on saatu selville, voidaan aloittaa itse testaus. Ensin täytyy kuitenkin valita testauksen lähestymistapa. Olioiden tilojen muutokset ovat tärkeitä, joten on luonnollista tarkastella luokkien välistä integraatiota tilojen pohjalta.

Olioilla on metodit toimintoja varten ja attribuutit tiedon säilyttämistä varten. Lisäksi olioilla on aina jokin tila. Esimerkiksi pino saattaa olla tyhjä, osittain täysi tai täysi. Tilaa voidaan muuttaa olioiden metodien kautta. Olion tilan täytyy aina säilyä oikeana, jotta olion toiminta on oikea. Olion ajatellaan toimivan eritavalla eri tiloissa. Esimerkiksi, jos pino on täynnä, alkion lisäyksen ei pitäisi onnistua. Osittain täynnä olevaan pinoon lisäys taas onnistuu. Pino täytyy siis testata kaikissa mahdollisissa tiloissa.

Olioiden tilojen testaus on uutta proseduraalisten ohjelmien testaukseen verrattuna. Proseduraalisista ohjelmista testataan vain ohjelman kontrollivirta ja tietovirta. Tilojen testausta ei voida lukea kumpaankaan näistä. Proseduraalisten ohjelmien integraatiotestauksessa riittää miettiä vain syötearvot ja tulosarvot, kun taas olio-ohjelmien testauksessa täytyy ottaa mukaan myös nykyinen tila ja uusi tila. Tätä havainnollistetaan kuvassa 7.5. [KuH98]



Kuva 7.5: Tilojen analysointi mukana testauksessa.

Kun testataan olioiden tilojen muutoksia, on ensin luokiteltava, mitkä mahdolliset tilojen muutokset ovat. Turner on esittänyt neljä eri vaihtoehtoa, miten olion tila voi muuttua metodin suorittamisen jälkeen:

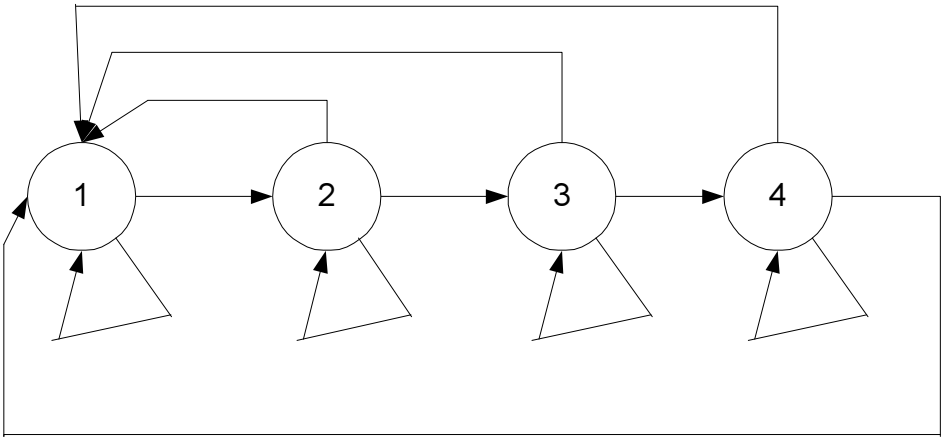
1 Olion tila voi muuttua uuteen hyväksyttyyn tilaan.
2 Olion tila voi jäädä entiselleen.
3 Olion tila voi muuttua määrittämättömään tilaan, mikä on virhe.
4 Olion tila voi muuttua määritettyyn, mutta ei hyväksyttävissä olevaan tilaan, mikä on myös virhe.

Vaihtoehdot 3 ja 4 ovat luonnollisesti virheellisiä, mutta myös vaihtoehto 2 voi olla virhe, jos tarkoituksena olisi olla vaihtoehto 1. Turner on esittänyt nämä vaihtoehdot yhden luokan tilojen testaukseen, mutta nämä sopivat myös erittäin hyvin tilojen testaukseen luokkien integraatiovaiheessa [TuR93]. Yhden luokan tilojen testausta on käsitelty kappaleessa 5. Itseasiassa tilojen testaus on lähes tärkeämpää integraatiovaiheessa kuin luokkatestausvaiheessa. Olioiden viestinvälitys toisilleen voi aiheuttaa yllättäviä vaikutuksia olioiden tiloissa. Luokkien välisessä testauksessa tilojen testauksen päämääränä on tutkia, reagoiko olio oikein viestiin toiselta oliolta kaikissa mahdollisissa tiloissa ja onko tilanmuutos oikea. Lisäksi on tärkeää tarkastella, miten viestin lähettävä olio reagoi palautusarvoihin.

7.4.1 Menetelmä tilojen testaukseen

Muutamia eri tilojen testausmenetelmiä on kehitetty. Useimmat menetelmät pohjautuvat jonkinlaiseen tilakaavioon. Tämän kaavion avulla tunnistetaan testattavat tilat ja tilasiirtymät. Kaavion avulla kehitetään testitapaukset sekä määritetään nykyinen ja uusi tila. Seuraavassa on esitelty yksi malli olio-ohjelmien tilojen testaukseen integraatiotestauksessa.

Turnerin kehittämä tilojen testausmenetelmä perustuu äärellisen tila-automaatin (Finite State Automata) ympärille. Pelkistetty tila-automaatti on esitetty kuvassa 7.6 Kuvassa tiloja havainnollistetaan ympyröillä ja tiloista toiseen johtavat nuolet ovat tilasiirtymiä. Nuolet muista tiloista takaisin ensimmäiseen tilaan kuvaavat tilan palautusta alkutilaan. Nuoli tilasta takaisin samaan tilaan on tilan arvon palautusta varten. Kaavion perusteella testataan olion tilan muutosta kappaleen alussa esiteltyjen neljän kohdan mukaisesti. [TuR93]



Kuva 7.6: Pelkistetty tila-automaatti.

Testauksessa mietitään testattavien luokkien kaikki mahdolliset toiminnot. Integraatiotestauksessa keskitytään viestinvälityksiin, jotka menevät luokan rajan ulkopuolelle toiseen luokkaan. Tässä vaiheessa luokan sisäiset viestinvälitykset pitää olla jo testattu. Kun toiminnot on tunnistettu, jokaiselle toiminnolle määritetään tilat, joissa oletetaan toiminnon olevan sallittu sekä tila, joka oletetaan olevan oliolla toiminnon jälkeen. Testauksen jälkeen verrataan näitä tiloja toiminnon alku- ja lopputiloihin, jotka saatiin testauksen tuloksena. Jos virhettä ei ole tapahtunut tuloksena saadut toiminnon alku- ja lopputilat ovat yhtenevät testauksen alussa oletettuihin alku- ja lopputiloihin. [TuR93]

Olion nykyinen tila muodostuu sen attribuuttien arvoista jollain tietyllä hetkellä. Oliolla attribuutteja voi olla hyvin monta. Olion tila saattaakin olla hyvin laaja käsite. Testausta helpotettaessa voidaan määritellä oliolle alitiloja. Alitila muodostuu olion joidenkin tiettyjen attribuuttien arvoista tietyllä hetkellä. Alitilan muodostaviin attribuutteihin valitaan luonnollisesti sellaisia, jotka eniten vaikuttavat johonkin tiettyyn toimintoon. [TuR93]

Attribuuteilla, jotka määräävät tilan voi olla myös hyvin monta erilaista arvoa. Esimerkiksi jos pinoon mahtuu sata alkiota ja attribuutin arvo kertoo alkioiden määrän, voi attribuutilla olla nollasta sataan erilaista arvoa. Tämän takia on hyödyllistä jakaa tilojen arvot yleisiin ja tarkkoihin arvoihin. Tarkka tilan arvo koostuu attribuuttien arvoista, jotka ovat oikeita tarkkoja arvoja. Näillä arvoilla testataan suoraan luokkien koodia. Pinon alkioiden määrän kertovan attribuutin tarkkoja arvoja ovat esimerkiksi 1, 12 tai 99.

Tilan yleinen arvo koostuu attribuuttien eri arvoista, jotka käyttäytyvät samalla tavalla. On hyödytöntä testata jokaista attribuutin tarkkaa arvoa erikseen, kun ollaan testaamassa olioiden tiloja integraatiotestaus vaiheessa. Käyttämällä tilojen yleisiä arvoja vähennetään testitapauksien määrää huomattavasti. Pinon tilojen yleiset arvot voisivat olla esimerkiksi tyhjä, osittain täynnä ja täynnä.

Tietenkään testauksessa ei voida unohtaa ihan kokonaan tilojen tarkkoja arvoja. Hyvä menetelmä on käyttää sekä tarkkoja että yleisiä arvoja sopivasti yhdessä. Esimerkiksi kokonaisluku-tyyppisen attribuutin *luku* arvo muuttaa erään tilan merkitystä, jos attribuutin arvo on positiivinen tai negatiivinen. Tällöin voitaisiin määritellä:

- luku < 0, yleinen arvo
- luku = 0, tarkka arvo
- luku > 0, yleinen arvo

Turnerin kehittämä testausmenetelmä etenee seuraavasti. Ensimmäiseksi suunnitellaan ja piirretään tila-automaatti niistä tiloista, joita ollaan testaamassa. Sitten määritellään jokaiselle tilalle tarvittavat alitilat, joita käytetään oikein tilojen sijaan. Alitiloille kehitetään sekä tarkkoja että yleisiä arvoja. Myös laittomat arvot täytyy muistaa ottaa mukaan. Tämän jälkeen määritetään jokaiselle toiminnolle sallitut alku- ja lopputilat. Toimintoja aletaan testaamaan sellaisesta viestistä, jota ei

kutsuta mistään muualta. Jos tämä ei ole mahdollista, joudutaan tekemään tynkä-luokkia simuloimaan testauksen ulkopuolelle jääviä osia. Jokainen toiminto testataan yksi kerrallaan. [TuR93]

Testauksessa suoritetaan siis jokainen viestinlähetys, joka on mahdollista testattavien luokkien välillä. Testataan pystytäänkö lähettämään viesti siitä olion tilasta, joka ollaan oletettu. Lisäksi tarkistetaan aiheuttaako viesti tarvittavan tilanmuutoksen siinä oliossa, mihin viesti on lähetetty. Testauksen rajaamiseksi tarkemmaksi apuna käytetään tilojen alitiloja ja tarkkoja ja yleisiä arvoja. Lisäksi tietenkin käytetään tila-automaattia, joka auttaa havainnollistamaan eri tilasiirtymät.

Alla on esitetty luokat *Tili* ja *Asiakas*.

```
Class Tili {
  double korko;
  double saldo;

  boolean pano(double määrä) {//…}
  boolean nosto(double määrä {//…}
  void saldoKysely(){//…}
}
Class Asiakas {
  String nimi;
  String osoite;
  Tili numero;

  void maksaLasku(double raha) {
    //…
    numero.nosto(raha);
    //…
  }
  void talletaRahaa(double raha) {
    //…
    numero.pano(raha);
    //…
  }
  void tarkistaTili() {
    //…
    numero.saldoKysely();
    //…
  }
}
```

Asiakas-luokka kutsuu siis *Tili*-luokan metodeita. Näillä kahdella luokalla on yhteensä viisi attribuuttia, jotka muodostavat luokkien tilan. Jos ollaan testaamassa tilin saldon vaihteluita, on hyödytöntä ottaa kaikkien attribuuttien muodostamaa tilaa mukaan. Muodostetaan alitila, johon kuuluvat vain *Tili*-luokasta *saldo*-attribuutti ja *Asiakas*-luokasta *numero*-attribuutti. *Numero*-attribuuttihan kertoo, mikä asiakkaan tili on kyseessä. Lisäksi muodostetaan tälle alitilalle yleisiä ja tarkkoja arvoja, joita testataan. Yleiset arvot voisivat olla *Tili*-luokalle tilinsaldo negatiivinen ja tilin saldo positiivinen. *Asiakas*-luokalle puolestaan käteistilit ja talletustilit. Näistä saadaan alitilan yleiset arvot eli ne olisivat:

- tilin saldo negatiivinen käteistileillä ja talletustileillä
- tilin saldo negatiivinen käteistileillä ja positiivinen talletustileillä
- tilin saldo positiivinen käteistileillä ja negatiivinen talletustileillä
- tilin saldo positiivinen käteistileillä ja talletustileillä.

Luokkia testataan muodostamalla testitapauksia niin, että alitilan arvo muuttuu tilasta toiseen. Lähetetään siis metodien viestejä siinä järjestyksessä, että saadaan tilan arvo muuttumaan halutusta tilasta haluttuun tilaan ja tutkitaan vaihtuiko tila odotusten mukaisesti. Alitilan tarkat arvot ovat tarkkoja saldon arvoja ja tilien tarkkoja numeroita. Myös näitä täytyy tietenkin testata, vaikka tilatestauksessa yleiset arvot ovatkin tärkeämpiä.

7.5 Muita menetelmiä integraatiotestaukseen

Tilojen testaus on yksi tapa suorittaa integraatiotestausta. Vaikka tilojen testaus on ehkä tärkein menetelmä, on muitakin lähestymistapoja integraatiotestaukseen. Voidaan käyttää myös formaaleja menetelmiä. Yksi uusista menetelmistä on tapahtuma-pohjainen tekniikka. Metodien viestipolkuihin liittyvä taktiikka on esitettävissä myös käytännönläheisenä. Menetelmä onkin varsin hyödyllinen.

7.5.1 Tapahtuma-pohjainen tekniikka

Tämä menetelmä on vielä tutkinnan alla. Menetelmä pohjautuu samanaikaisiin tapahtuma pareihin. Testaus lähtee siis ohjelman tapahtumien pohjalta ei tilojen. Menetelmässä otetaan enemmän huomioon viestien lähetysjärjestykseen ja ohjelman logiikkaan liittyviä asioita kuin olion tiloja.

Esimerkiksi kahden eri tapahtuman välinen suhde voidaan määritellä olevan aina voimassa, mahdollisesti voimassa tai ei koskaan voimassa oleva. Tämä kuvaa tilannetta, jossa ensimmäisen tapahtuman pitää, saattaa pitää tai ei pidä seurata toista tapahtumaa. Määritetään siis kahden ohjelman tapahtumien välille rajoitteita sen mukaan, miten ajatellaan ohjelman toimivan. Testauksessa testa-

taan, pitävätkö nämä määritetyt rajoitteet paikkansa ja suoritetaanko ohjelman tapahtumat oikeassa järjestyksessä. [ChC02]

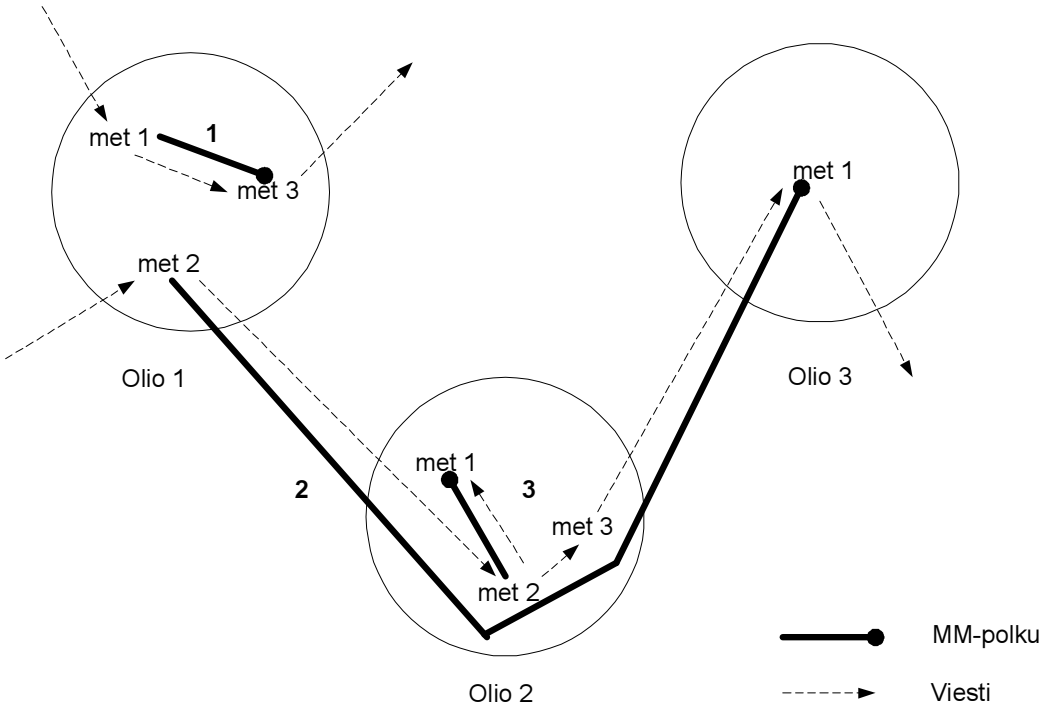
Tapahtumien järjestystä määrääviä rajoitteita voidaan myös muodostaa toisista rajoitteista. Esimerkiksi jos oletetaan, että ”tapahtumaa P seuraa tapahtuma Q”, on aina voimassa olevaa –tyyppiä. Puolestaan ei koskaan voimassa olevaa –tyyppiä olisi, että ”tapahtumaa Q seuraa tapahtuma R”. Tällöin voidaan päätellä, että ”tapahtumaa P seuraa tapahtuma R” on ei koskaan voimassa olevaa –tyyppiä.

Tapahtumaparien rajoituksia voi olla toisinaan hankala määritellä. Monesta rajoituksesta tulee helposti mahdollisesti voimassa olevaa –tyyppiä. Tämä hankaloittaa testauksen tuloksen analysoimista. On vaikea arvioida, onko tapahtumien järjestys todella sallittu tietyssä tilanteessa, jos määritelmä on pelkästään mahdollisesti voimassa oleva. Myös piirteiden uudelleenmäärittely vaikeuttaa tapahtuma-pohjaista testausta. On tilanteita, jossa tapahtumien järjestys voi olla mahdollisesti voimassa oleva yläluokan olioille, mutta aina voimassa oleva alaluokan olioille. [ChC02]

7.5.2 MM-polku -menetelmä

Kolmas menetelmä integraatiotestaukseen on formaalit menetelmät. Nämä ovat monimutkaisuutensa takia harvemmin käytössä kuin esimerkiksi olioiden tiloihin pohjautuva testaus. Näistä menetelmistä kuitenkin ehkä metodien viestipolkuihin perustuva menetelmä on eniten käytössä, koska se on helposti käsitettävissä myös ei-formaalissa muodossa.

Menetelmässä muodostetaan testauspolkuja metodien suoritusjärjestyksen mukaan. Näitä polkuja kutsutaan *MM-poluiksi* (Method/Message Path). MM-polku alkaa metodista, joka lähettää viestin jollekin toiselle metodille ja loppuu metodiin, joka ei enää lähetä viestiä muille olioille. Tämä on siis kohta, jossa viestiketju katkeaa. Kun MM-polut muodostuvat metodi-viesti –pareista olioiden verkostossa, ne haarautuvat ja menevät limittäin muiden MM-polkujen kanssa. Kuvassa 7.7 on kuvattu kolme eri oliota, jotka sisältävät kolme eri MM-polkua. Polku 1 on olion 1 sisällä, polku 2 kulkee oliosta 1 olioon 3 olion 2 kautta ja polku 3 on olion 2 sisällä. Polkujen muodostamisen jälkeen ne testataan suorittamalla polussa olevien metodien viestin lähetykset polun määräämässä järjestyksessä. Polut testataan yksi kerrallaan. [JoE94]

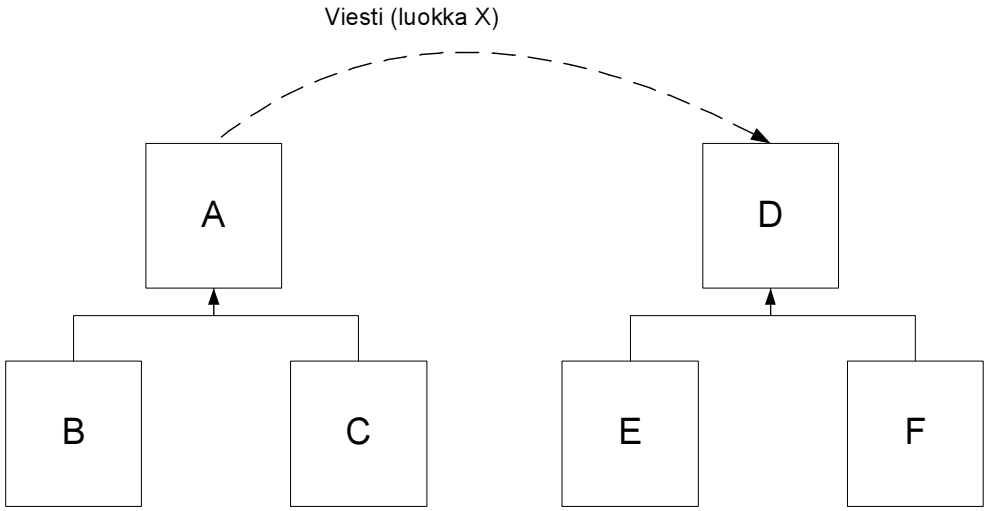


Kuva 7.7: Kolme MM-polkua ja viestinvälitykset.

Menetelmä on hyvin laaja, koska se käsittää kerralla kaikki testauksessa olevat oliot. Yksi MM-polku voi mennä kaikkien olioiden läpi. Toisaalta MM-polku saattaa olla vain kahden metodin välinen. Jos MM-polut on muodostettu oikein, menetelmä on myös hyvin kattava. Se käy läpi kaikki metodien viestit. Tähän menetelmään on helposti yhdistettävissä tilojen testaus, jolloin testauksesta saadaan hyvin kattava ja monipuolinen integraatiotestaus.

7.6 Polymorfismin testaus

Kahden luokan välinen vuorovaikutus tapahtuu, kun olio lähettää viestin toisen luokan oliolle ja saa aikaan viestin vastaanottavassa luokassa jonkin tapahtuman. Polymorfismin testauksessa täytyy ensin tarkastella, onko yhdistettävillä luokilla alaluokkia. Tämän testauksen kannalta helppo tapaus on, jos luokilla ei ole alaluokkia. Tällöin ei polymorfisia sidontoja voi olla metodien välillä. Jos luokilla on alaluokkia riippuen lähetettävän viestin tyypistä, metodien välille voi syntyä polymorfisia sidontoja.



Kuva 7.8: Kaksi yksinkertaista luokkahierarkiaa.

Kuvassa 7.8 on kaksi yksinkertaista luokkahierarkiaa. Luokan A olio lähettää viestin luokan B oliolle. Päämääränä näiden kahden luokan yhdistämisen testauksessa on tarkastaa viestin toiminnallisuus ja luokan D rajapinta. Kun testataan, että luokan rajapinta on yhteensopiva toisen luokan kanssa, täytyy tarkistaa viestissä olleiden parametrien määrä, parametrien järjestys, parametrien tyypit ja mahdolliset arvot. Näiden perustestausten jälkeen täytyy tietenkin myös testata viestin lähettämisen vaikutuksia olioihin. On tarkastettava olioiden attribuuttien arvot ja tilat. Tilojen testausta hankaloittaa vielä jokaisen attribuutin ja metodin oma näkyvyysmääre, mikä täytyy ottaa huomioon.

Kahden luokan integraatiotestaus on siis hyvin lähellä proseduraalisten ohjelmien integraatiotestausta. Olio-ohjelmien luontaiset piirteet tuovat kuitenkin tähänkin testaukseen omat hankaluutensa ja niitä täytyykin tarkastella enemmän. Olioiden tilojen lisäksi polymorfisten korvautuvuuksien tarkastelu on kahden luokan integraatiotestauksessa tärkeää.

Periytyvillä luokilla voi olla polymorfisia korvautuvuuksia. Polymorfiaa on käsitelty tarkemmin kappaleessa 2.3. Ohjelman ajonaikana jokin metodi voi siis korvautua luokan jonkin jälkeläisluokan metodilla. Jos mietitään uudelleen kuvaa 7.8, huomataan, että luokkien välillä voi olla polymorfisia korvautuvuuksia. Luokat B ja C periytyvät luokasta A ja voivat siis korvata A:n metodin ajonaikana. Tämä tarkoittaa käytännössä, että joko B:n olio tai C:n olio voi A:n olion sijaan lähettää viestin D:n oliolle. Mutta tietenkin myös luokan D polymorfiset korvautuvuudet täytyy ottaa huomioon. Luokat E ja F voivat yhtä hyvin kuin D luokkakin ottaa vastaan lähetetyn viestin. Testattaessa luokkien A ja D viestinvälitystä täytyy ottaa huomioon sekä A:n että D:n mahdolliset polymorfiset korvautuvuudet ja kaikki niiden yhdistelmät. Jos tätä ei otettaisi huomioon, testauksessa olisi vain kahden luokan, A ja D luokan, välinen viestinvälitys. Ottamalla polymorfian huomioon testattavien tapausten määrä nousee yhdestä yhdeksään. [McM94]

Kahden luokan väliltä voi kuitenkin löytyä vielä lisääkin polymorfisia korvautuvuuksia. Välitettävän viestin parametrien tyypit täytyy tutkia. Jos parametrit ovat olioita, myös näillä voi olla polymorfisia korvautuvuuksia. Testitapausten määrä kasvaa jälleen nopeasti, jos viestin parametreina on olioita ja niillä monia alaluokkia, joiden välistä polymorfiaa täytyy tutkia. Esimerkiksi jos kuvan 7.8 luokkien A ja D olioiden välisellä viestillä on kaksi parametria, jotka kummatkin ovat olioita. Näillä parametreilla on vielä kummallakin kolme alaluokkaa. Testattaessa huolella tällaisen viestin välitystä täytyy lähettää 16 erilaista viestiä. Tämä luku kattaa kahden olioparametrin juuriluokan ja kolmen alaluokan mahdolliset polymorfiset korvautuvuudet ja kaikki niiden yhdistelmät. Tähän ei ole kuitenkaan vielä laskettu luokkien A ja D polymorfisia korvautuvuuksia. Ottamalla tämä huomioon testitapausten määrä nousee 144 kappaleeseen. Tässä on oletettuna, että testauksessa riittää yksi testitapaus yhtä yhdistelmää kohti. Suorittamalla nämä 144 testiä on kuitenkin vasta testattu yksi viestinvälitys kahden luokan välillä. Täytyy muistaa, että kaikilla luokkahierarkioiden olioilla on viestejä välitettävänä, jotka pitäisi myös testata. [McM94]

7.6.1 Polyformismi-testauksen riittävyys

Ottaen huomioon polymorfiset korvautuvuudet viestin lähettävältä ja vastaanottavalta oliolta sekä parametreina olevilta olioilta erilaisissa tiloissa, testauksesta tulee todella haastavaa. Testitapausten määrä nousee nopeasti hyvin suureksi. Huolellinen testaus vaatii paljon työvoimaa ja aikaa, mikä saattaa tuottaa monesti ongelmia. Tämän vuoksi testauksen kattavuudesta täytyy joskus tinkiä. Testauksen riittävyyteen on ainakin kolme eri tasoa luokkien välisessä testauksessa. Nämä ovat laaja (exhaustive), suppea (minimal) ja edustava (representative). [McM94]

Laajassa testauksessa täytyy testata kaikki mahdolliset vuorovaikutukset. Toisin sanoen otetaan huomioon kaikki mahdolliset yhdistelmät viestin lähettävän olion, viestin saavan olion, parametreinä olevien olioiden ja polymorfisten korvautuvuuksien ja olioiden eri tilojen välillä. Laaja testaus vaatii paljon työtä, mutta on todella kattava. [McM94]

Suppeassa testauksessa vähennetään testitapauksia laajaan testaukseen verrattuna. Siinä otetaan huomioon vain viestin lähettävä olio, vastaanottava olio ja parametreinä olevat oliot. Kaikki oliot testataan yhdessä tietyssä tilassa, jossa olioiden arvioidaan yleensä olevan viestinlähetys hetkellä. Polymorfisia korvautuvuuksia ei tässä tapauksessa huomioida. Tästä syystä testaus ei ole täysin kattava. [McM94]

Edustava testaus on kattavuudeltaan kahden edellisen mallin välimuoto. Tässä testauksessa otetaan myös huomioon viestin lähettävä ja saava olio sekä parametreinä olevat oliot kaikissa tiloissa. Lisäksi polymorfisia korvautuvuuksia käsitellään. Erona laajaan testaukseen on, että kaikkia mah-

dollisia tilojen ja polymorfisten korvautuvuuksien yhdistelmiä ei käsitellä. Jokainen olio ja niiden tilat tulee testattua, mutta ei kaikkia kombinaatioita tai niihin liittyviä tiloja. Edustava testausmalli on kattavuudeltaan ja tehokkuudeltaan erittäin hyvä malli luokkien väliseen testaukseen. [McM94]

8 Yhteenveto

Olio-ohjelmien testaus on haastava tehtävä. Olio-ohjelmoinnin erot proseduraaliseen ohjelmointiin verrattuna ovat niin suuret, että samojen testausmenetelmien käyttö ei ole mahdollista. Joitakin testausmenetelmiä voidaan kuitenkin hyödyntää uudestaan hieman muuttamalla niitä. Esimerkiksi metodien testauksessa voidaan käyttää lähes samoja menetelmiä kuin proseduraalisten aliohjelmien testauksessa. Muutoin samojen testausmenetelmien hyödyntäminen on aika vähäistä.

Olio-ohjelmien rakentuminen luokista ja olioista aiheuttaa sen, että pelkkien metodien testaaminen yksistään ei ole kovin järkevää. Metodit liittyvät kiinteästi johonkin luokkaan, joten metodien testauksen jälkeen on luonnollista testata luokat. Luokissa metodien lisäksi on myös attribuutteja, joiden testauksesta ei ole vielä paljon tutkimustuloksia. Tämä johtuu ehkä siitä, että attribuuttien testaus on aika yksinkertaista varsinkin, kun ne tulevat testattua metodien testauksen yhteydessäkin. Luokan testaukseen sen sijaan on useampia menetelmiä. Menetelmät jakaantuvat luokan modaalisuuden mukaan. Modaalisuus kertoo luokan olioiden tilojen ja viestijärjestyksen rajoitukset. Tämän jaottelun ansioista saadaan jokaiselle erityyppiselle luokalla oma ja tehokas testimenetelmä. Luokkien testauksessa täytyy kuitenkin aina muistaa, että pelkästään luokkia ei voi testata vaan ne testataan aina luokkien ilmentymien eli olioiden kautta.

Yksittäisen luokkatestauksen jälkeen seuraavaksi testausvuorossa ovat luokkahierarkiat. Olio-ohjelmien periytyminen asettaa luokat luokkahierarkiaan. Siinä yläluokilta periytetään piirteitä alaluokille. Hierarkian testauksessa voidaan hyödyntää yläluokkien testitapauksia, kun ollaan testaamassa alaluokkia. Tärkeää on kuitenkin tutkia, onko alaluokan piirre peritty suoraan yläluokalta, yläluokan uudelleenmääritelty piirre vai aivan uusi piirre, jota ei esiinny yläluokassa. Aivan uuden piirteen tapauksessa ei ole luonnollisesti mitään testitapauksia, mitä voitaisiin käyttää hyväksi yläluokalta. Muissa tapauksissa soveltuvin osin saadaan valmiita testituloksia suoraan yläluokalta. Luonnollisesti hierarkian juuriluokka, joka ei periydy mistään, on testattava kokonaan luokkatestauksen tavoin.

Viimeisenä vaiheena olio-ohjelmien testauksessa on testata luokkien integrointi. Siinä tärkeää on valita oikea testausjärjestys. Paras tapa yhdistellä luokkia on sellainen, jossa luokkia simuloivia tynkäluokkia tarvittaisiin mahdollisimman vähän. Olio-ohjelmoinnissa olioilla on paljon riippu-

vuuksia muihin olioihin ja tämän vuoksi oikean testausjärjestyksen löytäminen voi olla hankalaa. Integraatiotestauksessa kannattaakin käyttää apuna oliosuhdekaaviota, mikä helpottaa olioiden suhteiden hahmottamista. Luokkien yhdistämisessä myös olioiden tilojen tarkkailu on tärkeää. Olioiden tilojen pitäisi säilyä vahingoittumattomana.

Polymorfismi tuo myös hankaluuksia testaukseen. Varsinkin luokkahierarkioiden integroinnissa täytyy olla tarkkana, mikä metodi lähettää millekin metodille viestin. Polymorfinen metodihan voi korvautua ajonaikana luokan jonkin jälkeläisluokan metodilla. Testauksessa tämä otetaan huomioon lisäämällä testausta ja yrittämällä testata mahdollisimman monia eri vaihtoehtoja viestin lähettäjästä ja vastaanottajasta.

Testausta ei siis voida pitää yksinkertaisena ohjelmistotuotannon vaiheena. Testaukseen pitää varata aikaa ja siihen täytyy paneutua huolella. Ennen kaikkea olio-ohjelmien testaus tuo työhön lisää uusia ja mielenkiintoisia piirteitä. Olio-ohjelmoinnin luontaiset piirteet eivät tee testausta helpoksi, mutta koko ajan kehitetään uusia ja parempia testausmenetelmiä.

LÄHTEET:

[BaS94] Barbey Stéphane, Strohmeier Alfred: The Problematics of Testing Object-Oriented Software. Software Quality Management II Building Quality into Software, UK, 1994, p. 411-426.

[Bin99] Binder, Robert V.: Testing Object-Oriented Systems. Addison-Wesley, 1999.

[Bin99b] Binder, Robert V.: "Testing Objects: Myth and Reality". *Object Magazine*, v 5, n 2, May 1995, p.73-75.

[Bir92] Birss, Bob: Testing object-oriented software. SunProgrammer – The Newsletter for Professional Software Engineers, 1(3):15-16, 1992.

[ChC02] Chan W.K., Chen T.Y., Tse T.H.: An Overview of Integration Testing Techniques for Object-Oriented Programs. Proceedings of the 2nd ACIS Annual International Conference on Computer and Information Science, 2002.

[FoS00] Fowler Martin, Scott Kendall: UML Distilled Second Edition. Addison-Wesley, 2000.

[FrF01] Fröhlich Peter H., Franz Michael: On Certain Basic Properties of Component-oriented Programming Languages. Information and Computer Science University of California, Irvine, 2001.

[HaM92] Harrold Mary Jean, McGregor John D., Fitzpatrick Kevin J.: Incremental Testing of Object-Oriented Class Structures. Proc, 14th Int'l Software Eng., 1992, p. 68-80.

[JoE94] Jorgensen Paul C., Erickson Carl: Object-Oriented Integration Testing. Communications of the ACM, Vol. 37, No 9, 1994, p. 30-38.

[Kos98] Koskimies, Kai: Pieni oliokirja. Suomen Atk-kustannus Oy, Jyväskylä, 1998.

[KuG95] Kung D., Gao J., Hsia P.: Class Firewall, Test order, and Regression testing of Object-Oriented Programs. Journal of Object-Oriented Programming, vol. 8, 1995, p. 51-65.

[KuH98] Kung David C., Hsia Pei, Gao Jerry: Testing Object-Oriented Software. IEEE Computer Society, California, 1998.

[Lab97] Labiche, Yvan: On Testing Object-Oriented Programs. France, 1997.

[LaT00] Labiche Y., Thévenod-Fosse P., Wayeselynck H., Durand M.-H.: Testing Levels for Object-Oriented Software. ICSE, 2000, p.136-145.

[LeW90] Leung Hareton, White Lee: A Study of Integration Testing and Software Regression at the Integration Level. IEEE Transactions on Software Engineering, 1990, p. 290-300

[McS01] McGregor John D, Sykes David A: A Practical Guide to Testing Object-Oriented Software. Addison-Wesley, 2001.

[McM94] McDaniel Robert, McGregor John D.: Testing the Polymorphic Interactions between Classes. Department of Computer Science Technical Report, Clemson University, 1994.

[MeK98] Menger Gisela, Keedy James Leslie, Evered Mark, Schmolitzky Axel: Collection Types and Implementations in Object-Oriented Software Libraries. University of Ulm, Technology of Object-Oriented Languages, 1998, p. 97 –109.

[Paa00] Paakki Jukka: Ohjelmistojen testaus, Helsingin yliopisto, 2000.

[SoF98] Soundarajan Neelam, Fridella Stephen: Reasoning About Polymorphic Behavior. Computer and Information Science The Ohio State University, Technology of Object-Oriented Languages, 1998, p. 346 –358.

[TuR93] Turner C. D., Robson D. J.: The State-Based Testing of Object-Oriented Programs, Proc. IEEE Conf. on Software Maintenance, Los Almitos, 1993, p. 302-310.

[WiH92] Wilde Norman, Huitt Ross: Maintenance Support for Object Oriented Programs. IEEE Transactions on Software Engineering, vol. 18, 1992, p.1038-1044.