Mikko Aitta-aho

Code Contracts ja ComTest-yksikkötestausgenerointi .NET-kielissä

Tietotekniikan
kandidaatintutkielma
4. toukokuuta 2015

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Mikko Aitta-aho
Yhteystiedot: mikko.p.aitta-aho@jyu.fi
Työn nimi: Code Contracts ja ComTest-yksikkötestausgenerointi .NET-kielissä
Title in English: Code Contracts and ComTest Unit Test Generation in .NET languages
Työ: Tietotekniikan kandidaatintutkielma
Sivumäärä: 21
Tiivistelmä: Opetuksen tehostamiseen suunnattu työkalu ComTest osaa luoda yksikkötestejä koodin kommentteihin kirjoitettujen ohjeiden perusteella. Sopimuspohjaisessa suunnittelussa olion metodeille asetetaan ehtoja, joiden on oltava voimassa ennen operaation suorittamista tai sen jälkeen. Tällaiset ehdot voidaan automaattisesti kirjoittaa osaksi koodin kommentteja. Code Contracts on laajennos .NET-kieliin, jonka avulla sopimuspohjainen suunnittelu saadaan osaksi sovelluskehitystä. Tutkimuksessa selvitetään, miten ComTest ja Code Contracts liittyvät toisiinsa.
Abstract: ComTest, a tool mainly directed to make teaching more efficient, is able to create Unit Tests based on directions written in the code comments. In Coding by Contracts, conditions are set to the classes methods that have to be valid before executing the operation or after it. This kind of conditions can be appended to the code comments by the compiler software. Code Contracts is an add-on to .NET-languages, which brings Coding by Contracts into the software development. This research investigates how ComTest and Code Contracts relate to each other.
Avainsanat: Sopimuspohjainen suunnittelu, Code Contracts, ComTest, Yksikkötestaus, .NET Framework, .NET Ohjelmistokehys, C#, VB.NET, F#
Keywords: Code by Contracts, Code Contracts, ComTest, Unit Test, .NET Framework, C#, VB.NET, F#

Copyright © 2015 Mikko Aitta-aho

All rights reserved.

Sisältö

1 Johdanto

Tietokoneohjelma voidaan toteuttaa monella eri tavalla. Ajatellaan tässä siis yhtä tehtävää, jonka suorittamiseksi toteutetaan ohjelmia[1]. Eri käyttötarpeisiin tehdyt ohjelmat suorittavat eri käskyjä, käyttävät eroavasti resursseja ja suoriutuvat tehtävästä eri nopeuksin. Kuitenkin ohjelmien päättyessä niiden lopputulema, eli tuloste, on sama mikäli ne toimivat oikein ja niiden syöte on sama. Käyttötarpeet ja ohjelmien suoritusympäristöt rajaavat jotkin ohjelmien toteutusvaihtoehdot pois. Tällöin oleellisinta ohjelman suorituksessa on sen tulosteen oikeellisuus.

Viime vuosina paljon huomiota saanut ohjelmien kehitykseen käytettävä metodi *testivetoinen ohjelmistokehitys* pyrkii parantamaan ohjelmien oikeellisuutta. Siinä ajatellaan, että ennen ohjelman toteuttamista kannattaa luoda testitapauksia, ns. *yksikkötestejä*, jotka varmistavat ohjelman suorituksen oikeellisuuden. Aivan kaikkia mahdollisia käyttötapauksia ei kannata testata sillä niiden lukumäärä on hyvin suuri. Sen sijaan kannattaa testata erikoistapaukset. Esimerkiksi jakolaskun toteuttavaa ohjelmaa kehittäessä kannattaa testata nollalla jakaminen, sillä sen ei pitäisi olla mahdollista.

ComTest on Jyväskylän yliopistossa kehitetty apuväline testauslähtöisen kehittämisen opettamisen helpottamiseksi. Sen avulla ohjelmoija voi ohjelmaa kirjoittaessaan lisätä yksikkötestit suoraan sen yhteyteen koodin kommentteihin. ComTest lukee kommentteja käännösaikana ja kirjoittaa niiden perusteella testiluokkia. Yksikkötestien kirjoittaminen välittömään yhteyteen ohjelman kanssa auttaa ohjelmoijaa ymmärtämään yksikkötestien ja ohjelman yhteyttä. Perinteinen tapa yksikkötestien toteutukseen on kirjoittaa täysin erillisiä luokkia, joissa yksikkötestit ovat omia ohjelmiaan täysin irrallaan varsinaisesta ohjelmasta.

Ohjelma voi päätyä oikeelliseen lopputulokseen vain jos sille annettu syöte täyttää ehdot, jotka ohjelman toteutuksen aikana sille on annettu. Näitä ehtoja ei eksplisiittisesti useasti edes määrätä, vaan ne ovat seurausta ohjelmien rajoituksista. Esimerkiksi ohjelmalle voidaan määrittää esiehdoksi kokonaisluku, jolloin syöte on yleensä rajoitettu ohjelman suoritusympäristön sille asettamien rajoitusten takia esimerkiksi olemaan esitettävissä kahdeksalla binäärillä. On kuitenkin kehitetty keinoja, joissa olio-ohjelmointiparadigmaa toteuttavissa kielissä voidaan määritellä monimutkaisiakin ehtoja syötteelle. Tällaista ohjelmointia sanotaan *sopimuspohjaiseksi*

[1] Termistöstä. Olio-ohjelmointiin tottunut lukija voi ajatella luokan metodeja tai funktioita ohjelmina. Ainakin arkikielessä ohjelma mielletään yleensä kokonaisuudeksi, joka on oikeammin *tietokoneohjelmisto*.

suunnitteluksi ja (eräs) sen .NET-kielinen toteutus on Code Contracts -laajennos. Sen avulla ohjelman kehittäjä määrittelee ohjelmalle ne ehdot, joita sen syötteen täytyy toteuttaa, joita sen tuloksen täytyy täyttää, ja joita luokat ohjelmien suorituksien jälkeen noudattavat.

Code Contracts -laajennoksella saadaan luotua automaattisesti koodin kommenttia, joka kuvaa ohjelman käyttäytymistä esi- ja jälkiehtojen määrittäminä. ComTest-työkalu käyttää omaa makrokieltään testitapausten kehittämiseksi. Sovelluskehitysympäristöt eivät osaa ilman laajennoksia tehdä mielekkäitä yksikkötestejä esi- ja jälkiehtojen määrittelyjen pohjalta, vaikka edellytykset tähän ovat olemassa. Yksikkötestien laatiminen sopimusten pohjalta vähentäisi bugien syntymistä toteutukseen sekä helpottaisi (ulkopuolisen henkilön) koodin seuraamista ja edelleenkehitystä.

Tässä kirjallisuustutkimuksessa verrataan Code Contracts -laajennosta ja ComTest-työkalua, sekä kartoitetaan tarkemmin niiden yhteiskäyttöä. ComTest-työkalulla on varmasti omat käyttötarpeensa ja hyötynsä ohjelmoinnin opetuksen puolella. Niihin ei kuitenkaan oteta kantaa tässä tutkimuksessa, pedagogiikan ollessa tutkijalle vieras ala. Tutkimuksen keskittyminen ohjelmistokehityksen näkökulmiin on varmastikin riittävä tuottamaan tarpeellista ja mielenkiintoista tutkimustulosta.

Tutkielman luvussa 2 esitellään tutkimusongelman kannalta keskeiset käsitteet ja luvussa 3 esitellään testitapauksia esimerkkien avulla. Luvussa 4 tutkitaan ComTestin ja Code Contractsin suhdetta. Lopuksi luvussa 5 kootaan yhteenveto tutkimuksesta.

2 Keskeiset käsitteet

Tässä luvussa esitellään tutkimusongelman keskeiset käsitteet. Aluksi käsitellään Hoaren logiikka ja kuinka sen avulla voidaan tehdä päättelyjä ohjelman oikeintoimivuudesta. Sen jälkeen esitellään testivetoisen ohjelmistokehityksen perusidea ja siihen liittyvä automaattinen testaus sekä yksikkötestit.

Testivetoisen ohjelmistokehityksen esittelyä seuraa aliluku koodien kommentoinneista, sillä sitä seuraavan aliluvun asia *ComTest* hyödyntää niitä. Sitten esitellään sopimuspohjaisen suunnittelun ajatus ja viimeisessä aliluvussa käydään läpi Code Contracts -laajennos .NET kieliin.

2.1 Hoaren logiikka

Ohjelman toiminta voidaan ajatella koostuvan kolmikosta *esiehto* (P), *ohjelma* (Q) ja, ohjelman suorituksen loppuessa, sen tuottama *tulos* (R). Hoare painottaa, ettei logiikalla voida todistaa ohjelman loppumisesta [Hoare(1969)]. Ohjelma voi siis jäädä suorittamaan päättymätöntä silmukkaa, jolloin *tulos* jää toteutumatta.

Kolmikkoa merkitään

$$P\ \{Q\}\ R,$$

joka tulkitaan "Esiehdon P ollessa tosi ennen ohjelman Q suorituksen aloittamista, väite R on tosi sen päättyessä." Esiehto voi puuttua, jolloin merkitään **tosi** $\{Q\}\ R$.

Hoaren logiikassa ohjelman voidaan ajatella olevan yksi alirutiini tai kokonainen ohjelmisto. Kun yksittäiset alirutiinit ovat todistettu Hoaren logiikalla toimiviksi, voidaan niiden todistuksia käyttää hyväksi todistettaessa niitä kutsuvaa ohjelmistoa [Hoare(1969)].

Erityisesti tämän tutkimuksen kannalta on oleellista huomata, että ohjelma voidaan toteuttaa usealla eri tavalla. Eli on olemassa useita toisistaan poikkeavia ohjelman toteutuksia, mutta niiden antamat lopputulokset ovat samoilla esiehdoilla keskenään samat. Tälläiset ohjelmat voidaan vaihtaa keskenään ja silti niitä kutsuvan ohjelman toiminnallisuus pysyy samana [Hoare(1969)]. Otetaan esimerkiksi sovellus *Qmain*, jossa eräässä alirutiinissa pitää löytää annetussa verkossa lyhin reitti haluttujen solmujen välille. Sovelluksessa ongelman ratkaisee alirutiini *Q1*. Myöhemmin on kuitenkin kehitetty alirutiini *Q2*, joka ratkaisee saman ongelman nopeammin kuin *Q1*. Sovelluksen suorituskykyä halutaan parantaa, joten alirutiini *Q1* korvataan alirutiinilla *Q2*. Koska molemmat alirutiinit ovat todistettu tuottamaan saman lopputuloksen (kun ne täyttävät samat esiehdot), voidaan korvaus tehdä ja sovellus *Qmain* tuottaa saman lopputuloksen kuin ennen alirutiinien vaihtoa. Nyt sovellus vain suorittaa lyhimmän reitin laskemisen aikaisempaa nopeammin.

2.2 Testivetoinen ohjelmistokehitys

Ohjelmat, ja niitä yhdistämällä ohjelmistot, voidaan siis toteuttaa usealla eri tavalla, kuten Hoaren logiikka osoittaa. Ohjelmat, jotka tuottavat samalla syötteellä saman lopputuloksen ovat keskenään vaihdettavissa [Hoare(1969)]. Silloinhan eri toteutuksien oikeintoimivuus voidaan testata ja varmistaa. Annetaan saman tehtävän eri toteutuksille samat syötteet ja tarkastetaan lopputuloksen oikeellisuus. Testivetoinen ohjelmistokehitys kehoittaa sovelluskehittäjiä miettimään, kuinka toteutuksen

toimivuus voidaan testata. Ohjelmiin tehdään useita muutoksia ohjelmiston koko elinkaaren aikana. Testit takaavat, ettei muutokset riko jotain aikaisemmin tehtyä ratkaisua muutettuun ohjelmaan tai sitä hyödyntävään ohjelmaan tai siitä riippuvaiseen ohjelmaan [Beck(2005)]. Testivetoisen ohjelmistokehityksen testejä kutsutaan *yksikkötesteiksi*.

Automaattinen testaus tarkoittaa, että testin voi suorittaa ilman käyttäjän syötteitä. Testitapaus on tallessa ja sen voi suorittaa halutessaan. Yksikkötestit tulisi tehdä nopeasti suoritettaviksi, jotta niiden suorittamiseen ei kulu liikaa aikaa. Testitapauksia mietittäessä pitäisi koko ohjelmiston ratkaisema ongelma pilkkoa pienempiin ongelmiin, jotka ovat testattavissa. Kokonaisuuden toiminnallisuuden todentaminen perustuu siihen, että pienet ongelmat muodostavat yhdessä koko ohjelmiston ongelman. Kaikki toiminnallisuus tulee siis testattua, mutta kokonaisuutta pienemmissä osissa [Beck(2005)].

Yksikkötestit luodaan kirjoittamalla uusia ohjelmia, joissa testataan varsinaista ohjelmaa. Testiohjelmien erottaminen varsinaisesta ohjelmasta noudattaa ohjelmoinnissa tunnettua periaatetta *vastuiden erottamista*. Siinä on kyse ohjelman tarkoituksen rajaamisesta. Ei pyritä yhdellä ohjelmalla ratkaisemaan useita tehtäviä, vaan keskitytään ratkaisemaan vain yksi. Testaamalla toiminnan varmistaminen on kokonaan eri tehtävä kuin se tehtävä, jonka ratkaisuun ohjelma on kehitetty. Tästä syystä ohjelman ei tulisi olla sen itsensä testaaja. Testien suorittaminen kuuluu ohjelman kehitysvaiheeseen eikä sen käyttövaiheeseen.

Testien erottamisessa varsinaisista ohjelmista on myös se hyöty ettei testiohjelmat tule mukaan käännettyyn ohjelmistoon. Jos testiohjelmat olisivat mukana varsinaisessa ohjelmistossa, ne paisuttaisivat käännettyjen ohjelmien kokoa.

2.3 Koodin kommentit

Ohjelmointikielissä on yleensä mahdollisuus kommentoida ohjelmakoodia. Koodien kommentit ovat tekstiä, jotka eivät vaikuta varsinaisen tehtävän suorittamiseen. Niitä käytetään selkeyttämään koodia ja opastamaan koodia lukevia ihmisiä. Esimerkiksi koodissa saatetaan kommentoida miksi jokin tietty aliohjelma on valittu toisen, melkein saman asian suorittavan, sijaan. Tällaisia kommentteja kirjoitetaan tavallisesti koodiin ohjelman varsinaisten käskyjen sekaan ja ne näkyvät vain koodia tarkasteltaessa.

Olio-ohjelmointikielissä, kuten .NET-kielissä, on mahdollista kirjoittaa myös *julkisia kommentteja*. Tällaisia kommentteja kirjoitetaan dokumentoimaan ohjelmien ja

aliohjelmien käyttötarkoitusta, käyttötapauksia, rajoituksia, sekä muita huomion arvoisia asioita. Koodien dokumenteista hyötyy ensisijaisesti muut kehittäjät, jotka käyttävät ohjelmia tai aliohjelmia. Julkiset kommentit kirjoitetaan yleensä luokkien tai ohjelmien esittelyjen välittömään yhteyteen. Esimerkki julkisista ja ei-julkisista kommenteista listauksessa 1. Huomion arvoista on, kuinka ohjelman toimintaa kuvaavat kommentit ovat välittömässä yhteydessä luokan alirutiineihin ja attribuutteihin.

Listaus 1: Esimerkki koodin kommenteista

```
/// <summary> [Julkinen kommentti luokalle:]
/// Jono luokka toteuttaa yksinkertaisen
/// ensimmäisenä-sisään-ensimmäisenä-ulos (FIFO) -tyyppisen
/// kokoelman.
/// </summary>
public class Jono {

    public List<object> _kokoelma; // sisäinen kokoelma

    /// <summary> [Julkinen kommentti aliohjelmalle:]
    /// Alustaa tyhjän Jono-luokan, joka varaa muistia
    /// määrätylle määrälle alkioita.
    /// </summary>
    /// <param name="lkm">
    /// Jonolle varattavien alkioiden määrä.
    /// </param>
    public Jono( int lkm ) {

        //[Ei-julkinen kommentti aliohjelman suorituksesta:]
        // Aloitetaan tyhjällä jonolla
        _kokoelma = new List<object>( lkm );
        ...
    }
    ...
}
```

.NET-kielissä koodien kommentit luodaan XML-tekstitiedostoksi, jota muut ohjelmistot osaavat tulkita ja jatkokäsitellä. Sekä ComTest että Code Contracts hyö-

dyntävät tätä ominaisuutta, tosin molemmat toisistaan poikkeavilla tavoilla. Niistä tarkemmin seuraavissa aliluvuissa. Yleensä koodin julkinen dokumentointi muodostetaan tämän XML-tekstitiedoston perusteella ja sitä varten on olemassa useita valmiita ohjelmistoja.

2.4 ComTest

Jyväskylän yliopistossa aloitetussa projektissa pyritään parantamaan testivetoisen ohjelmistokehityksen opettamista johdantotason ohjelmointikursseilla. Se on havaittu tutkimuksissa haasteelliseksi opiskelijoihin kohdistuvan teknisen ja kognitiivisen kuorman lisääntymisen vuoksi [Lappalainen ym.(2010)].

ComTest on työkalu, joka luo yksikkötestit aliohjelmien koodin julkisiin kommentteihin kirjoitettujen ohjeiden mukaan. Tällä tavalla kirjoitettuna testin tarkoitus on selitetty lähempänä opetuksessa käytettävää esimerkkikoodia kuin perinteisessä tavassa kirjoittaa testit kokonaan erillisiin luokkiin. Lappalaisen, ym. mukaan testien irroittaminen varsinaisesta suoritettavasta koodista vaikeutti testivetoisen ohjelmistokehityksen opettamista. Tutkimukset ovat kuitenkin osoittaneet testivetoisen ohjelmistokehityksen varhaisen opettamisen tärkeyden [Lappalainen ym.(2010)].

Ohjelmoijan täytyy kirjoittaa ComTestin käyttöön tarkoitetut yksikkötestien ohjeistukset ComTest-makrokielellä. ComTest-makrokieli muistuttaa syntaksiltaan Java-kieltä. Sitä käyttämällä kirjoitetaan yksikkötestien ohjeistukset koodin julkisiin kommentteihin **<example>**-elementtiin **<pre name="test">**-alielementin sisään. Sieltä ne välittyvät käännettyyn XML-kommenttitiedostoon normaalisti muiden koodin julkisten kommenttien tavoin. Etuna tällaisessa toteutuksessa on myös se, että yksikkötestien ohjeistukset tulevat varmasti mukaan myös koodin julkiseen dokumentointiin.

ComTest työkalu on tuettu Java-kielessä sekä .NET-kielissä ja se on toteutettu sovelluskehitysympäristöjen laajennoksina. ComTest-laajennokset toimivat ainakin suosituissa sovelluskehitysympäristöissä Eclipse ja Visual Studio [Lappalainen ym.(2010)].

2.5 Sopimuspohjainen suunnittelu

Sopimuspohjainen suunnittelu on Bertrand Meyerin ideoima olio-ohjelmoinnin menetelmä, jossa abstraktien tietotyyppien ideaa laajennetaan sopimuskäsitteellä [Meyer(1992)]. Menetelmässä sopimukset mukailevat reaalimaailmaa. Niissä on kaksi osapuolta: asiakas (metodin kutsuja) ja toimittaja (kutsuttu metodi). Lisäksi sopimusta määri-

teltäessä sovitaan kummankin osapuolen velvoitteet, jotka määritellään ohjelmointikielen loogisin lausein.

Velvoitteet määritetään olioiden julkisissa metodeissa esiehdoin ja jälkiehdoin. Esiehto asettaa velvoitteen asiakkaalle ja se on ehto, joka on oltava voimassa kun metodia kutsutaan. Esiehto voi olla esimerkiksi ”parametri *A* ei saa olla tyhjä osoitin”, jolloin asiakkaalle on ilmoitettu ettei ohjelma pysty antamaan oikeellista tulosta, jos sen parametri *A* on tyhjä osoitin. Etuna esimerkin esiehdossa on se, ettei ohjelman toteutuksessa tarvitse käsitellä erikoistapausta, jossa parametri *A* on annettu tyhjänä osoittimena. Jälkiehto asettaa velvoitteen toimittajalle ja se on ehto jonka on oltava voimassa kun metodi on suoritettu. Jälkiehto voi olla esimerkiksi ”tulos on epätyhjä merkkijono”, jolloin toimittaja lupaa että tuloksen merkkijono ei ole tyhjä. Etuna esimerkin jälkiehdossa on se, ettei asiakkaan tarvitse huolehtia tuloksena saatavan merkkijonon tyhjyyden tarkistamisesta.

Luokkainvarianteilla voidaan määritellä ehtoja, jotka ovat voimassa koko luokan eliniän aikana. Tarkemmin sanottuna luokkainvarianttien määräämien ehtojen on oltava voimassa välittömästi luokan luonnin jälkeen, ennen jokaista julkista metodia kutsuttaessa, sekä jokaisen julkisen metodin kutsun jälkeen. Luokkainvariantit luonnehtivat luokkaa kokonaisuutena, toisin kuin esi- ja jälkiehdot, jotka kuvaavat vain luokan yksittäisiä rutiineja [Meyer(1992)]. Esimerkkinä luokkainvariantista toimii Pino-luokan attribuutti *Koko*, joka ilmoittaa pinossa olevien alkioiden määrän. Määrätään Pino-luokan *Koko* attribuutin arvon olevan aina yhtäsuuri tai suurempi kuin nolla: ”Pino.Koko >= 0”.

On syytä huomata, että luokkainvariantin asettamaa ehtoa voidaan rikkoa metodien suorituksen ollessa kesken. Voi olla hyödyllistä rikkoa sääntöä algoritmin suorituksen ajaksi. Luokkainvariantit voidaan tulkita olevan sekä esi- että jälkiehto jokaiselle luokan julkiselle metodille. Otetaan esimerkiksi edellisen kappaleen Pino-esimerkkiluokka. Sen attribuutti *Koko* on määritetty ”Pino.Koko >= 0”. Luokassa voisi olla metodi *JärjestäPinonElementit*, jonka toteutuksessa Pino-luokan attribuutille *Koko* asetetaan aluksi arvo -1. Tämä ei vielä riko luokkainvarianttia. Kun aliohjelman *JärjestäPinonElementit* suoritus loppuu, täytyy luokkainvariantin ehdon kuitenkin olla **Tosi** eli *Koko*-arvon täytyy olla yhtäsuuri tai suurempi kuin nolla tai luokkainvarianttia on rikottu.

2.6 Code Contracts -laajennos

Microsoft on kehittänyt Code Contracts -laajennoksen, jolla sopimuspohjaisen suunnittelun vaatimat esi- ja jälkiehdot, sekä invariantit voidaan ottaa käyttöön .NET-kielillä toteutetuissa ohjelmissa. Laajennos on kieliriippumaton, joten sitä voidaan käyttää kaikissa .NET-kielissä eli C#:ssa, VB.NETissä sekä F#:ssa [Code Contracts User Manual].

Laajennokseen liittyvien työkalujen avulla voidaan luoda automaattisesti koodin kommentteihin kuvaukset sopimusehdoista. Code Contracts -laajennoksessa on mukana työkalu, jolla ehdot liitetään mukaan koodin julkisista kommenteista muodostettavaan XML-tekstitiedostoon. Työkalu lisää esiehdot elementtiin **<requires>**, jälkiehdot elementtiin **<ensures>** ja luokkainvariantit elementtiin **<invariant>**. Elementtiin kirjoitetaan sen sisältöön varsinainen ehto. Lisäksi jos ehdon määrityksessä on annettu vapaamuotoinen kuvaus ehdosta, se lisätään elementin **description**-lisämääreen arvoksi. Esimerkiksi kommentissa oleva kuvaus

```
<requires description="Parametri 'merkkijono' ei saa olla tyhjä
osoitin!">merkkijono != null</requires>
```

ilmoittaa elementin sisällössä sopimuksen ehdon "merkkijono != null" eli ettei ohjelmaa kutsuttaessa sille välitettävän parametrin *merkkijono* arvo saa olla tyhjä osoitin.

Koodin julkisiin kommentteihin tehdyt lisäykset ovat pääasiassa tarkoitettu koodin ymmärtämistä helpottamaan. Kommentit kirjoitetaan koodikielellä, joten on mahdollista saada ComTest käyttämään hyväksi Code Contractsin luomaa kommentointia. Tämä vaatii tietysti ComTestin tulkkaaman kielen laajentamista, sillä Code Contractsin kommenttien elementtejä se ei tällä hetkellä tulkitse ollenkaan.

3 Testitapaukset

Luvussa esitellään muutama esimerkkiohjelma ja pohditaan niihin sopivia testitapauksia. Esimerkkiohjelmien avulla tutkitaan niihin sopivien esi- ja jälkiehtojen sekä luokkainvarianttien käyttöä.

Ensimmäisessä aliluvussa pohditaan luokkainvariantteja ja niiden yksikkötestaamista. Luokkainvarianttien käsittelyn jälkeen omassa aliluvussaan tarkastellaan esiehtoja ja niiden yksikkötestaamista. Viimeisessä aliluvussa tarkastellaan jälkiehtoja ja niiden yksikkötestaamista.

3.1 Luokkainvariantin testiin sopivien lähtöarvojen valitseminen

Tehdään esimerkin vuoksi luokka aikavälien käytön helpottamiseksi. Luodaan siis luokka *Aikavali*, jolla on kaksi julkista DateTime-tyyppistä ominaisuutta: ”Alkuaika” ja ”Loppuaika”. Kirjoitetaan luokkainvariantti `Alkuaika < Loppuaika`, joka takaa että luokan julkisten metodien kutsujen jälkeen pätee aina ehto ”alkuaika on ennen loppuaikaa”. Nyt Code Contracts kirjoittaa koodikommentteihin rivin ”**<invariant>** Alkuaika < Loppuaika **</invariant>**”. ComTest-työkalua voitaisiin muokata niin, että se osaa luoda tästä testitapauksen. Millaisilla arvoilla metodia sitten pitäisi kutsua? Loogisesti vaihtoehtoja ovat ”alkuaika < loppuaika”, ”alkuaika = loppuaika” ja ”alkuaika > loppuaika”, joista vain ensimmäinen vaihtoehto on sallittu sovelluksen ehtojen mukaan.

Tarkastellaanpa, miten sopimuslähtöisesti esiehtoa käyttämällä luodaan esimerkin luokka. Listauksessa 2 kuvataan osa luokasta, joka on tarkoitettu helpottamaan aikavälien käyttöä esimerkin mukaisesti.

Listaus 2: Esimerkkiluokka

```
public class Aikavali {
    public DateTime Alkuaika;
    public DateTime Loppuaika;
    public Aikavali( DateTime alku, DateTime loppu ) {
        this.Alkuaika = alku;
        this.Loppuaika = loppu;
    }

    // luokkainvariantti:
    [ContractInvariantMethod]
    private void OnkoValidi() {
        Contract.Invariant( Alkuaika < Loppuaika,
            "Alkuajan on oltava ennen loppuaikaa!" );
    }
}
```

ComTest-työkalun pitäisi osata tehdä ComTest-makrokielellä koodin kommentteihin listauksen 3 kuvaama testitapaus.

Listaus 3: Testiluokka esimerkkiluokalle

```
<pre name="test">
    DateTime pienempi = new DateTime( 2000, 1, 1 );
    DateTime suurempi = new DateTime( 2012, 1, 1 );
    Aikavali testattava = new Aikavali( suurempi, pienempi );
            #THROWS ( Exception ex ) {
                ex.Message.Contains(
                "Alkuajan␣on␣oltava␣ennen␣loppuaikaa!" )
                    === true;
                }
</pre>
```

Ohjelmoija pystyy valitsemaan listauksessa 3 käytetyt päivämäärät intuitiivisesti, koska testiin kelpaa mitkä tahansa päivämäärät joiden järjestys on oikein virheen aiheutumisen kannalta. ComTest-työkalun on kuitenkin mahdotonta tehdä samanlainen ratkaisu.

3.2 Esiehtojen testitapausesimerkki

Code Contracts -laajennoksessa kirjoitetaan esiehdot aina metodien alkuun. Esiehto on Boolen lauseke, joka määrittelee millaisen ehdon kutsujan on täytettävä jotta kutsu on oikeellinen. Esiehtoja voidaan määrittää yhteen ohjelmaan useita, joista kaikkien on oltava kutsuttaessa voimassa. Code Contracts -laajennoksessa käytetään seuraavaa syntaksia: ”Contract.Requires(boolean_ehtolause, virheviesti)”. Esimerkissä *virheviesti* on esiehtoa *boolean_ehtolause* rikottaessa tilannetta kuvaava merkkijono, joka välitetään sitä rikkovaan ohjelman suorituksen kohtaan. Esimerkiksi ohjelman käännössovellus voi ilmoittaa *virheviestin* ohjelmoijalle tai suorituksen aikana se voidaan näyttää ohjelmiston käyttäjälle.

Edellisen aliluvun tapaan tästä voitaisiin luoda ComTestillä testitapaus, jossa testin ideana on varmistaa, että esiehtoa rikottaessa ohjelma todellakin heittää virheen. Esimerkiksi pitäisi siis ComTestin pystyä luomaan listauksen 4 kuvaama yksikkötesti.

Listaus 4: Esimerkkitestiohjelma esiehdoista

```
<pre name="test">
Luokka testattava = new Luokka;
tapaus_jolla_ehtoa_rikotaan;
```

```
testattava.TestattavaMetodi; #THROWS ( Exception ex ) {
    ex.Message.Contains( virheviesti ) === true;
}
</pre>
```

Koska ehdon rikkomiseen on olemassa useita eri mahdollisuuksia, on jälleen mahdotonta tehdä automaattisesti testitapaus. Testitapauksia, joita kannattaa testata, on käytännössä useita. Hoaren kolmikossa oletuksena on nimenomaan, että ohjelma toimii, kun esiehdot ovat kunnossa [Hoare(1969)]. Sopimuspohjaisessa suunnittelussa esiehdon varmistamiseksi on siis tavallaan turha tehdä testitapausta, koska ohjelman voidaan ajatella toimivan aina oikein *vain* esiehtojen ollessa kunnossa.

3.3 Jälkiehtojen testitapausesimerkki

Jälkiehdot rajaavat ohjelman mahdollisia tuloksia [Meyer(1992)]. Ne siis rajaavat järkeviä yksikkötestejä. Jos jälkiehto rajaa ohjelman tuloksesta pois negatiiviset luvut, yksikkötesteistä voidaan jättää pois testit joissa tulokseksi tulisi jokin negatiivinen luku. Tarkemmin; yksikkötesteissä ei voi testata tapauksia, joissa jälkiehto rikkoutuisi. Varsinkin, jos käytetään käännöksen aikaista tarkastusta, ei ohjelmakoodissa edes voi olla jälkiehtoa rikkovaa koodia (edes yksikkötesteissä). Esimerkiksi listauksen 5 kuvaama ohjelma palauttaa sille annettujen kokonaislukujen summan. Jostain syystä esimerkkiohjelman esiehdoissa ei rajoiteta parametrien kokonaislukuja millään tavalla, mutta sen jälkiehdoissa rajataan negatiiviset kokonaisluvut pois mahdollisesta tulosjoukosta.

Listauksen 5 ohjelma on hankala esimerkki. Ohjelmasta puuttuvia esiehtoja on hankala perustella. Esi- ja jälkiehdoilla on suhde, jonka ohjelma määrittää [Hoare(1969)]. Tässä esimerkissä tiedetään, että yhteenlaskun tulos voi olla negatiivinen luku, mikäli yhteenlaskettavia lukuja ei millään tavalla rajata. Oletetaan tässä kuitenkin esimerkin vuoksi, että tällaiselle ohjelmalle on jokin tarve ja esimerkkiohjelman olevan siten perusteltu tällaisenaan.

Listaus 5: Esimerkkiohjelma jälkiehdoista

```
public static int LaskeSumma( int a, int b ) {
    // Jälkiehdolla varmistetaan, ettei ohjelma koskaan palauta
    // negatiivista kokonaislukua.
    Contract.Ensures( Contract.Result<int>() >= 0 );
```

```
    // Esimerkin vuoksi toteutus on näin yksinkertainen ja
    // esiehdot ovat jätetty toteuttamatta.
    return a + b;
}
```

Jälkiehtojen yksikkötestauksessa on sama tilanne kuin esiehtojen ja luokkainvarianttien yksikkötestauksissa. Testitapauksen täytyy tarkastaa oikeanlaisen poikkeuksen tapahtuminen sopimusrikkeestä. Automaattisesti listauksen 5 jälkiehdosta *tulos >= 0* on erittäin hankala kirjoittaa yksikkötesti joka rikkoisi sopimuksen. Tällaisen yksikkötestin kirjoittaminen vaatii ohjelman toteutuksen tutkimista, jotta voidaan tietää millä parametrien arvoilla ohjelmaa kutsuttaessa jälkiehtoa rikotaan.

4 ComTestin ja Code Contractsin suhde

Tässä luvussa analysoidaan ComTest-työkalun ja Code Contracts -laajennoksen suhdetta. Ensimmäisessä aliluvussa tarkastellaan yksikkötestauksen ja sopimuspohjaisen suunnittelun suhdetta. Toisessa aliluvussa pohditaan koodin julkisten kommenttien tulkkaamista ja niiden avulla muodostettuja testitapauksia. Viimeisessä aliluvussa perehdytään yksikkötesteihin joilla testataan ohjelman käyttäytymistä sopimuksia rikottaessa.

4.1 Yksikkötestaus ja sopimuspohjainen suunnittelu

Sopimuspohjaista suunnittelua voidaan pitää vaihtoehtona yksikkötestien kirjoittamiselle. Sitä voidaan pitää myös laajennoksena yksikkötesteille. Beck mainitsee asian ohimennen lähdeluettelossaan muttei perustele kumpaakaan väitettään [Beck, Andres(20 ss. 171]. Tarkastellaan yksikkötestien ja sopimuspohjaisen suunnittelun suhdetta hieman tarkemmin.

Ohjelmisto koostuu useasta pienestä ohjelmasta, jotka ovat keskenään vuorovaikutuksessa. Yksikkötesti testaa tällaisen pienen ohjelman toimintaa. Tässä mielessä yksikkötestit määrittävät ohjelman toiminnalle sopimuksia. Erityisesti testivetoisessa ohjelmistokehityksessä tämä on vahvasti esillä. Siinä pyritään kirjoittamaan testiohjelma ennen varsinaista ohjelmaa [Beck(2005)]. Tällöin testitapaukset luovat valmiiksi sopimuksia varsinaiselle ohjelmalle, jota ollaan vasta kehittämässä.

Testivetoisessa ohjelmistokehityksessä yksikkötestien on tarkoitus testata ohjelman toimivan, kuten ohjelmistosuunnittelija on sen tarkoittanut [Beck(2005)]. Ku-

ten Meyer kirjoittaa, sopimuspohjaisessa suunnittelussa sen sijaan määritellään esiehdoin millaiset vaatimukset kutsuvan ohjelman täytyy toteuttaa. Jälkiehdoilla määritellään, millaiset vaatimukset ohjelman tulee toteuttaa suorituksen päättyessä. Luokkainvariantit määrittelevät, millaiset luokan tilat ovat mahdollisia ohjelman suoritusta kutsuttaessa ja millaisessa tilassa se tulee olemaan ohjelman suorituksen jälkeen.

Vaikka yksikkötestejä voisi luonnehtia sopimuksiksi, on mielestäni sopimuspohjaisen suunnittelun kuvaamat sopimukset täysin eri asia. Yksikkötestien sopimukset liittyvät Hoaren kolmikon ohjelman Q oikeelliseen suoritukseen. Sopimuspohjaisessa suunnittelussa taas määritellään Hoaren kolmikon esiehdot P ja jälkiehdot R. Luokkainvariantit voidaan ajatella määrittävän sekä esi- että jälkiehdon jokaiselle luokan alirutiinin kutsumiselle [Meyer(1992)]. Esi- ja jälkiehdot liittyvät ohjelmien väliseen vuorovaikutukseen, kun taas yksikkötestit liittyvät oleellisesti vain ohjelman sisäiseen toimintaan.

Luokkainvariantit kuitenkin rajaavat mahdollisia luokan tiloja. Tässä mielessä yksikkötestien ja luokkainvarianttien käyttö on jossain määrin päällekkäistä. Silti ne myös tukevat toisiaan. Luokkainvariantti luonteensa vuoksi sulkee pois testitapauksia. Esimerkiksi rajoitus `Alkuaika < Loppuaika` listauksessa 2 rajaa pois kaikki yksikkötestit, joissa luokan attribuutin *Alkuaika* arvo voisi olla yhtäsuuri tai suurempi kuin attribuutin *Loppuaika* arvo. Luokkainvariantilla on esimerkin tapauksessa määritetty oleellinen toiminnallisuus luokalle, jota pelkästään ohjelmointikielellä ei voida määrittää. Esimerkin luokan yksikkötesteissä alirutiineille testataan niiden oikeellinen toiminta. Esimerkiksi luokassa voisi olla alirutiini *LisääKuukausiLoppuaikaan()*, jonka yksikkötesteissä varmistetaan vain *Loppuaika* attribuutin arvon muuttuminen oikealla tavalla.

4.2 Kommenttien käsittely

Code Contracts -laajennosta käytettäessä esi- ja jälkiehdoista sekä luokkainvarianteista muodostetaan koodin käännöksen aikana lisäykset koodin julkisiin kommentteihin [Code Contracts User Manual]. ComTest tulkkaa koodin julkisiin kommentteihin kirjoitettua ComTest-makrokieltä [Lappalainen ym.(2010)]. Tutkimuksessa havaittiin ettei koodin julkisten kommenttien muokkaaminen molempien laajennosten toimesta aiheuttanut käännöksenaikaisia virheitä eivätkä laajennokset muokanneet toistensa kommentteja.

Jos ComTestin halutaan ottavan sopimukset jollain tavalla huomioon, ComTestin

makrokielen tulkkaajaa pitäisi laajentaa ymmärtämään Code Contracts -laajennoksen luomia kommentteja. Tämän ei pitäisi olla hankalaa, sillä sopimusten kommentit ovat eksplisiittisia ja ne noudattavat kielen omaa syntaksia. Lisäksi kommenttien sisältö on koodikieltä, eikä vapaamuotoista oikeaa kieltä. Tässä tutkimuksessa ei kuitenkaan oteta tähän ongelmakenttään tarkemmin kantaa, todetaan vain että sen oletetaan olevan mahdollista edellä mainittujen seikkojen johdosta.

Kuten tutkimuksessa on käynyt ilmi, on mielekkäiden testitapausten luominen sopimusten esi- ja jälkiehtojen sekä luokkainvarianttien perusteella erittäin hankalaa ellei jopa mahdotonta. ComTest-laajennoksen toiminnallisuuden laajentaminen tähän tavoitteeseen vaatisi siis erittäin mittavan kehitystyön.

Sovelluskehittäjä saattaa epähuomiossa kirjoittaa ComTestin-makrokielellä sellaisen testitapauksen, jossa ohjelman sopimusta tullaan rikkomaan. Tällainen tapaus tulee ilmi vasta testiohjelman suorituksen aikana. Testiohjelman oletetaankin ilmoittavan virheellisistä tilanteista, joten tämän ei olettaisi olevan ongelma sovelluskehitystyön prosessissa.

4.3 Sopimusten rikkomisten yksikkötestit

Yksikkötesteillä on järkevää testata ohjelman toimiminen oikein sopimuksen rikkoutuessa. Sopimuksen rikkoutuessa ohjelman täytyy aiheuttaa oikeanlainen poikkeus tai se toimii virheellisesti. Esimerkiksi jos ohjelmaa kutsuva ei huolehdi esiehtojen täyttymisestä, ei ohjelmalla ole velvollisuutta suorittaa toimintoaan [Hoare(1969)].

Listaus 6: Esimerkkiohjelma, jossa esiehtoa rikotaan

```
public int LaskeMerkkienMaara( string merkkijono ) {
    // Esiehdolla varmistetaan, ettei ohjelmaa kutsuta
    // parametrin arvolla, joka on tyhjä osoitin.
    Contract.Requires( merkkijono != null );

    return merkkijono.Length;
}

KutsuvaMetodi() {
        string lause = null;
        int merkkienMäärä = LaskeMerkkienMaara( lause );
}
```

Tarkastellaan esimerkkiä listauksessa 6. Siinä alirutiinissa *KutsuvaMetodi* yritetään sijoittaa muuttujaan *merkkienMäärä* lauseen merkkien lukumäärä käyttäen ohjelmaa *LaskeMerkkienMaara*. Esimerkissä on määritetty esiehto, joka vaatii ohjelman *LaskeMerkkienMaara* parametrina annettavan merkkijonon joka ei ole tyhjä osoitin. Esimerkin alirutiinissa *KutsuvaMetodi* ei kuitenkaan ole esiehdon noudattamisesta huolehdittu. Esimerkistä on helppo nähdä, ettei ohjelma *LaskeMerkkienMaara* voisikaan toimia oikein tässä tilanteessa. Parametrin arvon ollessa tyhjä osoitin on kohdassa ”return merkkijono.Length;” selvä virhe, sillä tyhjällä osoittimella ei voi olla attribuuttia *Length*. Ilman esiehtoa ohjelmassa aiheutuisi kyseisessä kohdassa poikkeus **NullReferenceException**.

Listauksen 6 ohjelmalle *LaskeMerkkienMaara* olisi järkevää kirjoittaa yksikkötesti, joka testaa sopimusrikkeen virhepoikkeuksen lähettämisen tällaisessa tilanteessa. Yksikkötestissä pitäisi kutsua ohjelmaa parametrin arvon ollessa tyhjä osoitin ja yksikkötestin onnistuminen tapahtuisi vain kun tällaisessa tilanteessa tapahtuu sopimuksen rikkomisen virhepoikkeus. Tällaisen testin voisi ohjelmoija kirjoittaa itse myös käyttämällä ComTest-työkalun makrokieltä.

Tutkimuksessa myös havaittiin, että Code Contractsin sopimusten rikkomisten aiheuttamat poikkeukset kannattaa käsitellä hieman yksikkötesteissä totutusta tavasta. Microsoft suosittelee testiympäristön alustuksessa määritettävän tapahtumakoukun, jossa asetetaan sopimusrikkeistä johtuvat poikkeukset käsiteltäväksi tavallisina yksikkötestien epäonnistumisina [Code Contracts User Manual]. Tällä tavalla yksikkötestejä suorittaessa halutut sopimusten rikkomiset eivät aiheuta poikkeuksia vaan tavallisia testausvirheitä. Yksikkötesti voidaan kirjoittaa odottamaan tällaista virhettä, jolloin voidaan testata ohjelmien oikeellinen toimiminen myös sopimuksia rikkovissa tilanteissa. ComTest-työkalussa pitäisi ottaa huomioon tällainen tapa kirjoittaa yksikkötesti, jos sen halutaan muodostavan sopimusrikkeen virhepoikkeuksen testaavia yksikkötestejä.

5 Yhteenveto

Tässä tutkimuksessa tarkasteltiin ComTest-työkalun ja Code Contracts -laajennoksen suhdetta ja niiden mahdollista yhteiskäyttöä. Code Contracts -laajennos on menetelmä, jolla .NET-kielissä voidaan toteuttaa sopimuspohjaista suunnittelua [Code Contracts User M
Siinä annetaan ohjelmille ehtoja joita sekä ohjelman kutsujan että kutsutun ohjelman tulee noudattaa [Meyer(1992)]. Code Contracts -laajennos kirjoittaa (koodikieliset)

ehdot luokkien julkisiin kommentteihin. ComTest-työkalulla puolestaan voidaan luoda yksikkötestejä koodin kommentteihin kirjoitettujen ohjeiden mukaisesti. Testien luominen tapahtuu ComTestin omalla makrokielellä [Lappalainen ym.(2010)].

Code Contracts -laajennos lisää omat kommenttinsa käännösvaiheessa ohjelman julkisiin kommentteihin. Tämän jälkeen ComTestin kehitystyökalun laajennos tulkitsee julkisiin kommentteihin sen omalla makrokielellä kirjoitettuja yksikkötestien kuvauksia. Näistä kuvauksista se muodostaa ohjelmakielisiä yksikkötestejä. ComTest ei kuitenkaan osaa tulkita Code Contracts -laajennoksen kirjoittamia kommentteja. Tulkitsemisen pitäisi olla mahdollista, sillä Code Contractsin luomat kommentit sisältävät ehdot ohjelmakielisinä lauseina. Tämä vaatisi ComTest-työkalun tulkin laajentamista ymmärtämään Code Contracts -laajennoksen sopimusten ehtojen kuvaavien kommenttien formaattia.

Kuten tutkimuksessa todettiin luvussa 3, on mielekkäiden testitapausten luominen pelkästään sopimusten ehtojen perusteella erittäin hankalaa. Sopimusten ehdot rajaavat mahdollisia ohjelmien käyttötapauksia. Testitapauksia luodessa on mahdollista automaattisesti tarkastaa, onko testitapauksen luoma tilanne edes mahdollinen. Itse asiassa tällaisen tarkastuksen tekee viimeistään kääntäjä (testiohjelma ei käänny) tai testitapaus (testin lopputulos ei ole odotettu lopputulos). Esiehtojen tapauksessa sopimuksissa määritellään millaiset ehdot kutsuvan ohjelman täytyy toteuttaa ohjelmaa kutsuttaessa. Yksikkötestit sen sijaan tarkastelevat ohjelman sisäistä toimintaa eli tässä on jonkinlainen ristiriita. Kuitenkin yksikkötesteissä voidaan varmistua ohjelman heittävän ajonaikainen poikkeus sopimuksen rikkomisesta [Code Contracts User Manual]. Loppuehtojen tapauksessa sopimuksissa määritellään millaiset ehdot kutsutun ohjelman täytyy toteuttaa ohjelman suorituksen päättyessä. Jälkiehto siis rajaa millaisen tuloksen ohjelma antaa. Yksikkötesteillä pyritään tarkastamaan ohjelman päätyvän oikeelliseen tulokseen. Jälkiehdoille voidaan kirjoittaa sopimuksen rikkomisen tarkastus samaan tapaan kuin esiehdollekin [Code Contracts User Manual]. Luokkainvarianttien ehtojen sopimusrikkeistä voidaan niin ikään myös kirjoittaa yksikkötestejä [Code Contracts User Manual]. ComTest-työkalulla voisi automatisoida tällaisten yksikkötestien luominen. Sekin vaatisi ComTest-tulkin laajentamista ymmärtämään Code Contractsin luomia kommentteja. Lisäksi se vaatisi toiminnallisuuksien lisäämistä ComTestin kehitystyökalun laajennokseen, jotta se osaisi kirjoittaa tällaiset yksikkötestit oikealla tavalla.

Mielekkäiden testitapausten luominen sopimusehtojen perusteella ei välttämättä kuitenkaan ole täysin mahdotonta. Esimerkiksi *Pex*-laajennos analysoi .NET-kielillä

toteutettua koodia ja etsii siitä mahdollisia syötteitä ja tulosteita joilla ohjelma kaatuu tai käyttäytyy väärin [Getting Started with Microsoft Pex and Moles]. *Pex* tutkii ohjelman koodia ja tekee sitä kautta arvauksia syötteistä ja tulosteista, jotka voivat aiheuttaa ongelmia. Samalla menetelmällä olisi ehkä mahdollista tutkia esi- ja jälkiehtojen sekä luokkainvarianttien vaikutusta mielekkäisiin testitapauksiin. Tämän mahdollisuuden tutkiminen on kuitenkin oma tutkimuksensa, joten sitä ei tässä käsitellä tämän syvällisemmin.

Lisäksi on syytä muistuttaa, ettei tässä tutkimuksessa ole huomioitu ComTestin ja Code Contracts -laajennoksen yhteiskäytön opetuksellisia hyötyjä. Tällaisia saattaa hyvinkin olla olemassa, joten niihin keskittyviä tutkimuksia varmastikin kannattaisi tehdä.

Lähteet

[Beck(2005)] Beck, Kent 2005. *Test-Driven Development: By Example*, Addison-Wesley, 2005.

[Beck, Andres(2005)] Beck, Kent, ja Andres, Cynthia. 2005. *Extreme Programming Explained*, Addison-Wesley, 2005.

[Code Contracts User Manual] Microsoft Corporation, *Code Contracts User Manual*, saatavilla www-muodossa <url: `http://research.microsoft.com/en-us/projects/contracts/`>, viitattu 17.3.2015.

[Getting Started with Microsoft Pex and Moles] Microsoft Corporation, *Getting Started with Microsoft Pex and Moles*, saatavilla www-muodossa <url: `http://research.microsoft.com/en-us/projects/pex/`>, viitattu 14.4.2015.

[Hoare(1969)] Hoare, C. A. R. 1969. *An Axiomatic Basis for Computer Programming*, Communications of the ACM, 12-10 (1969), s. 576–580 ja 583.

[Lappalainen ym.(2010)] Lappalainen, Vesa, Jonne Itkonen, Ville Isomöttönen ja Sami Kollanus, *Comtest: A Tool to Impart TDD and Unit Testing to Introductory Level Programming*, saatavilla www-muodossa <url: `https://trac.cc.jyu.fi/projects/comtest/wiki`>, viitattu 17.3.2015.

[Meyer(1992)] Meyer, Bertrand 1992. *Applying "Design by Contract"*, Computer (IEEE), 25-10 (1992), s. 40–51.