Perinnekoodin refaktorointi ja testattavuus

Teemu Ruotsalainen

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Erkki Mäkinen
Toukokuu 2014

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Teemu Ruotsalainen: Perinnekoodin refaktorointi ja testattavuus
Pro gradu -tutkielma, 51 sivua, 5 liitesivua
Toukokuu 2014

Ohjelmistoalan kehittyessä kasvavalla vauhdilla on entistä yleisempää löytää järjestelmästä perinnekoodia. Tekniikat uusiutuvat nopeasti, ja kiihtyvän kilpailun myötä aikataulut tiukentuvat. Paras tapa välttyä perinnekoodilta on tehdä paljon testejä. Järjestelmien koon kasvaessa ja monimutkaistuessa testaus on entistä olennaisempaa ohjelmistokehityksessä. Jos perinnekoodia on kuitenkin päässyt syntymään, pääsee siitä eroon refaktoroimalla. Tässä työssä pyrin selventämään, mitkä koodin ominaisuudet ovat tärkeitä ohjelmistokehityksessä, jotta koodista tulisi ylläpidettävää ja testattavaa. Tutkin myös, kuinka refaktorointia tulisi suorittaa. Lopuksi sovelsin ominaisuuksia ja refaktorointia telekommunikaatiojärjestelmässä olevaan perinnekoodiin. Perinnejärjestelmän ympäristön ja tekniikoiden takia refaktoroinnin mahdollisuudet kuitenkin ovat rajatut. Kyseisessä ohjelmistossa tärkeintä on pilkkoa koodia pienempiin kokonaisuuksiin ja hyödyntää yksikkötestejä.

Avainsanat ja -sanonnat: koodin ominaisuudet, testattavuus, perinnekoodi, refaktorointi, telekommunikaatio.

Sisällys

1. JOHDANTO 1

2. KOODIN YLEISIÄ OMINAISUUKSIA 3

2.1 SELKEYS 3

2.1.1 NIMEÄMINEN 3

2.1.2 KOMMENTOINTI 5

2.1.3 TYYLIASU 5

2.2 RAKENNE 6

2.2.1 LUOKAT JA TIETORAKENTEET 7

2.2.2 FUNKTIOT 8

2.3 VAKAUS 9

2.3.1 VAKUUTUKSET 10

2.3.2 VIRHEIDENKÄSITTELY 11

2.3.3 POIKKEUKSET 13

2.3.4 SÄIKEISTYS 14

2.4 OMINAISUUKSIEN SOVELTAMINEN ERILAISIIN JÄRJESTELMIIN 15

3. OHJELMISTON TESTAUKSEN KANNALTA TÄRKEÄT OMINAISUUDET 17

3.1 MODULAARISUUS 18

3.1.1 YHTENÄISYYS 19

3.1.2 RIIPPUMATTOMUUS 19

3.2 TESTIVETOINEN KEHITYS 21

4. PERINNEKOODIN REFAKTOROINTI 23

4.1 PERINNEKOODI 23

4.2 REFAKTOROINTI 24

4.2.1 HAISEVA KOODI 27

4.2.2 Riippuvuuksien rikkominen 28

4.2.3 Toiminnallisuuden testaus 29

4.2.4 Koodin muokkaus 30

4.3 Automatisoitu refaktorointi 31

5. Tapaustutkimus Nokia: Konfiguraationhallinnan analysointi ja refaktorointi 33

5.1 Tutkimuksen suorittaminen 33

5.2 Ympäristö 34

5.2.1 Radio Network Controller 34

5.2.2 Ohjelmistoarkkitehtuuri 36

5.3 Tutkimuksen kohde 38

5.4 Miksi arkkitehtuuria pitäisi uudistaa? 40

5.5 Järjestelmän koodin analysointi 41

5.6 Koodin refaktorointi testattavammaksi 44

6. Yhteenveto ja päätelmät 46

Viiteluettelo 48

Liitteet 52

1. Johdanto

Monissa yrityksissä painitaan perinnekoodin eli ajan myötä rappeutuneen koodin kanssa päivittäin. Ominaisuuksien lisääminen ja ylläpito on vaikeaa, koska ohjelmistolla ei ole riittävän kattavaa testauskokonaisuutta. Koodi voi olla huonosti tehtyä ja sitä voi olla vaikea ymmärtää ilman alkuperäisen tekijän tukea, sillä koodin lukeminen on aina vaikeampaa kuin sen kirjoittaminen. Perinnekoodia voi syntyä lukemattomista eri syistä, joten on hyvin todennäköistä, että jokainen ohjelmoija joutuu jossain vaiheessa sen kanssa tekemisiin. Perinnekoodin hallinta kuluttaa paljon resursseja, mutta usein ajatellaan, että perinnekoodin refaktorointi, eli koodin parantaminen ilman toiminnallisuuden muuttamista, on liian työlästä, sen tulokset eivät näy välittömästi asiakkaalle eikä siihen ole aikaa. Hyvin ja perusteellisesti suoritettu refaktorointi voi kuitenkin pidemmän päälle osoittautua todella kannattavaksi. Ylläpidettävyys, testaus ja uusien ominaisuuksien lisäys helpottuvat, koodi selkeytyy ja tehdyt virheet vähenevät [Kim et al., 2012].

Refaktorointi on hyvä suorittaa suunnitelmallisesti. Suunnittelematta suoritettu refaktorointi voi pahimmillaan rikkoa koko järjestelmän. On tärkeää varmistaa, ettei muuteta toiminnallisuutta, vaan pelkästään koodin rakennetta ja ulkoasua. Sen takia testaaminen on olennainen osa refaktorointia. Normaalisti testien ajatellaan varmistavan, että järjestelmä toimii virheettömästi, mutta refaktoroinnin yhteydessä testit varmistavat, että järjestelmä toimii juuri niin kuin se toimi ennen refaktoroinnin aloittamista. Ei kiinnitetä siis huomiota siihen, onko toiminnallisuus vaatimusten mukaista vai ei.

Luvussa 2 kerron yleisistä koodin ominaisuuksista, jotka tulisi muistaa aina ohjelmoidessa ja refaktoroidessa. Ilman näitä ominaisuuksia järjestelmän ylläpito tulee olemaan mahdotonta ja projektin onnistuminen hankaloituu huomattavasti. Luvussa 3 esittelen tärkeitä ominaisuuksia testauksen näkökulmasta. Ne helpottavat testauksen käyttöönottoa ja ovat nykypäivänä erittäin tärkeitä arkkitehtuuria ja tekniikoita suunniteltaessa. Luvussa 4 kerron tehokkaasta tavasta refaktoroida. Menetelmällä varmistetaan, että järjestelmä toimii refaktoroinnin jälkeenkin. Luvussa 5 suoritan tapaustutkimuksen laajalle, sulautetulle ja hajautetulle järjestelmälle liittyen perinnekoodin ominaisuuksiin ja refaktorointiin. Tapaustutkimuksen keskeisenä tehtävänä on

saada järjestelmään testattavampia komponentteja, jotta testaus voitaisiin suorittaa erillään muista komponenteista ja mahdollisesti virtualisoidussa ympäristössä. Työn päätelmät ja yhteenveto ovat esiteltyinä luvussa 6.

2. Koodin yleisiä ominaisuuksia

Koodin kirjoittaminen on yksinäistä puuhaa ja tehdyt ratkaisut ovat usein yksittäisen suunnittelijan tekemiä, vaikka ketterät kehitysmenetelmät ovatkin vieneet ohjelmointia sosiaalisempaan suuntaan. Monesti kuitenkin ”lopullisesta” tuotoksesta unohtuu se, että koodia joudutaan luultavasti vielä muokkaamaan jälkeenpäin. Se saattaa tapahtua muutaman minuutin, muutaman päivän tai monen vuodenkin jälkeen. Tällöin on todella tärkeää, että koodista saa selvää vielä silloinkin, kun alkuperäinen ohjelmoija on jo aikaa sitten unohtanut kyseisen koodin olemassaolon. Hyvä ohjelmoija osaa kirjoittaa koodia muille, ei vain itselleen tai kääntäjälle, ja ohjaa lukijan huomion tärkeisiin asioihin [Goodliffe, 2006]. Hieman ristiriitaisesti sanottuna; ohjelmoi niin, että sinut voidaan korvata milloin vain. Seuraavaksi esittelen koodin selkeyttä, hyvää rakennetta ja vakautta, jotka kaikki helpottavat koodin lukua, ymmärtämistä ja ylläpitoa.

2.1 Selkeys

Selkeä koodi on todella tärkeää modernissa ohjelmistokehityksessä. Ohjelmistoja kehitetään globaalisti, jolloin samalla koodilla voi olla tekijöitä kymmeniä tai satoja. Selkeä koodi on luettavaa ja helposti ymmärrettävää. On tärkeää keskittyä hyvään nimeämiseen, kommentointiin ja tyyliasuun, joita käsittelen tarkemmin jatkossa. Itsekseen koodatessa on helppo ajatella, ettei koodia kuitenkaan kukaan muu koskaan lue, joten miksi siis vaivautua tekemään selkää koodia, kun paljon nopeammin saa rumempaa jälkeä. Onko kuitenkaan järkevää opetella ammattimaista koodia vasta silloin, kun tietää, että joku muu tulee ja arvioi koodiasi? Hyvä ohjelmointityyli tehostaa myös omaa työskentelyä.

2.1.1 Nimeäminen

Koodissa on runsaasti rakenteita, jotka joudutaan nimeämään; muuttujat, funktiot, luokat, paketit ja tiedostot. Martin [2008] painottaa, että on tärkeää osata antaa näille kuvaavat ja yksiselitteiset

nimet, sillä muuten koodista on aivan mahdoton nähdä, mitä sillä tarkoitetaan, ja ohjelmiston toiminnan ymmärtämiseksi joudutaan käyttämään paljon kommentteja. Hyvin nimetty koodi tarvitsee harvoin kommentteja, sillä se selittää itse itsensä [Goodliffe, 2006]. Tarkoituksen mukaan nimetyt muuttujat ja elementit paljastavat lukijalle olennaisen sisällön.

Nimien tulisi mieluummin olla pitkiä ja kuvaavia kuin lyhyitä ja epäselviä. Aina tulisi pyrkiä mahdollisimman tiivistettyihin ratkaisuihin, mutta sitä ei pidä tehdä selkeyden kustannuksella. Nimeämisessä kannattaa varoa myös väärinymmärrysten syntyä. Sanat, joilla on monta merkitystä, voivat aiheuttaa turhaa aivotyötä tai johtaa jopa virheisiin. Monesti myös nimet, joilla on vain hyvin vähän eroa, saavat aikaan hämmennystä. Kannattaa varoa niin ikään esimerkiksi pienen L-kirjaimen (”l”) ja numero yhden (”1”) sekä nollan (”0”) ja suuren O-kirjaimen (”O”) käyttöä sekoittavasti, sillä ne näyttävät usein lähes identtisiltä. [Martin, 2008] Aina kun mahdollista, tulisi käyttää vakioita. Vakioiden käyttö helpottaa todella paljon koodista etsimistä ja tietysti vakion arvon päivitystä. [Goodliffe, 2006]

Martin [2008] mainitsee myös, että luokkien nimien tulisi olla substantiiveja ja funktioiden nimien verbejä. Tämä on varsin loogista, sillä luokat yleensä kuvaavat jotain tiettyä asiaa, kun funktiot taas suorittavat jonkun toiminnon, esimerkiksi hakevat tai muuttavat tietoa. Funktioiden nimien keskimitta on vain 9-15 merkkiä, mutta ne voisivat aivan hyvin olla 20-35 merkkiä pitkiä. Jos funktio palauttaa jonkin muuttujan, tulisi nimi antaa sen mukaan. [McConnell, 2004] McConnellin [2004] mukaan hyvä ohjenuora funktion nimen antoon on, että ensin tulee vahva verbi, jota seuraa objekti. Goodliffe [2006] myös lisää, että täytesanoja tulisi niin ikään välttää, kuten esimerkiksi ”manager”, ”class”, ”object” ja ”data”, jotka eivät tarjoa itsessään mitään hyödyllistä informaatiota. Hän mainitsee myös, että koodiin ei kannata soluttaa vitsejä tai kaskuja, sillä ne voivat antaa lukijalle epäammattimaisen kuvan ohjelmoijasta.

Nimeämisen tulisi olla myös yhtenäistä koko ohjelmistossa. Jos esimerkiksi haku-funktioille päätetään antaa etuliite *get,* tulisi se antaa jokaiselle haku-funktiolle järjestelmässä [Martin, 2008]. McConnell lisää myös, että vastakohtia tulisi hyödyntää. Esimerkiksi *get* ja *set* tai *add* ja *remove.* Nimien keksimiseen käytetään aivan liian vähän aikaa. Nimien miettimiseen ja arviointiin tulisi käyttää resursseja, sillä se on kannattavampaa pidemmän päälle. [Goodliffe, 2006]

2.1.2 Kommentointi

Kuten aiemminkin jo mainittiin, hyvin nimetty ja rakennettu koodi ei tarvitse kommentteja juuri lainkaan [Martin, 2008]. Kommenteilla ei tulisi täydentää tai oikeuttaa huonoa koodia. Niitä ei tarvita myöskään selostamaan, mitä kullakin koodirivillä tapahtuu. [Goodliffe, 2006]

Yleensä koodia kirjoitettaessa kommentti voi olla ohjelmoijan mielestä tarpeellinen, mutta koodi kuitenkin kehittyy ja usein kommentit jäävät ennalleen. Hyvin kommentoituun funktioon on voitu vuoden päästä tehdä lisää ominaisuuksia tai sen toimintaa on voitu muuttaa, mutta sama vanha kommentti on edelleen sekoittamassa koodin lukijaa. On myös mahdollista, että koko funktio on voitu siirtää toiseen tiedostoon ja kommentti jää siirtämättä. Ohjelmoijien usein käyttämää tekniikkaa, koodin pois kommentointia, ei kannata käyttää. Tällaisissakin tilanteissa usein kuitenkin käy niin, että pois kommentoitu koodi jää vain lojumaan, vaikka alkuperäinen tarkoitus oli olla vain väliaikainen ratkaisu. Mahdollisilla uusilla kehittäjillä ei riitä uskallusta niitä poistaa, koska he eivät tiedä, onko kyseessä jotain tärkeää. [Martin, 2008]

Toki on tilanteita, jolloin kommentit ovat oikeutettuja. Goodliffe [2006] mainitsee, että käytettäessä kommentteja niiden tulisi kertoa, miksi jotakin tehdään, eikä miten se on tehty. Hän myös toteaa, että joskus voi olla hyvä jättää kommentti sellaiseen paikkaan, josta voi tapahtua jotakin arvaamatonta. Poikkeustapauksia ovat myös esimerkiksi Javalla kirjoitetut julkiset kirjastot, joille Javadoc:t ovat lähes välttämättömiä sujuvan käytön takaamiseksi. Myös hyödynnettäessä kirjastoja, joita ei voida muokata, voidaan selittää esimerkiksi, mitä kirjastoon tehty funktiokutsu tekee. Joissain tapauksissa voidaan jättää ”TODO”-kommentti, jotta tiedetään palata takaisin tekemään ominaisuus valmiiksi, mutta sitäkin tulee käyttää harkiten. [Martin, 2008]

2.1.3 Tyyliasu

Huolellisesti muotoiltu tyyliasu on myös erittäin tärkeä tekijä koodin helppolukuisuuden takaamiseksi. Usein koodi itsessään voi muuttua ajan saatossa, mutta tyyliasu säilyy [Martin, 2008]. Tyyliasun muokkaaminen on helpottunut paljon viime aikoina kehittyneiden ohjelmointiympäristöjen ja koodinmuokkausohjelmien myötä. Niillä voi automaattisesti muuttaa

koodin ulkonäköä ja rakennetta annetun kaavan mukaiseksi. Niihin ei kuitenkaan tulisi luottaa liikaa, vaan aina ensisijaisesti kirjoittaa alusta asti hyvän tyyliasun mukaista koodia.

Tyyliasun muotoilussa tärkeää on välttää liian isoja kokonaisuuksia. Martin [2008] tuo esille, että yhdelle riville ei tule mahduttaa liikaa tavaraa. Hän tosin myös huomioi, että nykyinen 80 merkin standardi on hieman vanhentunut, sillä näyttöjen pikselimäärä on kasvanut huimasti verrattuna aikoihin, jolloin kyseinen kultainen sääntö muodostettiin. Lähdekooditiedostojakaan ei saa paisuttaa liian suuriksi. Ne pitää jäsentää niin, että korkeamman tason funktiot ovat ylempänä ja matalamman tason alempana. Näin koodi etenee abstraktimmasta yksityiskohtaisempaan päin. Pitää kuitenkin muistaa välttää asiayhteyksien pirstomista erilleen, eli toisiinsa liittyvät kokonaisuudet tulisi olla mahdollisimman lähekkäin. [Martin, 2008]

Koodia selkeyttävät myös oikein ja systemaattisesti käytetyt sisennykset. Ne luovat tärkeän hierarkkisen rakenteen, josta näkee yhdellä vilkaisulla, mitkä ovat elementtien väliset suhteet. Rivinvaihdoilla voidaan taas kätevästi erotella toisistaan kokonaisuuksia ja toimintoja. [Martin, 2008]

2.2 Rakenne

Koodin rakenne muodostuu erilaisten tietorakenteiden ja funktioiden käytöstä ja niiden muodostamista kokonaisuuksista. Rakenteeseen vaikuttaa voimakkaasti ohjelmistolle määritetty arkkitehtuuri. Ohjelmistoarkkitehtuuri määrittelee koko järjestelmän järjestyksen, komponentit ja niiden yhteydet, sekä rajapinnat [Gorton, 2006]. Ohjelmoijalle jää kuitenkin, etenkin laajemmissa järjestelmissä, vielä tämän jälkeen suuri vastuu luokkien, tietorakenteiden ja funktioiden toteutuksesta [McConnell, 2004]. Rakenteilla on lähes yhtä suuri merkitys koodin luettavuuteen kuin koodin selkeydellä.

Ohjelmointi on aina iteratiivista toimintaa, ja toteutuksessa joudutaan usein palaamaan jo aikaisemmin tehtyihin ratkaisuihin ja itse toteutukseen. Martin [2008] ohjeistaakin, että alkuun voi kirjoittaa miten huonoa koodia tahansa, mutta toiminnallisuuden ollessa valmis kannattaa se refaktoroida hyvän mallin mukaiseksi.

Ohjelmistot kehitetään lähes poikkeuksetta tiimeissä. Jokaisella tiimin jäsenellä on oma taustansa ja osaamisensa. Jotkut ovat tehneet ohjelmistoprojekteja 20 vuotta, ja jotkut taas voivat olla itseoppineita ohjelmoijia. Ryhmän olisi kuitenkin hyvä sopia yhteinen ohjelmiston toteutustyyli, jota jokainen jäsen noudattaa [Goodliffe, 2006]. Ei toki kannata alkaa keksimään täysin uutta tyyliä, vaan ottaa jokin yleisimmistä tyyleistä ja käydä yhdessä ryhmän kanssa läpi, mitä kukin ajattelee mistäkin asiasta ja näin muodostaa kaikkien yhteinen näkemys. Jokainen joutuu tekemään kompromisseja omasta näkemyksestään, mutta aina tulisi olla avoimena uusille näkökulmille, eikä vain pitää omaa tapaansa ainoana oikeana. [Goodliffe, 2006]

Jos tehtäväksi tulee ylläpitää jonkun muun tekemää koodia, tulisi aina käyttää jo olemassa olevaa tyyliä, olettaen tietysti, että käytetty tyyli on selkeä ja yhtenäinen [Goodliffe, 2006]. Martin [2008] mainitsee myös partiopoika-säännön (The Boy Scout Rule), joka käskee jättämään koodin aina parempaan kuntoon kuin mitä se oli, kun sen sai käsiinsä. Aina löytyy parannettavaa.

2.2.1 Luokat ja tietorakenteet

Luokkien ja tietorakenteiden tulisi olla mahdollisimman kompakteja ja yhtenäisiä yksiköitä. Koodirivien ja funktioiden määrässä tulee pysyä kohtuudessa. Luokissa ei myöskään tule olla liikaa funktioita, jotka hyödyntävät vain pientä osaa luokan omista attribuuteista. Luokilla tulisi olla vain yksi tarkoitus. [Martin, 2008]

Kapselointi, eli tietorakenteiden näkyvyyden rajoittaminen luokan ulkopuolelle, on myös erittäin tärkeää käsiteltäessä luokkia ja niiden attribuutteja. Se asettaa abstraktion asiakaskoodin suuntaan, jotta rakennetta olisi helpompi käsitellä. Usein muuttuvien luokkien yhteydessä tulee hyödyntää myös rajapintoja. Näin luokan sisäinen toiminta piilotetaan asiakaskoodilta täysin ja sitä käytetään vain rajapinnan kautta. Luokan sisäiset muutokset eivät riko ulkoisia kutsuja, kunhan se vain toteuttaa rajapinnan. Jokaiselle toiminnolle kannattaa myös tehdä omat funktionsa, jotta asiakkaan ei tarvitse ketjuttaa toimintoja. [Martin, 2008] Esimerkiksi luokan instanssista, joka pitää yllä hedelmäkorin tietoja, voisi löytyä seuraavanlainen tilanne: *fruitBasket.getFruits().size()*. Tässä siis halutaan saada hedelmien määrä selville. Tilanteesta

voitaisiin saada hieman selkeämpi luomalla luokkaan *getFruitsCount()* -metodi, jolloin voidaan vain kutsua: *fruitBasket.getFruitsCount()*.

2.2.2 Funktiot

Funktioiden tarkoitus on suorittaa jokin toiminto eristämällä se muusta koodista, jotta sitä voidaan käyttää huolehtimatta sen toteutuksesta ja muu koodi yksinkertaistuu. Hyvin suunniteltu funktio antaa asiakaskoodin vain keskittyä siihen, mitä tehtiin, eikä siihen, miten se on tehty. [Kernighan and Ritchie, 1988] Funktion tehtävä voi olla esimerkiksi jonkun tietorakenteen päivittäminen, joka ei palauta mitään, tai vastaaminen asiakaskoodin kyselyyn [Martin, 2008].

Martin [2008] mainitsee, että yhdellä funktiolla tulisi olla vain yksi toiminnallisuus. Mikäli funktiolla kertyy monia eri tehtäviä, kannattaa ne jakaa alafunktioihin. Näitä Martin [2008] kutsuu abstraktiotasoiksi. Näin ollen funktioille rakentuu selkeä hierarkia, joissa ylemmän tason funktiot tarjoavat abstraktimman toteutuksen ja yksityiskohdat piilotetaan alemmalle tasolle. Samalla funktioista on helppo tehdä vain muutaman rivin mittaisia ja helposti ymmärrettäviä, jolloin yhdellä silmäyksellä pystyy näkemään funktion toiminnallisuuden. McConnellin [2004] mukaan hyvä indikaattori siitä, että funktio tarvitsee alifunktion, on kun siinä esiintyy sisäkkäisiä ehtolauseita, silmukoita tai esimerkiksi try-catch -rakenteita.

Funktiot ovat erityisen käteviä myös toistuvan koodin vähentämiseen [Martin, 2008]. Hyvä ohjelmoija löytää toiminnoista toistuvia geneerisiä ominaisuuksia, jotka voidaan toteuttaa yhdessä funktiossa. Näin koodista tulee myös luotettavaa ja helposti ylläpidettävää, sillä mahdolliset muutokset ja testaus voidaan suorittaa keskitetysti. Vaikka toistuva koodinpätkä olisi kuinka pieni, ei kannata pelätä sen toteuttamista funktiona, mikäli se helpottaa koodin lukemista. Usein pienet operaatiot paisuvat ja näin koodia tarvitsee lisätä vain yhteen paikkaan. [McConnell, 2004] Beck ja Diehl [2011] kuitenkin huomioivat, että koodin kloonausta voidaan hyödyntää esimerkiksi suorituskyvyn ollessa kriittinen ominaisuus, haarukoidessa (eng. forking), automatisoidussa ohjelmoinnissa tai kun riippuvuuksia pyritään vähentämään.

Funktioiden määrässä kannattaa kuitenkin pysyä kohtuudessa, varsinkin mobiilialustojen ja massiivisten järjestelmien yleistyessä. Varoittavana esimerkkinä toimii Facebookin Android-sovellus, jonka yli 3 miljoonaa metodia olivat liikaa Android-käyttöjärjestelmälle [Reiss, 2013].

Funktioita tehdessä täytyy varoa toiminnallisuuksien piilottamista asiakaskoodilta. Niihin ei tule lisätä mitään ylimääräisiä toiminnallisuuksia, joita ei funktion nimestä ilmene, vaikka se olisikin vain jokin pieni ja mitätön asia. Pieniltä ja mitättömiltä tuntuvat asiat voivat usein saada aikaan suuriakin virheitä järjestelmässä, ja mikäli ne on piilotettu niille kuulumattomiin paikkoihin, on niitä todella vaikea löytää. [Martin, 2008]

Funktio on parhaimmillaan silloin, kun sillä ei ole ainuttakaan parametria. McConnell [2004] toteaa, että funktion parametrit tulisi rajoittaa seitsemään. Hän tosin lisää, että varsinkin primitiiviset kielet saattavat vaatia joskus useampiakin parametreja. Martin [2008] puolestaan esittää, että yksi tai joskus kaksi parametria on vielä hyväksyttävää, mutta tätä enempää ei oikeastaan kannata käyttää. Hän tosin keskittyy lähinnä Java-kieleen, joka on varsin korkean tason kieli. Toki hänellä on näihinkin poikkeuksia kuten esimerkiksi argumenttilistat. Erityisesti testauksen kannalta on tärkeää rajoittaa parametrien määrää, jotta testiskenaarioiden määrä pysyisi pienenä. Jos funktiolle annetaan Boolean-muuttuja parametrina, kannattaa harkita kahden erillisen funktion tekemistä. [Martin, 2008] Koskaan ei tulisi kuitenkaan käyttää parametreja muuttujina funktiossa, vaan pitäisi luoda paikalliset muuttujat. Hyvänä käytäntönä pidetään niin ikään oletusarvon palauttamista funktiossa. Sillä säästetään tapaukset, joissa arvo jää kokonaan palauttamatta. [McConnell, 2004]

2.3 Vakaus

Aikoinaan oli tapana sanoa, että jos ohjelmalle syöttää roskaa, saa roskaa takaisin. Eli vika oli käyttäjässä. Nykyään vikasietoisuus on yksi merkittävimmistä ominaisuuksista kehitettävälle ohjelmistolle [McConnell, 2004]. Vakaudella ei välttämättä tarkoiteta, että ohjelma on virheetön, vaan että virheet huomioidaan ja niihin reagoidaan heti. Hyvä keino vakauden tavoittelemisessa on olettaa, että aina voi tapahtua mitä vaan. Koskaan ei tulisi luottaa käyttäjään tai toiseen ohjelmistokomponenttiin, eli tulisi suojata oma ohjelma myös muiden tekemiltä virheiltä. Mitään

oletuksia ei saa tehdä. Tällaista menetelmää kutsutaan *puolustavaksi ohjelmoinniksi* (defensive programming). [McConnell, 2004; Goodliffe, 2006]

Ohjelman vakaudessa ei ole kyse vain pelkästään koodin pettämisestä, vaan on otettava huomioon myös ympäristön, kuten verkon, laitteiston ja ihmisten aiheuttamat vikatilanteet. Vakaan ohjelmiston toteuttaminen on paljon aikaa ja resursseja vievää toimintaa, mutta se vapauttaa kuitenkin usein enemmän resursseja ylläpidollisista tehtävistä. Virheitä kestävän ohjelman tekemistä helpottaa huomattavasti selkeän koodin kirjoittaminen. Kiireessä on lähes mahdotonta saada käsiteltyä kaikkia vikatilanteita. [Goodliffe, 2006]

Goodliffe [2006] huomioi, että usein ohjelmoijat eivät välitä toistuvista kääntäjien varoituksista tai pahimmassa tapauksessa jopa piilottavat ne. Varoitusten analysointi on kuitenkin todella tärkeä apu vakaan koodin tekemiseen. Samoin Goodliffe painottaa, että koodin analysointityökaluja tulisi hyödyntää sekä käyttää ensisijaisesti turvalliseksi luokiteltuja tietorakenteita. Resurssit tulisi myös vapauttaa aina heti, kun mahdollista. Muistin lisäksi on siis muistettava vapauttaa muun muassa tiedostolukot ja säikeet. Muuttujat olisi hyvä alustaa niitä luotaessa ja luonnin pitäisi tapahtua mahdollisimman lähellä käyttöä. Muuttujien tyyppimuunnokset on tehtävä todella huolellisesti, etenkin matalan tason kielissä kuten C ja C++. Näissä kielissä tulisi myös käyttää const-määrettä aina, kun mahdollista. Tämä selkeyttää koodia, ja mikä tärkeintä, estää tehokkaasti virheitä. [Goodliffe, 2006] McConnell [2004] lisää vielä, että erityisesti ulkoisista rajapinnoista tulevaa informaatiota tulisi käsitellä erityisen tarkasti esimerkiksi tietoturvallisuussyistä.

2.3.1 Vakuutukset

Vakuutukset (eng. assertion) ovat koodin sekaan lisättäviä varmistuksia siitä, että ohjelman suorituksessa ei tapahdu mitään odottamatonta. Niillä testataan esimerkiksi, onko sallittu datamäärä ylittynyt [*assert(dataCount > MAX_DATA_COUNT)*]. Jos ehto toteutuu, annetaan asiasta virheilmoitus. [McConnell, 2004] Vakuutuksia käytetään yleensä vain ohjelman kehityksen aikana, jotta ne eivät ole turhaan kuormittamassa lopullista tuotosta. Niitä ei kuitenkaan tarvitse manuaalisesti poistaa koodista, sillä monesti kääntäjän esikäsittelijä osaa jättää ne huomiotta. [Goodliffe, 2006]

Vakuutuksia kannattaa käyttää erityisesti proseduraalisissa kielissä (C/C++). Vakuutuksia kannattaa lisätä esimerkiksi ohjelmalohkojen kutsujen parametriarvojen ja palautusarvojen tarkistukseen sekä esimerkiksi silmukoiden yhteydessä. Ne toimivat samalla dokumentointina siitä, mitä arvojen tulisi olla. Esimerkiksi parametriarvojen vakuutuksesta nähdään suoraan, mitä asiakaskoodi lupaa antaa parametrina, ja palautuksessa nähdään, mitä itse funktio lupaa palauttaa. Vakuutuksia ei tarvitse laittaa jokaiseen mahdolliseen paikkaan, vaan vain pääfunktioiden tarkistukset riittävät. Muita hyödyllisiä käyttökohteita ovat taulukoiden raja-arvojen tarkistus, objektin tilan tarkastus ennen sen käyttöä ja sellaisten paikkojen vartiointi, joihin ohjelman suorituksen ei koskaan pitäisi päätyä. [Goodliffe, 2006] McConnell [2004] lisää vielä, että yleisenä ohjeena vakuutuksien käyttöön on, että niitä tulisi käyttää sellaisiin tilanteisiin, joita ei koskaan odoteta tapahtuvan.

Vakuutuksia ei pidä käyttää kuitenkaan esimerkiksi käyttäjän virheellisiin syötteisiin, vaan näihin tulisi käyttää tavanomaista virheiden käsittelyä. Vakuutuksen ”lauetessa” toki annetaan virheviesti, mutta tämä tehdään vain sen takia, että ohjelmoija tietää sen lauenneen ja käy korjaamassa koodia niin, ettei se tapahdu uudelleen. Vakuutuksilla tarkistetaan siis virheitä itse koodissa eikä esimerkiksi syötearvoja. [McConnell, 2004]

Vakuutuksiin ei tule laittaa myöskään minkäänlaista ohjelman toiminnallisuutta, koska niitä ei enää käytetä lopullisessa tuotteessa [McConnell, 2004]. Helppo tapa asettaa ohjelmointivirheitä vakuutuksiin on myös olemalla huolimaton. Yksi hyvä esimerkki on, jos käytetään vahingossa ”=”-operaattoria ”==”-operaattorin sijasta vakuutuksen vertailussa. Tällöin kehitysvaiheessa oleva koodi toimii eri tavalla kuin tuotantoon käännetty koodi. [Goodliffe, 2006]

2.3.2 Virheidenkäsittely

Virheet voivat johtua monesta eri tekijästä. Ne voivat juontaa juurensa käyttäjän, ohjelmoijan tai ympäristön aiheuttamiin tilanteisiin. Goodliffe [2006] luokittelee virheiden ilmenemisen kolmeen eri pääkategoriaan:

- *Ohjelma ei käänny* — Virhe on helppo havaita ja paikantaa. Yleensä tällaiset virheet ovat kirjoitusvirheitä koodissa tai muita ns. helppoja virheitä, jotka on helppo korjata.

- *Ajonaikainen kaatuminen* — Korjaukset hieman vaikeampia käsitellä, mutta usein varsin yksinkertaisia.
- *Odottamaton käyttäytyminen* — Kaikkein vaikein tapaus, jonka syyn löytämiseen yleensä kuluu huomattavasti pidempi aika.

Virheidenkäsittelyyn on olemassa niin ikään myös monia eri tekniikoita. Seuraavassa on lueteltu näistä yleisimmät:

- *Palautetaan arvo, joka voi olla esimerkiksi nolla, tyhjä String-muuttuja tai osoitin tai jokin oletusarvo* — Virhe selviää vain tutkimalla palautusarvoa. Pelkän virhemuuttujankin palautusta voi käyttää (kuten Boolean-muuttuja), mutta mikäli tarvitsee palauttaa jotain muutakin kuin virhemuuttuja, tulee ongelmia. [McConnell, 2004] Martin [2008] mainitsee myös, ettei null-arvoa tulisi koskaan palauttaa, sillä se joudutaan kuitenkin tarkistamaan jossain vaiheessa.
- *Seuraavalla tai lähimmällä validilla arvolla korvaaminen* — Tämä on erityisen hyödyllinen menettely, kun käsitellään nopeasti päivittyvää tietovirtaa tai jos raja-arvo on ylitetty. [McConnell, 2004]
- *Korvataan edellisen kerran arvolla* — Voidaan käyttää muun muassa tietokonepelien ruudunpäivityksessä ilmenevissä virheissä. [McConnell, 2004]
- *Kirjoitetaan virheilmoitus lokitiedostoon* — Tätä tekniikkaa kannattaa käyttää jonkin muun tekniikan yhteydessä. [McConnell, 2004]
- *Virhekoodin antaminen* — Jos virheitä ei käsitellä paikallisesti, voidaan esimerkiksi asettaa globaali virhemuuttuja tai asettaa poikkeus. [McConnell, 2004]
- *Keskitetty virheiden hallinta* — Käytetään globaalia virheidenkäsittelijää tai virheobjektia. Tämä tekniikka kuitenkin rikkoo järjestelmän modulaarisuuden, koska jokainen komponentti on riippuvainen virheidenhallintayksiköstä. [McConnell, 2004] Goodliffe [2006] toteaa myös, ettei kyseistä tekniikkaa tulisi käyttää, sillä se ei ole säieturvallinen ja virheet voivat jäädä helposti huomioimatta.
- *Virheviestin välittäminen* — Tätä on käytettävä harkiten, sillä liiallisen tiedon paljastaminen voi aiheuttaa tietoturvariskejä. [McConnell, 2004] Goodliffen [2006] mielestä kyseistä tekniikkaa kannattaa käyttää vain, jos ei ole mitään muuta vaihtoehtoa.

- *Ohjelman sulkeminen* — Käytetään erityisesti suurta tietoturvaa tai luotettavuutta vaativissa järjestelmissä, joissa pienetkin virheet voivat olla kriittisiä. [McConnell, 2004]
- *Signaalit* — Käyttöjärjestelmä ilmoittaa signaalilla, että virhe on tapahtunut ja ohjelma ottaa virheen vastaan ja käsittelee asianmukaisesti. Tämän jälkeen ohjelman suoritus jatkuu siitä kohtaa, missä ennen virhettä oltiin menossa. Monet kielet tarjoavat signaaleille oletuskäsittelijöitä, joita voi tarvittaessa kuormittaa. [Goodliffe, 2006]
- Goodliffe [2006] mainitsee vielä lisäksi yhden menetelmän: *ei tehdä mitään* — Tätä tekniikkaa tulisi kuitenkin välttää.

Menetelmän valitseminen riippuu suuresti kyseessä olevasta tilanteesta sekä ohjelmiston luonteesta. Kuluttajatuotteet on yleensä rakennettu olemaan mahdollisimman virheenkestäviä, kun taas esimerkiksi sädehoidossa käytettävässä järjestelmässä ei suvaita minkäänlaisia virheitä. Tärkeää on kuitenkin olla koko järjestelmän laajuudelta yhtenäinen myös virheidenkäsittelyn kannalta. Ei ole myöskään viisasta käyttää kaikkia tekniikoita sekaisin. [McConnell, 2004]

2.3.3 Poikkeukset

Martin [2008] painottaa, että aina kun vain mahdollista tulisi suosia poikkeuksia. Nykyään poikkeusten käyttö on todella yleistä, mutta ne eivät ole tuettuja kaikissa kielissä. Poikkeusten kohdalla ohjelman suoritus loppuu ja lähdetään menemään kutsupinossa ylöspäin niin kauan, että löydetään kyseisen virheen käsittely [Goodliffe, 2006].

Poikkeukset voidaan jakaa kahteen eri tyyppiin: keskeyttämis- ja jatkamismalli. Keskeyttämismallissa ohjelma jatkuu virheen käsiteltyä poikkeuskohdan jälkeen, kun taas jatkamismallissa palataan siihen, missä poikkeus tapahtui. Jälkimmäisessä mallissa on siis mahdollista korjata tehty virhe. [Goodliffe, 2006]

Poikkeukset voidaan käsitellä lokaalisti tai antaa ”kuplia” ylöspäin. Yleensä, jos vain mahdollista, kannattaa käsitellä virheet heti, mutta usein virheen tapahtuessa ei tiedetä, mitä pitäisi tehdä, joten poikkeus ohjataan abstraktiotasolla ylöspäin. Kun poikkeus ohjataan ylöspäin, ei sitä voida olla huomioimatta. [McConnell, 2004] Goodliffe [2006] painottaa, että tällöin

virhetilanteen täytyy olla hyvin dokumentoitu, jotta virhe osataan käsitellä oikein. Martin [2008] lisää, että myös virheviestien tulee olla erityisen kuvaavia.

Poikkeuksia tulisi kuitenkin käyttää vain, jos ei ole muuta keinoa välttää virhetilannetta. Tämä johtuu siitä, että poikkeukset heikentävät kapselointia, sillä asiakaskoodin täytyy tietää, mitä poikkeuksia käsitellään [Martin, 2008]. Parhaimmillaan ne selventävät koodin ymmärrettävyyttä huomattavasti, mutta väärin käytettynä ne voivat tehdä koodista lähes mahdottoman lukea [McConnell, 2004].

Poikkeuksia ei kannata käyttää rakentajissa tai tuhoajissa, mikäli niitä ei voida käsitellä lokaalisti. Pahimmassa tapauksessa ne voivat johtaa muistivuotoihin, kun rakentaja ei ole suoriutunut loppuun ja tuhoajaa ei kutsuta. Poikkeuksien ohjaamisessa tulisi ottaa huomioon myös abstraktiotasot. Korkean abstraktiotason funktiossa ei pidä ohjata liian yksityiskohtaista virhettä ylöspäin. Tyhjien virhekäsittelyelementtien (kuten catch-elementti) käyttö ei myöskään ole suotavaa. Jos kuitenkaan ei nähdä muuta vaihtoehtoa, tulee dokumentoinnissa selventää, miksi näin on tehty. [McConnell, 2004]

2.3.4 Säikeistys

Säikeistys on usein houkutteleva tekniikka prosessorien kehittyessä. Mahdollisia kohteita säikeistetylle järjestelmälle ovat esimerkiksi reaaliaikaiset järjestelmät ja käyttöjärjestelmät, serveriohjelmointi sekä järjestelmät, jotka vaativat responsiivisen käyttöliittymän samalla, kun suorittavat taustalla toimenpiteitä [Duffy, 2008]. Usein kuitenkin unohdetaan, kuinka virheherkkää monen säikeen käyttäminen on. Martin [2008] suositteleekin käyttämään säikeistystä vain ja ainoastaan silloin, kun sille on todellista tarvetta, sillä tekniikka tuo mukanaan ongelmia, joita ei tarvitse miettiä käytettäessä vain yhtä säiettä.

Säikeistettyä koodia on vaikea testata, sillä testien ajaminen ei aina takaa, että kaikki menee aina samalla tavalla. Siksi testejä pitää ajaa lukemattomia kertoja yhä uudestaan, koska testien suorittaminen ei aina tapahdu identtisesti. [Martin, 2008] Erityisesti, kun joudutaan käyttämään lukkoja informaation kirjoitukseen, syntyy kilpailutilanteita säikeiden välille. Tällaiset tilanteet vaihtelevat koneen arkkitehtuurin ja monimutkaisten ajoitusten mukaan ja ovat erittäin vaikeita

toistaa. [Duffy, 2008] Virheen löytyessä pitää välittömästi selvittää, mistä virhe johtui, eikä olettaa, että virhe johtui ympäristön häiriötekijästä [Martin, 2008].

Jos kuitenkin on tarvetta säikeistykselle, on hyvä muistaa tehdä asiat oikein. Säikeistettyä koodia oppii käsittelemään vain kokemuksen kautta [Duffy, 2008]. Ensinnäkin kannattaa aloittaa säikeistämättömästä koodista ja varmistaa sen toimivuus ennen kuin siirtyy säikeistettyyn osioon. Säikeistetyn koodin erottaminen muusta koodista on myös erittäin tärkeää. Sitä voidaan testata monella eri konfiguraatiolla ja itsenäisesti. Jaettu data saadaan myös keskitettyä, jolloin on helpompi muistaa, mitkä rakenteet tulee suojata ja päivittää säieturvallisiksi. Kapselointi nousee entistäkin tärkeämmäksi, jotta asynkroninen koodi ei pääse vahingossakaan käsiksi sille kuulumattomaan koodiin. Hyvä keino välttää virheitä säikeistyksessä on myös käyttää datan kopiointia, jos ei ole tarvetta muokkaukselle. [Martin, 2008]

Säikeistetyn koodin testauksessa tulee muistaa käyttää eri alustoja. Monesti ohjelma voi käyttäytyä eri tavalla esimerkiksi Windowsissa ja OS X:ssä. [Martin, 2008]

2.4 Ominaisuuksien soveltaminen erilaisiin järjestelmiin

Jokainen järjestelmä ohjelmistoalalla on erilainen. Järjestelmät koostuvat lukemattomista määristä muuttujia, jotka yhdessä muodostavat uniikkeja kokonaisuuksia. Muuttujia ovat muun muassa koko, arkkitehtuuri, ohjelmointikieli, ympäristö ja tietysti toiminta. On mahdotonta antaa tarkkaa kuvausta siitä, kuinka ominaisuuksia tulisi hyödyntää jokaisessa järjestelmässä. Noudatettaessa suositeltuja käytäntöjä on hyvä myös käyttää maalaisjärkeä. Tässä työssä olen pyrkinyt esittämään asiat tietyllä abstraktiotasolla, jotta niitä voidaan myöhemmin soveltaa mahdollisimman moneen eri järjestelmään.

Kohdassa 2.1 esitellyt koodin selkeyteen liittyvät ominaisuudet ovat sellaisia, jotka pätevät jokaiseen järjestelmään. Ei ole olemassa syytä, joka oikeuttaisi sivuttamaan hyvän nimeämisen tai selkeän tyyliasun. Näihin ominaisuuksiin eivät vaikuta järjestelmässä olevat muuttujat, vaan ne ovat universaaleja, huolimatta siitä, millä kielellä tai alustalla töitä tehdään. Myös vakaus, jota on käsitelty kohdassa 2.3, on tärkeää jokaisessa järjestelmässä. Vakauden toteutusvaihtoehtoja

on lukuisia ja valinta riippuu järjestelmän ominaisuuksista. Ei siis voida sanoa tiettyä oikeaa tapaa vakauden aikaansaamiseksi, vaan harkinta on tehtävä järjestelmäkohtaisesti.

Kohdan 2.2 rakenteeseen liittyvät ominaisuudet pätevät suurelta osalta olio-ohjelmointiin ja proseduraalisiin kieliin. Funktionaalisissa kielissä voi olla vaikeaa saada monimutkaisia kokonaisuuksia jaoteltua esimerkiksi Martinin [2008] asettamiin rajoihin funktioiden koosta ja parametrien määrästä. Suorituskyvyn ollessa erittäin kriittinen voidaan myös hylätä osa ehdotetuista ominaisuuksista. Esimerkiksi funktiokutsut voivat kuluttaa ylimääräisiä resursseja käytettäessä vanhoja kääntäjiä, ja siksi optimoidussa koodissa funktioiden jakamista voidaan välttää. Usein optimointi on ristiriidassa selkeän ja hyvän koodin kanssa [Fog, 2014]. Järjestelmässä joudutaan tekemään kompromisseja, riippuen siitä, onko ylläpidettävyys vai tehokkuus tärkeämpää. Ohjelmistojen koon ja tietokoneiden suorituskyvyn kasvaessa tärkeimmäksi valintakriteeriksi muodostuu kuitenkin yhä useammin ylläpidettävyys.

Optimointiin on olemassa myös työkaluja, jotka optimoivat koodin automaattisesti kääntämisen yhteydessä. Itse lähdekoodi pysyy ylläpidettävänä, mutta käännetty koodi ei ole selkokielistä. Esimerkiksi ProGuard-optimoija Java-ohjelmistoille muokkaa koodin mahdollisimman kompaktiksi [ProGuard, 2014]. Se esimerkiksi lyhentää käytettyjen muuttujien nimet, poistaa käyttämättömän koodin ja jopa yhdistää toistuvia koodilohkoja. Nämä toimenpiteet pienentävät itse ohjelman kokoa, nopeuttavat toimintaa ja vähentävät muistin käyttöä. [ProGuard, 2014] ProGuardin kaltaiset työkalut vaikeuttavat myös ohjelmien takaisinmallinnusta.

3. Ohjelmiston testauksen kannalta tärkeät ominaisuudet

Ohjelmistokehityksessä esiintyy useaa eri testauksen muotoa, kuten yksikkö-, integraatio-, komponentti- ja systeemitestausta. Tässä työssä keskityn kuitenkin koodin testattavuuteen yksikkö- ja komponenttitestauksen näkökulmasta. Näitä testejä suorittavat pääasiassa ohjelmoijat itse. Luvussa 2 esitetyt ominaisuudet vaikuttavat nekin suuresti testauksen suoritukseen. Tässä luvussa kuitenkin esittelen ratkaisuja menetelmien, arkkitehtuurin ja koodin tasolla, jotka edesauttavat erityisesti testattavuutta. Ensin kerron testattavista ominaisuuksista yleisesti, ja sen jälkeen modulaarisuudesta ohjelmistoissa ja testivetoisesta kehityksestä.

Koodin testattavuus on tärkeä osa nykyaikaista ohjelmistoprojektia. Se vähentää huomattavasti testaukseen tarvittavia resursseja, sillä testien ei tarvitse olla niin monimutkaisia ja niillä saadaan suuremmalla todennäköisyydellä löydettyä virheet. Testattavalla koodilla on myös hyvä rakenne, joka parantaa ylipäätään koodin selkeyttä ja laatua. [Gao, 2002]

Poston ja muut [2011] jaottelevat testauksen kannalta vaadittavat ominaisuudet kolmeen eri osaan: hallittavuus, tarkkailtavuus ja yksinkertaisuus. Hallittavuudella he tarkoitettavat sitä, että ohjelmaan menevien syötteiden on oltava hallittavissa. Jos niitä ei voida kontrolloida, ei voida olla varmoja siitä, miten syötettä käsiteltiin. Tarkkailtavuus keskittyy ohjelman tulosteiden tarkasteluun. Testaajan on kyettävä näkemään, että ohjelman lopputulos on saavutettu oikein. Yksinkertaisuudella he puolestaan viittaavat minimaalisiin ja yksinkertaisiin toimintoihin yhtenäisellä ja vähän riippuvuuksia omaavalla koodilla.

Kuten jo aiemmin todettiin, saa koodista yksinkertaisempaa pilkkomalla sen pienempiin kokonaisuuksiin. Selkeyttämisen lisäksi se myös helpottaa testausta suuresti. Pienempiä osia on helpompi testata, ja testit ovat niin ikään pienempiä ja yksinkertaisempia [Goodliffe, 2006].

Googlen ketterien menetelmien valmentaja Hevery [2008] mainitsee myös muutaman tekniikan, joilla voi parantaa koodin testattavuutta. Esimerkiksi rakentajissa ei tulisi olla mitään ”oikeaa” toiminnallisuutta, kuten uusien objektien luontia, staattisia metodikutsuja, ehtolauseita tai silmukoita. Rakentajaan pitäisi jättää vain muuttujien arvojen asettaminen. Kaikenlainen ylimääräinen toiminta rakentajassa lisää riippuvuuksia ja vaikeuttaa suuresti testausta, sillä myös

riippuvuudet joudutaan luomaan tai simuloimaan rakentajaa testattaessa. Monesti tämä kierretään tekemällä erillinen rakentaja testausta varten, mutta tämä ei kuitenkaan toimi, sillä jossain vaiheessa testeissä kuitenkin tullaan törmäämään myös ”oikeaan” rakentajaan asiakaskoodin yhteydessä. Paras keino on ennaltaehkäisevä, eli jo koodatessa tulisi miettiä, onko rakentajaa helppo testata erillään muista vai tuleeko ongelmia vastaan. Objektien luonnit kannattaa hoitaa asiakaskoodin puolella tai käyttää erillistä tehdasluokkaa, jossa voidaan luoda usean luokan objektit kerralla ja antaa objektit rakentajille parametreina.

Liian pitkiä funktioketjutuksia tulisi niin ikään välttää. Jos tiettyä funktiota kutsuttaessa joudutaan hyödyntämään muitakin funktiokutsuja, aivan kuten kohdan 2.2.1 hedelmäkoriesimerkissä, kannattaa miettiä koodiin korjauksia. Aina tulisi kommunikoida vain lähimpien elementtien kanssa, eikä antaa tai velvoittaa asiakaskoodia kaivelemaan tietoa, vaan tarjota suoraan tarvittavat funktiot. Liialliset ketjutukset voi usein välttää myös antamalla mahdollisimman tarkkoja objekteja parametreina, eikä objekteja, jotka vain kantavat sisällään tarvittavaa tietoa. [Hevery, 2008]

3.1 Modulaarisuus

McConnell [2004] määrittelee modulaarisuuden ”mustaksi laatikoksi”, jonka sisälle emme välttämättä näe, mutta tiedämme, mitä sen pitäisi tehdä. Kutsumme modulia ja osaamme odottaa tiettyä vastausta, mutta emme tiedä, mitä kutsun ja vastauksen välillä tapahtuu, eikä meidän tarvitsekaan tietää. Moduli on itsenäinen osa järjestelmää, jolla on hyvin määritellyt rajapinnat. Se mahdollistaa esimerkiksi saman järjestelmän eri modulien kehittämisen samanaikaisesti eri ohjelmoijien toimesta ja helpottaa suurten kokonaisuuksien hallintaa. [Grenning 2011; Goodliffe, 2006; Beck and Diehl, 2011] Moduleista voidaan muodostaa myös uudelleenkäytettäviä komponentteja, joita voidaan hyväksikäyttää samassa järjestelmässä tai viedä jopa toiseen järjestelmään.

Dantas [2011] ja Kästner ja muut [2011] muistuttavat, että aina liika modulaarisuus ei kuitenkaan ole välttämättä hyväksi. Ohjelmoijat saattavat joutua käyttämään liikaa aikaa yhteyksiä ylläpitävän koodin luomiseen ja suunnittelemiseen. Voi olla myös hankala ymmärtää,

mihin kaikkialle moduliin tehtävät muutokset vaikuttavat. Modulaarisuuteen pyrittäessä on hyvä muistaa myös Conwayn laki: *”Jos organisaatio, joka järjestelmän suunnittelee, ei ole itsessään modulaarinen, niin ei järjestelmä itsekään tule sitä olemaan*” [Beck and Diehl, 2011].

Wong ja muut [2011] huomioivat, että modulaarisuuden rikkoutumista on hyvin vaikea havaita perinteisillä verifiointi- ja validointimenetelmillä. Ne eivät vaikuta suoranaisesti ohjelman toimintaan, vaan vaikeuttavat enemmänkin ohjelmoijan työtä tulevaisuudessa. Huonoon modulaarisuuteen voi johtaa esimerkiksi kiireessä tehdyt ratkaisut, tai jokin uusi vaatimus, joka ei taivu hyvin olemassa olevaan arkkitehtuurin. Tällöin voidaan joutua rikkomaan järjestelmän arkkitehtuuria tai tekemään kompromisseja. [Zazwork et al., 2013] Beckin ja Diehlin [2011] mukaan modulaarisuuden tärkein tekijä on periytyvyys ja kaksi tärkeintä ominaispiirrettä ovat yhtenäisyys ja riippumattomuus, joista kerron seuraavaksi.

3.1.1 Yhtenäisyys

Yhtenäisyydellä (eng. cohesion) tarkoitetaan sitä, että modulin sisällön tulisi olla mahdollisimman yhdenmukaista ja toimia tiiviisti yhdessä. Modulilla tulisi olla vain yksi toiminto tai toimintojen tulee liittyä kiinteästi toisiinsa [Harsu, 2003]. Moduliksi ei siis kelpaa esimerkiksi yleisesti käytetty ”utils”-paketti, johon laitetaan kaikki sekalainen, vaan modulilla tulee olla tietty rooli, jota jokainen sen osa edistää. [Goodliffe, 2006] Elementit ryhmitellään yhtenevien ominaisuuksien mukaan [Beck and Diehl, 2011].

Huonosta yhtenäisyydestä voi kertoa esimerkiksi se, että funktiot suorittavat liikaa toimintoja tai luokassa metodit tekevät täysin eri asioita. Huono yhtenäisyys modulissa johtaa yleensä riippuvuuksien luomiseen muihin moduleihin.

3.1.2 Riippumattomuus

Modulaarisuuden yhteydessä riippumattomuudella (eng. low coupling) tarkoitetaan yksinkertaisesti sitä, että modulit ovat mahdollisimman vähän riippuvaisia toisistaan [Goodliffe, 2006]. Modulien sisäinen viestintä on vapaampaa, ja niiden tulisikin olla riippuvaisia toisistaan,

mutta modulien väliset yhteydet tulisi jättää mahdollisimman vähäisiksi rajapintojen ulkopuolella [Beck and Diehl, 2011; Taube, 2011].

Riippuvuutta aiheuttavat esimerkiksi suorat funktiokutsut toiseen moduliin tai toisen modulin tietotyyppien käyttö [Goodliffe, 2006; Beck and Diehl, 2011]. Riippuvuutta voi aiheutua myös epäsuorasti, kuten jos luokat omaavat saman yliluokan tai toteuttavat saman rajapinnan. [Beck and Diehl, 2011] Hevery [2008] lisää myös, että singletonit, eli suunnittelumalli, jolla voidaan rajoittaa luokkaesiintymä vain yhteen, ja muut staattiset kutsut lisäävät riippuvuutta. Niiden kautta tietoa voidaan lukea ja muokata huomaamatta. Tämä voi aiheuttaa ongelmia testauksessa, jos aiempi testi on esimerkiksi asettanut muuttajan vääränlaiseen tilaan tai testien järjestys voi vaikuttaa lopputulokseen. Testejä ei myöskään voi ajaa yhtäaikaisesti. Liian suuret elementit ovat niin ikään riippuvuuksien aiheuttajia, joten koodin pilkkominen pienempiin osiin, niin modulien kuin funktioidenkin tasolla, usein parantaa riippumattomuutta [Taube, 2011].

Kannattaa olla myös varuillaan, mikäli huomaa muokkaavansa tiettyjä koodin osia aina yhdessä, osat käyttävät samaa koodia tai jopa, jos niillä on vain samat ohjelmoijat. Nämä ovat usein merkkejä riippuvuuksista tai johtavat riippuvuuksien luontiin modulien kesken. [Beck and Diehl, 2011] Taube [2011] huomioi myös elementtien iän ja riippuvuuden korrelaation. Mitä vanhempi elementti on, sitä suuremmalla todennäköisyydellä se on yhteydessä muihin elementteihin, sillä se on mitä luultavimmin hyödyllinen, tunnettu ja luotettu osa järjestelmää ja sitä hyödynnetään paljon ”maineensa” takia.

Riippuvuudet modulien välillä vaikeuttavat koodin muokkausta ja ymmärrystä sekä estävät eri modulien samanaikaisen kehityksen. Kun yhtä modulia muutetaan, joudutaan luultavasti muuttamaan myös siitä riippuvaisia moduleja. Riippumattomuudella pyritään siis ehkäisemään muutosten leviämistä modulin ulkopuolelle ja näin vakauttamaan järjestelmää. [Taube, 2011]

”Informaation piilotus” mainitaan usein modulaarisuuden ja riippumattomuuden yhteydessä. Termillä tarkoitetaan modulin toteutuksen ja rajapinnan erottelua [Ostermann, 2011]. Rajapinnalla pystytään tiivistämään isompi kokonaisuus helposti ymmärrettäväksi ja sen kautta hoidetaan kaikki kutsut moduleihin, minkä takia ei vaadita muutoksia asiakaskoodiin, mikäli modulin toteutusta muutetaan. Näin abstraktiotaso nousee ja asiakkaan ei tarvitse tietää itse toteutuksesta. Riittää, kun asiakaskoodi tietää, mitä tulisi tapahtua. [Ostermann, 2011; Goodliffe,

2006] Esimerkkinä voidaan käyttää listan järjestämistä. Järjestämisalgoritmi voidaan vaihtaa toiseen ilman, että asiakas tietää siitä mitään. Toteutuksella voi olla myös monta eri rajapintaa, jotta voidaan hallita modulin näkyvyyttä eri suuntiin. [Ostremann, 2011] Ostermann [2011] neuvoo hyvänä nyrkkisääntönä, että kannattaa arvioida koodia, ja etenkin ne osat, jotka tulevat todennäköisesti muuttumaan, kannattaa piilottaa rajapinnan taakse.

3.2 Testivetoinen kehitys

Testivetoinen kehitys (eng. test-driven development tai test-first coding) on yleistynyt kovaa vauhtia yritysten käytössä viimeisen vuosikymmenen aikana. Perinteisessä vesiputousmallissa ensin koodataan ja sitten testataan, kun taas testivetoisessa kehityksessä käännetään osat päälaelleen. Ensin kirjoitetaan mahdollisimman yksinkertainen testi matalan tason vaatimuksen perusteella, sitten testi ajetaan [Hammond and Umphress, 2012]. Tämän jälkeen vasta kirjoitetaan ohjelman koodi niin, että se juuri läpäisee testin. Viimeisenä vaiheena täytyy muistaa refaktoroida kirjoitettu koodi hyvien käytäntöjen mukaiseksi, ja sen jälkeen kirjoitetaan uusi testi. [Hammond and Umphress, 2012] Kehityksessä ei siis ole erillistä testausvaihetta, vaan testaus on keskeisessä osassa koko ajan. Uutta toiminnallisuutta ei lasketa toteutetuksi ennen kuin kaikki uudet ja jo aiemmilla ominaisuuksille tehdyt testit menevät läpi. [Dogša and Batič, 2011]

Testivetoinen kehitys tarjoaa monia parannuksia perinteisempään malliin verrattuna. Ensinnäkin se tekee koodin rakenteesta modulaarisempaa, sillä jokainen tehty osa täytyy olla testattavissa ennen kuin se hyväksytään ja näin riippuvuudet pienevät [Grenning, 2011]. Sen myötä myös koodin kompleksisuus vähenee huomattavasti ja laatu paranee ylipäätään, kun ohjelmoijat kirjoittavat vain sen verran koodia, että testi menee läpi. Näin monimutkaisia rakenteita ei pääse syntymään. Testivetoinen kehitys tarjoaa ikään kuin turvaverkon koodin muuttamiseen, jonka varassa on helppo tuoda uusia ominaisuuksia tai päivittää vanhoja. Regressiotestauksesta tulee huomattavasti helpompaa. Ohjelmiston ylläpito helpottuu niin ikään paljon ja siihen vaadittavat resurssit vähenevät. [Dogša and Batič, 2011]

Testivetoinen kehitys varmistaa kyllä, että koodi toimii matalalla tasolla, mutta joidenkin tutkimusten mukaan ongelmana siinä on, että se ei takaa korkeamman tasojen vaatimusten toteutumista tai järjestelmän suunnittelun virheettömyyttä. [Dogša and Batič, 2011; Hammond and Umphress, 2012] Se myös lisää koodaukseen vaadittavaa aikaa, kun joudutaan käyttämään paljon resursseja etukäteen testien miettimiseen ja valmisteluun. Ohjelmoijat voivat tuntea menetelmän myös vaikeaksi ja epämukavaksi toteuttaa, mikä sekin vähentää tuottavuutta. [Dogša and Batič, 2011]

Ohjelmiston koodin selkeys on tärkeä elementti ohjelmistokehityksessä, ja testien selkeys on lähes yhtä tärkeää, ellei jopa tärkeämpää. Sen taso tulisi olla ainakin yhtä hyvää kuin itse tuotteen koodi, ja siinä tulisi hyödyntää kaikkia samoja ominaisuuksia. Testien ei tulisi olla riippuvaisia toisistaan tai ympäristöstä. Hyvät testit tarjoavat suoraan informaation testien läpimenosta, eivätkä velvoita testaajaa tutkimaan testin lokeja. Jatkossa selkeys helpottaa testien päivittämistä huomattavasti. Testien sekavuus ja monimutkaisuus saattaa johtaa suurempiin ongelmiin myöhemmin, jos testien kirjoittaminen ja ylläpito käy liian vaivalloiseksi. [Martin, 2008]

4. Perinnekoodin refaktorointi

Ohjelmistokehityksessä on valtavat paineet saada ohjelmistoa tuotettua. Aikataulut ja vaatimukset kiristyvät entisestään kilpailun kasvaessa. Tekniikat kehittyvät niin huimaa vauhtia, että usein on mahdoton pysyä selvillä kaikesta uudesta. Kiire johtaa usein siihen, että ollaan liian lyhytnäköisiä tehtyjen ratkaisujen osalta ja hankaluudet ilmenevät vasta myöhemmin. Kriittisiin asioihin ei kiinnitetä tarpeeksi huomiota ja ratkaisuissa käytetään oikoteitä, mikä voi johtaa muun muassa perinnekoodin syntymiseen. Tässä luvussa kerron, mitä perinnekoodi on, miten se syntyy ja kuinka sitä tulisi refaktoroida.

4.1 Perinnekoodi

Perinnekoodille on olemassa monia eri määritelmiä. Yleensä se mielletään vanhentuneeksi lähdekoodiksi, jota ei enää ylläpidetä tai tuoteta. Esimerkiksi Pirkelbauer ja muut [2010] määrittelevät perinnekoodin sellaiseksi, joka on ohjelmointikielien ja tekniikoiden kehittymisen myötä jäänyt kehityksestä jälkeen. Muita yleisiä määritelmiä ovat muualta tai vanhasta järjestelmästä peritty lähdekoodi. Yksi suosituimmista moderneista määritelmistä tulee kuitenkin Feathersilta [2004], joka toteaa, ettei perinnekoodin välttämättä tarvitse olla edes vanhaa. Tälläkin hetkellä joku varmasti kirjoittaa perinnekoodia. Tärkein määritelmä hänen mukaansa perinnekoodille on, että sille ei ole testejä eikä se ole testattavaa. Tämän takia sitä on äärimmäisen vaikea lähteä muokkaamaan. Feathers [2004] toteaakin, että perinnekoodi itsessään ei ole paha asia, mutta jos sitä täytyy muuttaa, niin syntyy ongelmia.

Perinnekoodin syntymiseen voi olla useita syitä. Yksi yleisimmistä on vastuuhenkilöiden puute [Ritchie, 2010]. Jos vain yhden henkilön vastuulla on tietty osa järjestelmää ja hänen täytyy jättää kyseiset hommat, vaatii todella paljon joltain muulta opetella, kuinka se osa toimii ja miten sitä tulisi kehittää eteenpäin. Helposti voi käydä niin, ettei järjestelmän osa enää pysy kehityksessä muiden tahdissa tai kukaan ei oikeastaan halua enää koskea siihen, sillä sitä on vaikea muuttaa ja virheitä tulee helposti. Ritchie [2010] mainitsee myös ymmärtämättömyyden

yhtenä syynä perinnekoodiin. Ei yksinkertaisesti ymmärretä, miten asioiden pitäisi toimia suuressa kokonaisuudessa tai miksi jokin toimii niin kuin se toimii. Asiat toimivat kunnes saavutetaan kriittinen piste, jolloin ei enää tehdä muutoksia ns. korttitalon sortumisen pelossa. Grenning [2011] painottaakin, että vaikka järjestelmä olisi hyvin suunniteltu, se ei pysy hyvänä, jos ei käytetä resursseja koodin rapistumisen välttämiseksi. Jos päivitetään ominaisuuksia, mutta ei rakennetta, on ohjelmiston mädäntyminen väistämätöntä.

Tekninen velka on myös yleinen syy perinnekoodin syntymiseen. Ohjelmistokehittäjät voivat ottaa sitä tiukkojen aikataulujen pitämiseksi. Tekninen velka viittaa ajan lainaamiseen tulevaisuudesta, eli voidaan oikoa tiettyjä asioita sillä ehdolla, että oikomisen aiheuttamat viat korjataan myöhemmin [Zazwork et al., 2013]. Mikäli asiaan ei palata myöhemmin, kertyy teknistä velkaa entisen päälle. Koodiin voi syntyä ongelmallisia riippuvuuksia ja ylläpito hankaloituu, mikä voi myöhemmässä kehityksen vaiheessa johtaa suuriin ongelmiin tai jopa projektin epäonnistumiseen. Yleisimmin päätös ottaa teknistä velkaa ei tule ohjelmoijalta itseltään, vaan sidosryhmän jäseneltä, jolla ei ole teknistä ymmärrystä velan ottamisen seurauksista [Klinger et al., 2011]. Päätöksen tekijä on hyvin yleisesti korkeamman tason johtaja tai asiakas, jotka vastaavat tuotteen aikataulutuksesta ja resurssien käytöstä. Tekninen velka tulisi kuitata pois niin pian kuin mahdollista refaktoroinnin avulla.

4.2 Refaktorointi

Refaktorointi tarkoittaa lähdekoodin sisäisten rakenteen parantamista siten, että ohjelman ulkoinen toiminnallisuus ei muutu [Fowler et al., 1999]. Parantaminen voi olla koodin selkeyttämistä, rakenteen tai arkkitehtuurin muuttamista tai testien tekemistä. Usein termiä käytetään myös yleisesti koodia muutettaessa, sillä yleensä refaktorointia suoritetaan tehdessä muutoksia järjestelmän toimintaan [Kim et al., 2012].

Järjestelmän laajuista refaktorointia voi olla kolmea erilaista: yhtäkkinen refaktorointi, vähittäinen refaktorointi ja järjestelmän kehittymisen kautta tapahtuva refaktorointi. Yhtäkkisessä refaktoroinnissa järjestelmä uudistetaan kerralla. Mikäli joudutaan pitämään samaan aikaan vanhaa järjestelmää yllä ja tekemään sille uusia ominaisuuksia, kun siitä tehdään

refaktoroitua versiota, on järjestelmän yhtäkkinen uudistaminen työlästä, sillä muutokset joudutaan toteuttamaan myös uuteen järjestelmään. Vähittäisessä refaktoroinnissa järjestelmää muutetaan osa kerrallaan. Elementti voidaan ottaa refaktoroitavaksi aina, kun ilmaantuu tarvetta. Menetelmä on muuten hyvä, mutta se mahdollistaa vain yksittäisten osien refaktoroinnin, eikä anna uudistaa arkkitehtuurisia ratkaisuja kovinkaan helposti. Koko järjestelmän refaktorointi voi kestää myös todella pitkän aikaa. Järjestelmän kehittymisen kautta tapahtuva refaktorointi on samankaltainen edellä mainitun kanssa, mutta siinä eri järjestelmän osat ryhmitellään toiminnallisuuden perusteella, toisin kuin vähittäisessä refaktoroinnissa. Uuteen järjestelmään muodostetaan toiminnallisuuksiltaan samanlaisten osien kokonaisuuksia sitä mukaa kuin niitä tarvitaan. [Harsu, 2003]

Refaktoroinnista on suurta hyötyä ohjelmistoprojekteissa. Suurimmat hyödyt koostuvat ylläpidettävyyden, suorituskyvyn ja luettavuuden paranemisesta, modulaarisuuden lisääntymisestä sekä virheiden vähenemisestä kehityksen aikana ja jälkeen. Refaktorointi voi monesti myös alentaa kehittämisen kuluja pitkällä aikavälillä. [Kim et al., 2012] On yleistä, että järjestelmän arkkitehtuuri rapistuu ajan myötä, kun siihen lisätään muutoksia. Refaktorointi on rapistumisen vastakohta. Sen avulla voidaan tehokkaasti pelastaa vanhentuneesta tai huonosti suunnitellusta järjestelmästä helposti ylläpidettävä kokonaisuus [Fowler et al., 1999]. Myös vaatimukset voivat aina muuttua, oli järjestelmä kuinka vanha tai hyvin suunniteltu tahansa, ja refaktoroinnilla mahdollistetaan juuri vaatimusten aiheuttamat muutokset [Grenning, 2011].

Refaktoroinnista ei saa mitään välitöntä etua, jota asiakkaat tai rahoittajat osaisivat arvostaa. Sen takia eri osapuolet on vaikea saada innostumaan resurssien käytöstä refaktorointiin. Tämä voi johtaa liian vähäiseen resurssien varaamiseen, jolloin refaktorointia joudutaan tekemään kaiken muun ohella [Harsu, 2003]. Silloin refaktorointiprosessi voi venyä niinkin pitkäksi, että kun se saadaan valmiiksi, on järjestelmä jo uudelleen refaktoroinnin tarpeessa [Harsu, 2003].

Laajat järjestelmät, joissa on suuret määrät lähdekoodia, ovat vaikeita refaktoroida. Mitä enemmän koodia järjestelmästä löytyy, sitä suuremmalla todennäköisyydellä sinne on syntynyt paljon riippuvuuksia. Refaktorointi vaatii myös hyvää koordinointia eri kehittäjien ja tiimien välillä, etenkin jos muutokset vaikuttavat koko järjestelmän alueella. Puutteellinen regressiotestaus tuo myös oman haasteensa, ja sen myötä refaktoroinnin jälkeisen toiminnan varmistaminen voi käydä hankalaksi. Jos ei ymmärretä tiettyjä rajatapauksia, joille on voitu

tehdä poikkeuksia koodiin, saattavat ne helposti jäädä pois refaktoroidusta koodista. [Kim et al., 2012] Fowler ja muut [1999] huomioivat myös, että suunnittelumallien muutokset, tietokantojen kanssa toimivan koodin tai itse tietokannan refaktorointi voi vaatia suuria muutoksia koko järjestelmässä. Esimerkiksi, jos halutaan parantaa ohjelman tietoturvaa, voi se vaatia todella paljon muutoksia. Fowler ja muut [1999] muistuttavatkin, että kannattaa aina arvioida, onko refaktorointi vai koodin kokonaan uudelleen kirjoittaminen tehokkaampaa.

Refaktoroidun koodin yhdistäminen järjestelmään voi olla hankalaa [Kim et al., 2012]. Usein versionhallintajärjestelmät tukevat huonosti refaktoroinnin aikaansaamia muutoksia ja ohjelmoijan on itse ratkottava aiheutuneet konfliktit. Järjestelmissä, joissa täytyy muistaa pitää yllä yhteensopivuutta taaksepäin, refaktoronti on niin ikään työlästä, sillä muutoksien täytyy toimia kaikilla versioilla, jotka ovat käytössä. [Kim et al., 2012] Fowler ja muut [1999] mainitsevat tästä esimerkkinä rajapintojen muokkauksen, mikäli itsellä ei ole pääsyä kaikkeen rajapintaa käyttävään koodiin. Esimerkiksi julkisissa kirjastoissa rajapinnan muutokset vaativat taaksepäin yhteensopivuutta, joten vanhaa rajapintaa on tarjottava niin kauan, että kaikki asiakaskoodi on saanut mahdollisuuden päivittyä uuteen. Tämä tosin onnistuu parhaiten kutsumalla vanhasta rajapinnasta uuden rajapinnan funktioita.

Refaktorointia tulisi suorittaa myös koko ohjelmiston kehityksen ajan. Aina koodin kirjoittamisen jälkeen on hyvä reflektoida omaa tuotosta ja siistiä se ymmärrettävämmäksi ja modulaarisemmaksi [Martin, 2008]. Etenkin lisättäessä funktioita tai korjattaessa virheitä järjestelmässä refaktorointi on olennaisessa osassa [Fowler et al., 1999]. Näin siitä tulee rutiinia, eikä sitä tule sivuutettua helposti.

Refaktoroinnin aloittaminen kannattaa aloittaa Osheroven [2009] mukaan koodin kompleksisimmista osista, sillä ne kaipaavat refaktorointia eniten. Jos refaktoroijat eivät ole kokeneita yksikkötestaajia, on parempi aloittaa osista, joissa esiintyy vähemmän riippuvuuksia, jotta tekniikoihin on helpompi päästä sisään. Tällöin kuitenkin kaikkein vaikeimmat kohdat jäävät odottamaan omaa vuoroaan ja voivat vain pahentua ajan myötä [Osherove, 2009]. Loppua kohden projekteissa usein kiire lisääntyy ja viimeinen asia mitä tällöin toivoo, on vaikeimpien kohtien refaktorointi. Eli jos refaktorointitiimissä on paljon kokemusta yksikkötestauksen haasteista, kannattaa hyökätä suoraan monimutkaisimman ja eniten riippuvuuksia sisältävän koodin kimppuun.

Alkuun oikeaoppinen refaktorointi voi tuntua kankealta ja hitaalta, mutta rutinoitumisen myötä oppii käyttämään heti oikeita menetelmiä ja muokkaus onnistuu tehokkaammin. [Feathers, 2004]. Seuraavaksi kerron, kuinka refaktorointia kannattaa suorittaa perinnejärjestelmään, jossa ei ole yksikkötestejä saatavilla.

4.2.1 Haiseva koodi

Haisevalla koodilla (eng. code smell) tarkoitetaan kohtia lähdekoodissa, jotka kielivät virheistä tai rappeutuneesta koodista [Counsell et al., 2010]. Haiseva koodi ei kehity tasaisesti muun järjestelmän kanssa, sillä se vaatii ylimääräistä huomiota [Ratzinger et al., 2005]. Hajut voivat johtaa suuriinkin ylläpidollisiin ongelmiin ja näin viedä turhia resursseja, mikäli niihin ei puututa [Counsell et al., 2010]. Ohjelmoija tarvitsee hyvän käsityksen ja paljon kokemusta hyvän ja huonon koodin eroista, jotta hän voi huomata pienetkin virheet koodin seasta ja hankkiutua niistä eroon ilman toiminnan rikkoutumista. [Grenning, 2011]

Hajujen haasteellisuus on subjektiivista; ne vaihtelevat monesti ohjelmoijan mukaan. Joillekin tietyt kohdat ovat vaikeampia löytää ja ratkoa kuin toisille. [Merlton and Tempero, 2006] Yleisin syy refaktoroinnille on toistuva koodi. Sitä syntyy varsin helposti, mutta se on myös suhteellisen helppo huomata ja korjata. Muita yleisiä hajuja ovat huonosti nimetty, monimutkainen tai paljon riippuvuuksia sisältävä koodi. Myös pitkät rutiinit ja silmukat ovat tavanomaisia, niin ikään huonosti yhtenäiset luokat. [Grenning, 2011; McConnell, 2004] Jotkin helposti tunnistettavat hajut, kuten pitkät parametrilistat sekä suuret luokat ja metodit, ovat kuitenkin yllättäen kaikista työläimpiä refaktoroida. Vähemmän tunnetut, kuten rinnakkaisperintä ja hierarkiat, eivät puolestaan vaadi kovinkaan paljoa resursseja korjaukseen. [Counsell et al., 2010]

Hajuja voi kaivaa muualtakin kuin vain lähdekoodia tutkimalla. Ratzinger ja muut [2005] esimerkiksi analysoivat versionhallinnan historiaa ja löysivät sitä kautta monia vihjeitä riippuvuuksista. Merkkejä riippuvuuksista ovat muun muassa tilanteet, joissa kahta luokkaa muutetaan aina yhtä aikaa tai jos luokan muuttaminen vaatii joka kerralla monen muun luokan muuttamista eri moduleissa [Ratzinger et al., 2005].

4.2.2 Riippuvuuksien rikkominen

Refaktoroimisen aluksi on etsittävä kohta, jota halutaan refaktoroida, eli muutospiste. Perinnekoodia ei voida mennä kuitenkaan muuttamaan noin vain, etenkään silloin, jos järjestelmästä ollaan vielä riippuvaisia [Feathers, 2004]. Perinnekoodin refaktoroinnin yksi suurimmista haasteista onkin siinä, että sille ei ole olemassa yksikkötestejä ja niiden kirjoittaminen on vaikeaa, mutta yksikkötestit ovat kuitenkin refaktoroinnin kannalta välttämättömiä. Tämän ongelman ratkaisuun tulee löytää järjestelmän koodista vaikutuspiste (eng. inflection point), jonka kautta näemme, jos refaktoroitavan osan käyttäytyminen muuttuu. Vaikutuspisteen tulisi olla siis mahdollisimman ohut rajapinta refaktoroitavaan koodiin, josta mahdollisimman pienellä vaivalla voidaan varmistaa toiminnan muuttumattomuus. Pisteiden löytämiseen ei tulisi käyttää valmiita kuvauksia järjestelmästä, sillä ne voivat olla vanhentuneita tai harhaanjohtavia. [Feathers, 2004]

Vaikutuspisteen muodostamat riippuvuudet voivat muodostaa pitkiäkin ketjuja tai jopa kehiä. Huonon koodin kanssa on helppo joutua riippuvuuskierteeseen [Feathers, 2004]. Jos halutaan testata yhtä elementtiä kehällä, on sitä mahdotonta ymmärtää tai testata ilman muiden elementtien tutkimista tai hyväksikäyttöä. Riippuvuuksien rikkomisella pyritään eristämään vaikutuspiste muista, jotta elementistä saataisiin ymmärrettävämpi ja hallittavampi kokonaisuus. [Melton and Tempero, 2006]

Rikkominen tarkoittaa yleensä vaikutuspisteen muokkausta tai erottamista rajapinnalla. Ulkoisissa riippuvuuksissa hyödynnetään muita objekteja tai moduleita. Esimerkkinä mainittakoon toisen objektin antaminen parametrina rakentajalle. Tällaisten riippuvuuksien rikkominen käy helpoiten rajapinnan avulla. Rajapinta lisätään testattavan elementin ja sen riippuvuuden välille. Näin testissä voidaan helposti luoda tyhjä instanssi riippuvuudesta, joka toteuttaa rajapinnan, mutta ei tarjoa toiminnallisuutta. Itse elementin, josta ollaan riippuvaisia, toteutusta ei siis tarvitse muuttaa juuri laisinkaan. [Feathers, 2004]

Sisäiset riippuvuudet puolestaan käyttävät elementin sisäisiä rakenteita. Ne ovat yleensä hieman hankalampia käsitellä ja niissä toteutusta joudutaan monesti muuttamaan, jotta testauksessa päästäisiin eteenpäin. Yleensä riippuvuuden aiheuttama koodi kannattaa eristää omaan funktioonsa ja kuormittaa omalla toteutuksella. [Feathers, 2004] Täytyy kuitenkin

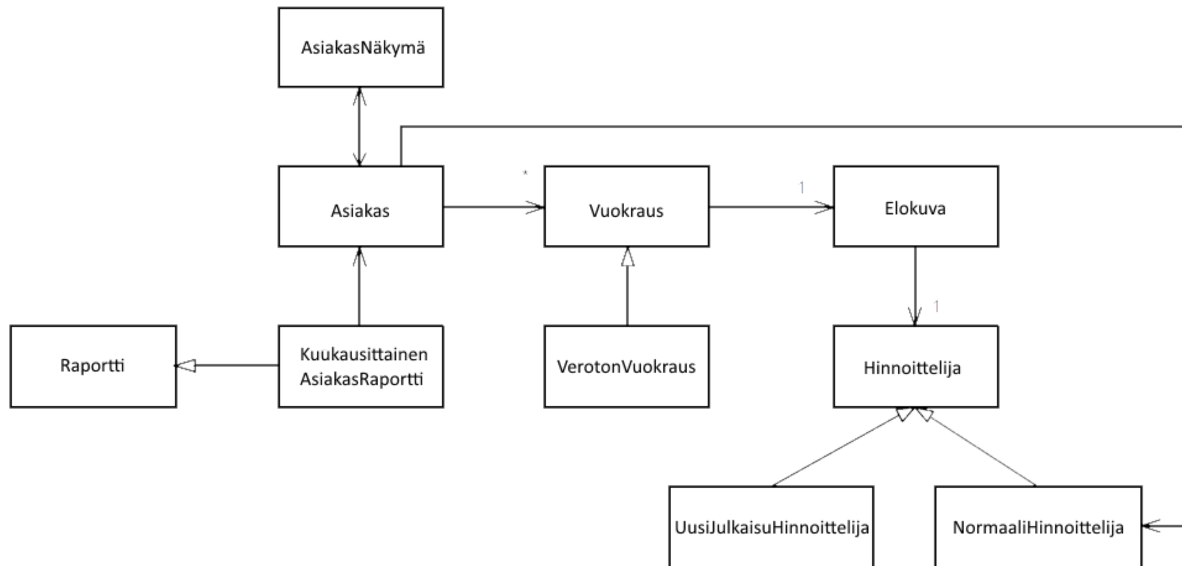
muistaa, että toiminta ei saa testattavassa elementissä muuttua, vaikka toteutusta muokattaisiinkin testattavammaksi.

Riippuvuudet eivät aina ole helposti nähtäviä fyysisiä riippuvuuksia. Ei riitä, että koodista poistaa näkyvät riippuvuudet. Riippuvuudet voivat muodostua esimerkiksi globaaleista muuttujista. Ajonaikana globaali muuttuja voi olla erilaisissa tiloissa ja se pitäisi huomioida myös testeissä [Feathers, 2004]. Esimerkiksi, jos testejä ajetaan monta peräkkäin, testien järjestys voi vaikuttaa niiden lopputuloksiin. Muuttujat tulee siis määritellä tarkasti ennen jokaista testiä, jotta testien ympäristö on jokaisella ajokerralla samanlainen ja testien lopputulokseen vaikuttaa vain testattava koodi.

4.2.3 Toiminnallisuuden testaus

Vaikutuspisteelle tulisi kirjoittaa varmistavia testejä (eng. characterization test), jotka toimivat turvaverkkona ja ilmoittavat, jos järjestelmän toiminta muuttuu refaktoroidessa [Fowler et al., 1999]. Testien ei siis pidä testata sitä, mitä järjestelmän pitäisi tehdä, vaan sitä, miten järjestelmä on toiminut aiemmin [Feathers, 2004]. Vaikka muutos olisi kuinka yksinkertainen, niin aina on mahdollista, että sen myötä järjestelmään ilmestyy uusi virhe. Jälkikäteen huomattua muutosta tai virhettä on luultavasti selvästi haastavampi jäljittää takaisin refaktorointikohtaan ja sillä voi olla suuriakin seurauksia.

Esimerkiksi jos haluaisimme tehdä muutoksia videon hinnoitteluun liittyvään toimintaan (kts. kuva 1), voisimme tehdä varmistavat testit Asiakas-luokalle. Sen kautta kulkee kaikki tieto, joka vuokraukseen liittyy, joten jos toiminnassa tapahtuu muutoksia Vuokraus-, Elokuva-, tai Hinnoittelija-luokissa, voidaan olla melko varmoja, että muutokset näkyvät myös Asiakas-luokassa.



Kuva 1. Videovuokraamon rakenne [Feathers, 2004].

Pisteille kirjoitetaan testejä, joilla voidaan varmistua siitä, ettei mikään niiden kautta kulkeva tai sen käsittelemä tieto muutu. Testatessa täytyy myös muistaa, että testattavan koodin on toimittava samalla tavalla kuin itse julkaistussa järjestelmässä. Ei voida siis tehdä oikopolkuja, jotta testauksesta saataisiin helpompaa, vaan koodi on saatava toimimaan sen omassa ympäristössä. [Feathers, 2004]

Hyviä testejä ovat esimerkiksi raja-arvojen testaukset. Pienten muutosten tekeminen muutospisteeseen voi antaa myös hyviä ideoita, mitä kannattaa testata. [Feathers, 2004] Testien tulisi antaa selkeää informaatiota. Joko testi menee läpi ja ilmaisee, että kaikki kunnossa, tai vastaavasti listaa, mitä eroavaisuuksia tai virheitä ilmeni [Fowler et al., 1999]. Varmistavia testejä tulee ajaa jatkuvasti refaktoroidessa, jotta huomataan välittömästi, mikäli käyttäytyminen muuttuu [Feathers, 2004].

4.2.4 Koodin muokkaus

On tärkeää muistaa tehdä muutoksia järjestelmään yksi kerrallaan, sillä liian isojen kokonaisuuksien korjaaminen voi muuten osoittautua ylivoimaiseksi. Kun tehdään paljon pieniä muutoksia, niin ajan mittaan ne kattavat koko järjestelmän. [Fowler et al., 1999] Luvuissa 2 ja 3

on esitelty ominaisuuksia, joiden pohjalta on hyvä lähteä muokkaamaan koodia parempaan suuntaan.

Ennen koodin muokkaamista on hyvä kirjoittaa muokattavalle koodille yksikkötesti, aivan kuten testivetoisessa kehityksessä. Sen jälkeen koodia voi muokata niin, että testi saadaan ajetuksi. Refaktoroinnin edetessä tulisi kirjoittaa koko ajan uusia yksikkötestejä refaktoroitavalle koodille, jotta varmasti katetaan kaikki ongelmakohdat ja esitellään mahdollisimman vähän uusia virheitä. [Feathers, 2004]

Helppoja muutoksia, joista voi lähteä liikkeelle, ovat esimerkiksi vakioiden luominen ja informatiivisempi nimeäminen. Tällaiset muutokset voivat tuntua mitättömiltä, mutta niillä on suuret vaikutukset koodin luettavuuteen ja ymmärrettävyyteen, ja näin ollen ylläpidettävyyteen jatkossa. Funktioiden ja luokkien eristäminen on myös varsin yksinkertainen toimenpide, jolla voidaan pilkkoa suurempia funktioita pienempiin osiin [McConnell, 2004; Ratzinger et al., 2005]. Esimerkiksi silmukoiden sisällön siirtäminen erilliseen funktioon selkeyttää paljon ja tarjoaa mahdollisuuden koodin uudelleen käyttöön. Eristämistä kannattaa suorittaa iteratiivisesti, jolloin voidaan huomata funktioiden välisiä riippuvuuksia ja esimerkiksi erottaa osasta uusia luokkia [Feathers, 2004].

Haastavimmasta päästä ovat varmastikin jo alakohdassa 4.2.2 mainitut riippuvuudet ja niistä eroon pääseminen. Toki tällä kertaa niiden muokkaaminen on hieman helpompaa, sillä onhan jo olemassa varmistavat testit vaikutuspisteessä, joiden varassa voimme tehdä muutoksia, mikäli vain riippuvuudet eivät ulotu vaikutuspisteen ulkopuolelle. Toistuva koodi on puolestaan yleisin muutoksen kohde [McConnell, 2004]. Uudelleen käyttämällä koodia voidaan pienentää lähdekoodin määrää huomattavasti ja vähentää tarvittavia koodimuutoksia jatkossa.

4.3 Automatisoitu refaktorointi

Automatisoitu refaktorointi on yleensä tehokkaampaa ja vähemmän virheitä tuottavaa kuin manuaalisesti suoritettu refaktorointi [Schäfer, 2013]. Työkaluja täytyy kuitenkin kehittää erikseen eri kielille, jolloin vähemmän suositut kielet jäävät väkisinkin taka-alalle. Suosituille kielille, ja etenkin oliopohjaisille, kuten Java, työkaluja löytyy todella runsaasti.

Ohjelmointiympäristöt, eli IDE:t (Integrated Development Environment), ovat yleistyneet ja kehittyneet hurjasti. Niiden avulla voidaan hoitaa esimerkiksi uudelleennimeämiset, kapselointi ja koodin rakenteen muokkaus automaattisesti. Niistä on tullut arkipäivää osalle ohjelmistokehittäjistä ja ne tekevät koodin hallinnasta huomattavasti helpompaa. Murphy-Hill ja muut [2009] kuitenkin osoittivat, että työkaluja käytetään yllättävän vähän. Syynä tähän he esittävät riittämättömät käyttöliittymät työkaluille. Yleisiä ohjelmointiympäristöjä ovat muun muassa Eclipse, Netbeans ja Visual Studio. Niihin on mahdollista luoda omia liitännäisiä, joilla voi parannella jo olemassa olevia ominaisuuksia. Esimerkiksi Zibran ja Roy [2011] kehittivät Eclipselle liitännäisen toistuvan koodin etsimiseen ja osittaiseen refaktorointiin.

Ohjelmistoympäristöistä riippumattomiakin työkaluja on tutkittu ja kehitetty. Esimerkiksi Munro [2005] rakensi prototyypin koodin hajujen automatisoituun paikantamiseen. Bavota ja muut [2013] puolestaan esittelivät työkalun, joka osaa automaattisesti jakaa huonosti rakennetun luokan kahteen tai useampaan yhtenäisempään luokkaan. Aikaisemmat tällaiset työkalut ovat kasvattaneet luokkien riippuvuuksia, mutta Bavota ja muut [2013] onnistuivat välttämään riippuvuuksien kasvamisen. Työkalu ei osaa itse kuitenkaan etsiä refaktorointia kaipaavaa luokkaa, vaan sille pitää erikseen osoittaa, mikä luokka sitä tarvitsee. Täysin automatisoituja työkalujakin toki on olemassa. Tällöin säästytään siltä, että refaktoroijan täytyy tuntea refaktoroitava koodi ja tietää, kuinka ja milloin refaktoroida. Griffith ja muut [2011] kehittelivät automatisoidun järjestelmän, joka etsii haisevaa koodia ja pyrkii korjaamaan niissä esiintyvät puutteet tai viat. Myös varmistavien testien tekemiseen voidaan käyttää automatisoituja työkaluja. Esimerkiksi BlackBoxRecorder-työkalu osaa automaattisesti generoida varmistavia testejä merkatuille metodeille .NET perinnekoodista [BlackBoxRecorder, 2014].

5. Tapaustutkimus Nokia: Konfiguraationhallinnan analysointi ja refaktorointi

Nokia on yksi maailman johtavista tietoliikenneverkkojen laitteiden ja ohjelmistojen tarjoajista. Nokian tietoliikenneverkkojen toiminnasta vastaava Networks toimii yli 150 maassa. Tuotteisiin kuuluvat radioverkoissa toimivat elementit ja niiden ohjelmistot.

Tässä tapaustutkimuksessa tutkin Networksin Radio Network Controller -tuotteen konfiguraationhallintaan liittyvien komponenttien ohjelmistoarkkitehtuuria. Ensin esittelen tutkimuksen suorittamista ja konfiguraationhallinnan ympäristöä, sitten itse konfiguraationhallintaskenaarion ja motiivin työlle. Lopuksi esittelen tulokset ja tulosten perusteella laaditut parannusehdotukset.

5.1 Tutkimuksen suorittaminen

Tutkittava sovellusalue oli hyvin laaja ja sisälsi lukuisia toiminnallisuuksia. Koska järjestelmä on edelleen tuotekehitysvaiheessa, ei dokumentaatio vastaa täysin itse ohjelmistoa. Dokumentaation lisäksi kävin läpi itse toteutusta sekä konsultoin kokeneempia työntekijöitä. Hyödyllisimmäksi tutkimusmateriaaliksi toiminnallisuuden ymmärtämisessä osoittautuivat skenaarioita kuvaavat sekvenssikaaviot. Järjestelmässä esiintyvistä epäkohdista sai niin ikään paljon vihjeitä ryhmän ohjelmistosuunnittelijoilta. Virallista haastattelutilaisuutta ei koskaan järjestetty, vaan tilaisuudet olivat vapaamuotoisempia, ja näin keskustelu vilkkaampaa. Parannusideoita sai tehokkaasti myös itse kokeilemalla, minkälaisiin ongelmiin refaktoroinnissa tulee törmäämään.

Analysoin koodia saadakseni ymmärryksen järjestelmästä, sen käyttämän koodin syntaksista ja järjestelmän sille asettamista rajoituksista. Analysoinnilla pyrin myös saamaan tarpeeksi hyvän käsityksen koodista, jotta pystyin arvioimaan koodia ja arkkitehtuuria kriittisesti sekä tuottamaan itse parannusehdotuksia.

Saatuani käsityksen koodista tavoitteena oli löytää refaktorointiin sopivia kohteita. Kohdan tuli olla sopivan yleinen, jotta käytettyjä tekniikoita voitaisiin soveltaa myös muualle järjestelmään. Löydetyistä vaihtoehdoista valitsin sopivan, jolle suunnittelin itse parannusehdotuksia. Tavoitteena oli antaa esimerkkejä ja mahdollisia refaktorointiehdotuksia. Tukena tekniikoiden toteuttamisessa olivat muut ohjelmistosuunnittelijat. Koodin täydellistä toimivuutta en pystynyt testeillä varmistamaan, sillä yksikkötestejä ei ole vielä otettu koodille käyttöön.

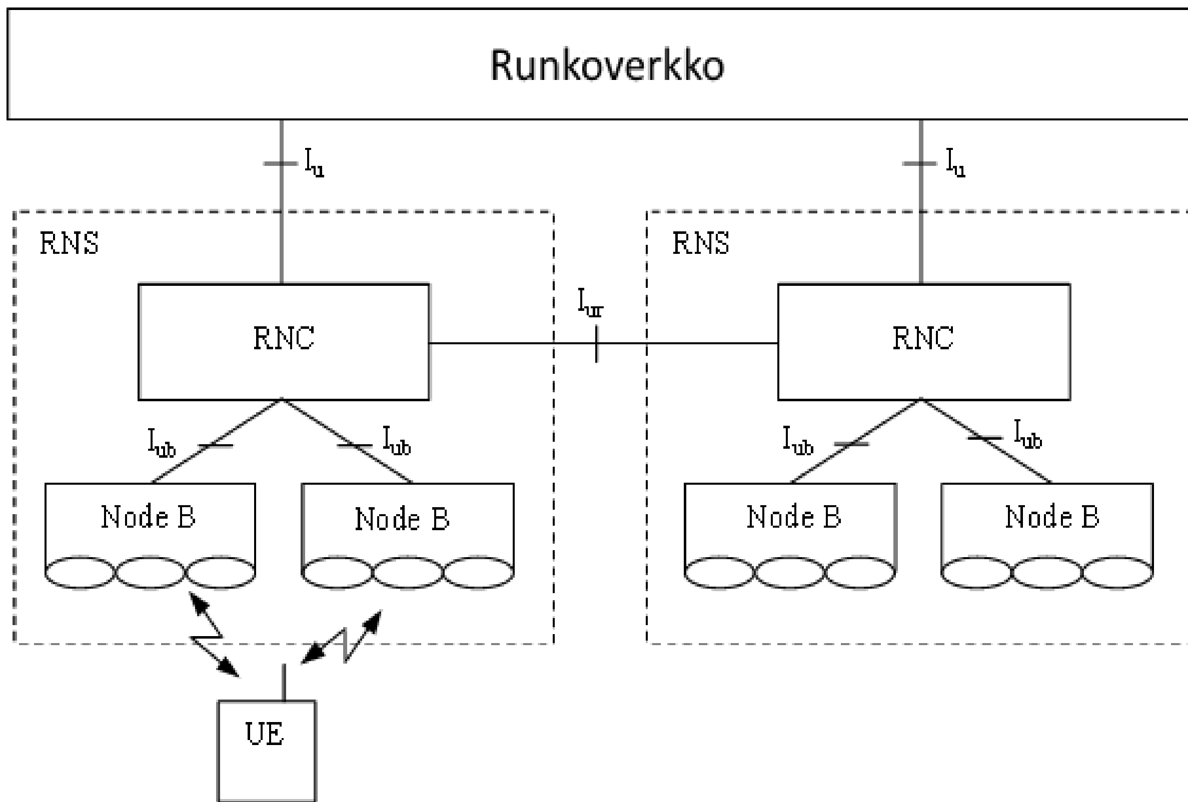
5.2 Ympäristö

Telekommunikaatiojärjestelmien kehityksessä vallitseva jatkuva hintakilpailu, nopeutuneet kehityssyklit, järjestelmien avoimuuden vaatimukset ja asiakkaiden vaatimukset luotettavasta ja vikasietoisesta järjestelmästä asettavat suuria haasteita ohjelmistolle ja niiden kehitykselle [Lindqvist et al., 2013]. Itse refaktoroitava järjestelmä on erittäin laaja, hajautettu ja sulautettu, reaaliaikainen käyttöjärjestelmä. Siinä on lukuisia eri komponentteja, joilla jokaisella on useita kymmeniä- tai satojatuhansia rivejä koodia. Tällaisen järjestelmän ylläpito vaatii paljon resursseja, jotta vältyttäisiin järjestelmän arkkitehtuurin taantumiselta. Järjestelmää kehitetään useassa eri maassa ympäri maapalloa. Kulttuurilliset erot, alihankkijoiden hyödyntäminen ja aikaerot tuovat omat haasteensa yhteistyöhön ja kommunikointiin.

5.2.1 Radio Network Controller

Radio Network Controller (RNC) on 3G-verkossa (WCDMA) toimiva elementti, joka on vastuussa UTRAN-verkon (UMTS Terrestial Radio Access Network) resursseista. UTRAN-verkko on itse käyttäjien laitteiden (UE, user equipment) ja runkoverkon välissä oleva verkko. Se koostuu Radio Network Subsystemeistä (RNS), jotka ovat yhteydessä runkoverkkoon [3GPP, 1999]. RNS muodostuu yhdestä RNC:stä ja sen tukiasemista. Tukiasemilla voi olla yksi tai useampi solu, jotka vastaanottavat ja lähettävät tietoa. RNC:llä on useita rajapintoja, joista kuvassa 2 on esitelty kolme tärkeintä. [3GPP, 1999] RNC:n päätehtäviin kuuluvat kuorman jakaminen tukiasemiin (I_{ub}), runkoverkkoon (I_u), ja myös muiden RNC:idenkin kesken (I_{ur}). Se

hoitaa myös tukiasemien (Node B) konfiguroinnin, kuten niiden suuntaamisen ja toimintasäteen valitsemisen. RNC:n vastuulla on niin ikään verkossa olevat salaukset. Käyttäjien laitteista tuleva tieto kryptataan ja runkoverkosta tuleva tieto dekryptataan.



Kuva 2. UTRAN-verkon arkkitehtuuri [3GPP, 1999]

Fyysisesti RNC koostuu useasta erillisestä pistoyksiköstä, joilla on omat tehtävänsä. Pistoyksiköitä ovat esimerkiksi ICSU (Interface Control and Signaling Unit), MXU (Multiplexer Unit) ja OMU (Operation and Maintenance Unit), jonka sisälle konfiguraationhallintakin sijoittuu. Yksikköjen lisääminen ja poistaminen on hyvin helppoa.

RNC:n ohjelmisto on hyvin vikasietoista. Mikäli virheitä pääsee järjestelmään, ne voivat johtaa tiedon, kuten puhelujen ja tekstiviestien, häviämiseen. Suorituskyky on niin ikään erittäin oleellinen seikka RNC:ssa. Etenkin mobiilidatan käytön kasvamisen seurauksena tullut kuorma aiheuttaa suuria haasteita. On kyettävä hallitsemaan ja jakamaan suuria määriä tietoa ilman merkittäviä viiveitä.

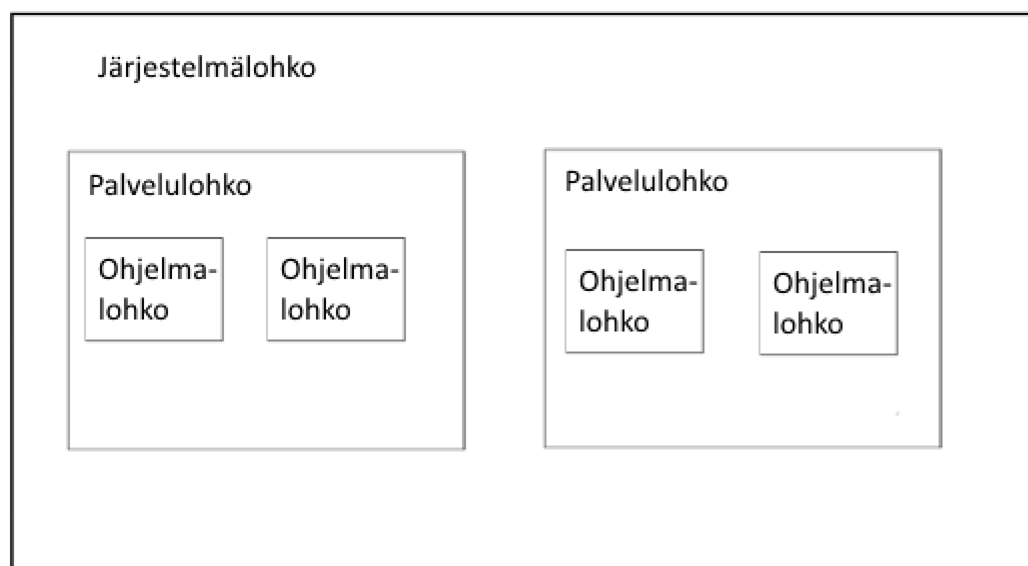
5.2.2 Ohjelmistoarkkitehtuuri

Tilakoneet mahdollistavat luotettavien ja asynkronisten järjestelmien rakentamisen. Esimerkiksi monet sulautetut järjestelmät, kuten ohjausjärjestelmät, hyödyntävät usein tilakoneita. Tilakone vastaanottaa syötteitä, joiden käsittely vaihtelee itse syötteen ja tilakoneen sen hetkisen tilan mukaan. Tila puolestaan määräytyy tulleista syötteistä ja tilakoneen edellisestä tilasta. Tilan vaihtuessa tilakone voi lähettää signaalin muualle, kuten toiselle tilakoneelle. Tilakoneen tulisi olla mahdollisimman pieni ja yksinkertainen. Tilakoneen määrittely piirrettynä graafiksi tulisi mahtua A4-paperille [Wagner et al., 2006]. Järjestelmä muodostuu näin ollen useista erillisistä tilakoneista. Tutkittu järjestelmä on toteutettu tilakoneita hyödyntäen. Järjestelmässä tilakoneet voivat muodostua kymmenistä eri tiloista ja signaaleista, jolloin Wagnerin ja muiden [2006] esittämä rajaus ylittyy huomattavasti.

Tutkitussa järjestelmässä käytetyt kielet ovat C ja Telenokia Specification and Description Language eli TNSDL. TNSDL on Nokian itse 1980-luvulla kehittelemä kieli, joka soveltuu asynkronisten, hajautettujen ohjelmien kehitykseen [Lindqvist et al., 2013]. Se pohjautuu SDL:ään, joka puolestaan perustuu muokattuun tilakoneeseen. Erona SDL:ssä perinteiseen tilakoneeseen on, että siinä voidaan lähettää useita eri signaaleja tilamuutoksissa sekä asettaa arvoja yleisiin muuttujiin ja tehdä päätöksiä myös niihin perustuen. SDL:ssä ei myöskään vaadita, että prosesseilla on jokin lopullinen tila, vaan päättymättömät prosessit ovat hyvinkin yleisiä. SDL koostuu järjestelmästä, jonka sisällä on useita lohkoja ja lohkojen sisällä prosesseja [Bræk, 1996].

TNSDL on suunniteltu erityisesti telekommunikaatiojärjestelmiä varten. Se koostuu kahdesta erillisestä osasta: määrittelevästä ja toiminnallisesta TNSDL:stä. Ensiksi mainitulla voidaan jakaa järjestelmä komponentteihin. Sillä määritellään suunnitteluvaiheessa korkean tason näkymä siitä, mitä järjestelmän pitäisi tehdä ja miten toiminnot järjestelmässä järjestetään. Sen avulla voidaan kuvata myös komponentista ulospäin näkyvät toiminnot eli esittää rajapinta. Määrittelevän TNSDL:n avulla voidaan niin ikään määritellä tietorakenteet, joiden avulla käsitellään ajonaikaista tietoa. Sillä ei voida kuitenkaan luoda mitään oikeaa toiminnallisuutta. Toiminnallisella TNSDL:llä puolestaan toteutetaan nimensä mukaisesti järjestelmän toiminnallisuus. Sitä käytetään vain prosessien tasolla ja se toteuttaa sen, minkä määrittelevä TNSDL kuvaa. TNSDL käännetään ennen käyttöönottoa C-kieleksi. [Lindqvist et al., 2013]

Ohjelmistoarkkitehtuuri jakautuu moneen erilliseen hierarkiatasoon eli lohkoihin (kuva 3). Korkeimmalla tasolla arkkitehtuuri koostuu järjestelmälohkoista, joista kukin sisältää useita palvelulohkoja. Järjestelmälohkot on muodostettu niiden sisältämien palvelulohkojen tarjoamien palveluiden mukaan. Arkkitehtuuri määräytyy SDL:n määrittelyn mukaan, josta myös lohkot tulevat. Järjestelmälohkot kommunikoivat keskenään viestien avulla ja tarjoavat toisilleen vain omat rajapintansa. Itse toteutusta järjestelmälohkot eivät näe. Palvelulohkot muodostuvat ohjelmalohkoista, jotka toteuttavat tietyn toiminnallisen kokonaisuuden. Vasta ohjelmalohkoista löydämme ohjelman varsinaisen toiminnallisuuden toteutuksen.



Kuva 3. Ohjelmistoarkkitehtuurin rakenne.

Ohjelmalohkoja voi olla kahdenlaisia: kirjasto- tai prosessilohkoja. Yhdessä ne muodostavat koodimodulin. Kirjastolohkot tarjoavat vain synkronoituja moduleita. Prosessilohkot puolestaan toteuttavat asynkroniset palvelut. Yhdessä prosessilohkossa on yksi prosessiperhe, joka koostuu yhdestä isäntäprosessista sekä renkiprosesseista. Isäntäprosessi on aina olemassa ja sen tehtävä on valvoa renkiprosesseja. Se luo uusia renkejä tarvittaessa dynaamisesti, ja niitä voi olla rajaton määrä. Isäntäprosessi luodaan järjestelmän käynnistyessä. Rengit toimivat vain sen aikaa kuin niitä tarvitaan.

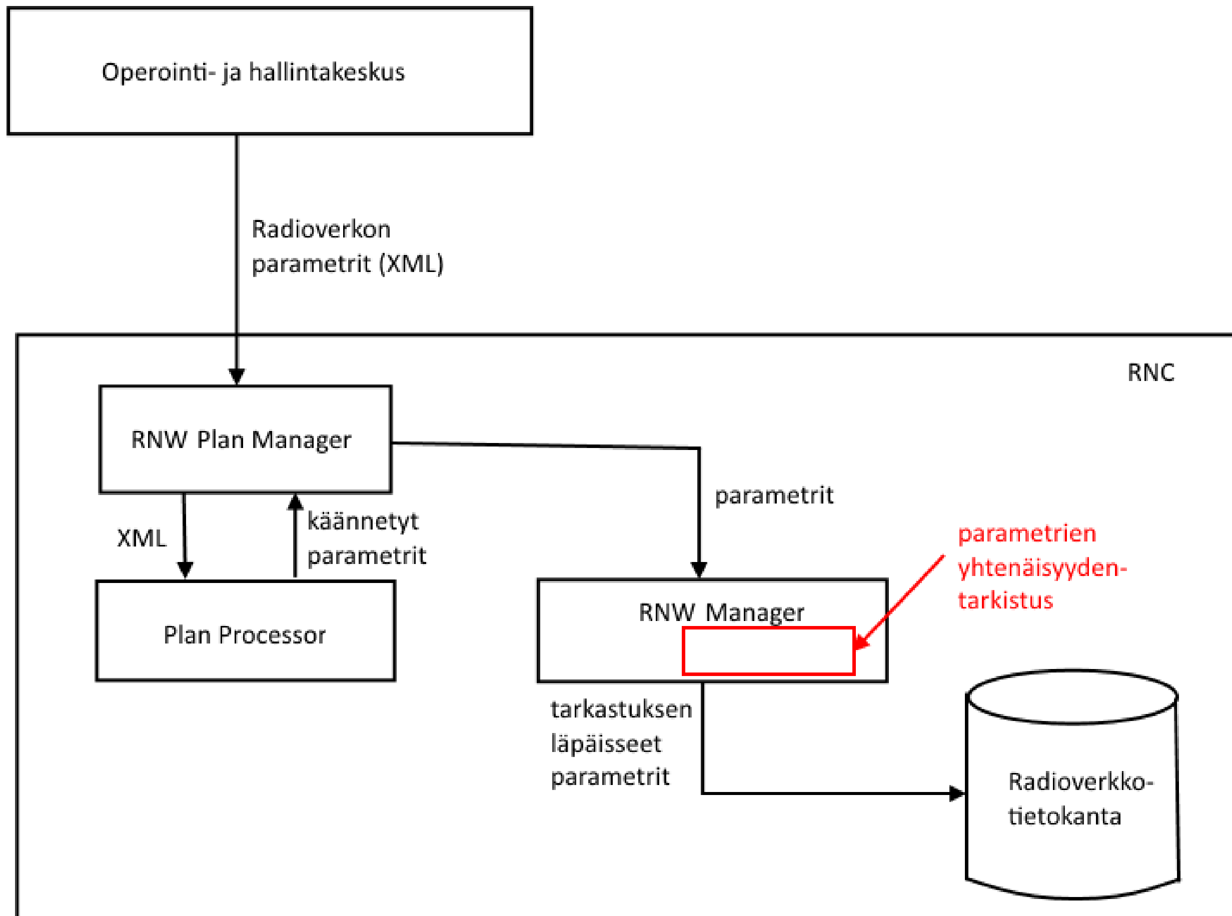
Muut komponentit antavat rakenteen järjestelmälle, mutta prosessit ovat kaiken ydin. Niillä toteutetaan kaikki toiminta käyttäen toiminnallista TNSDL:ää. Yksi prosessi muodostaa yhden tilakoneen. Tilakoneet muodostuvat signaalien välityksestä ja niiden välisistä siirtymistä.

Signaalit vastaanotetaan prosessin sanomajonoon, josta niitä käsitellään siirtymissä sen hetkisen tilan mukaan. Jonot toimivat first-in-first-out -periaatteella, eli jonon kärjessä olevat signaalit käsitellään ensimmäisenä. Jonoja tarvitaan, sillä prosessit pyörivät samanaikaisesti. Signaalit ovat viestejä, joiden avulla prosessit keskustelevat toisilleen, ja jokaisella prosessilla on yksilöivä tunniste, jonka avulla niitä voidaan kutsua. Prosesseissa toiminta käynnistyy vasta sitten, kun ne vastaanottavat signaalin.

Tietomäärittelyt tapahtuvat ”säkissä”. Säkkiin kerätään kaikki tietomäärittelyt, joita järjestelmässä tarvitaan, jotta ne ovat kaikille saatavilla. Keskitetysti tapahtuva määrittely helpottaa myös huomattavasti tietotyyppien muutoksia.

5.3 Tutkimuksen kohde

Konfiguraationhallinta (eng. configuration management) on yksi toiminnallisuus RNC:n sisällä. Se suoritetaan OMU-yksikössä. Konfiguraationhallinnan tarkoituksena on pitää yllä radioverkon parametrien hallintaa. Konfiguraationhallintaan sisältyy toiminnallisuus, jolla radioverkkosuunnitelma (radioverkon parametrit) ladataan XML-muodossa erillisestä verkkoelementistä radioverkkotietokantaan. Tietokanta sisältää kaikki järjestelmän vaatimat tiedot RNC:stä ja sen yhteyksistä, kuten esimerkiksi muista verkkoelementeistä. Tietoa käsitellään pääasiassa neljässä eri ohjelmalohkossa skenaarion aikana. Kyseiset lohkot ovat RNW Plan Manager, Plan Processor, RNW Resource Manager ja RNW Manager. Liitteessä 1 on kuvattu sekvenssiokaaviona konfiguraatiohallinnan toiminta onnistuneessa suunnitelman latauksessa ja kuvassa 4 sama skenaario yksinkertaistettuna. Liitteessä 2 puolestaan skenaarion toiminta on kuvattu tilakaaviona.



Kuva 4. Radioverkkosuunnitelman onnistunut lataus radioverkkotietokantaan.

RNW Plan Manager on järjestelmässä radioverkon määrittelyä hallinnoiva ohjelmalohko. Plan Processor on puolestaan erikoistunut määrittelytiedoston prosessointiin eli muun muassa tiedoston purkuun ja tiedon kääntämiseen. RNW Resource Manager jakaa käytettäviä resursseja, ja RNW Manager hallinnoi RNC:n radioverkkoa ja siinä käytettävien objektien konfigurointia.

Onnistunut tietokannan päivitys on ohjelmalohkotasolla varsin yksinkertainen. RNW Plan Managerissa XML-tiedosto vastaanotetaan ja se lähetetään käsiteltäväksi Plan Processorille. Siellä tiedostoformaatti muutetaan ja tieto konvertoidaan XML:stä järjestelmän käyttämiksi tietorakenteiksi. Sen jälkeen käännetty tieto lähetetään takaisin RNW Plan Managerille, joka tarkistaa tiedon oikeellisuuden hyödyntäen siihen tarkoitettua kirjastoa. Tarkistus sisältää muun muassa raja-arvojen testauksen ja tietotyyppien oikeellisuuden. Jos ongelmia ei ilmennyt, RNW Plan Manager syöttää tiedot tietokantaan RNW Managerin kautta sen tarjoamilla toiminnoilla. RNW Managerissa tarkistetaan tietoja syöttäessä myös tietokannan yhtenäisyys. Yhtenäisyys

tarkistetaan hyödyntämällä ennalta määrättyjä sääntöjä objektien suhteista, joiden avulla tarkistetaan, etteivät syötetyt objektit ole ristiriidassa keskenään. Konfiguraationhallinta toimii myös toiseen suuntaan. Tietokannasta voidaan hakea tiedot, muuttaa ne takaisin XML-muotoon ja toimittaa radioverkkokonfiguraatio takaisin verkon operointi- ja valvontakeskukseen. Prosessi toimii suurin piirtein samalla tavalla, mutta vain käänteisessä järjesteyksessä. Tällöin ei tosin suoriteta objektien tarkistuksia. Muita toimintoja ovat vanhan version palautus ja aktivointi.

Valitsin parametrien hallinnan yhtenäisyydentarkistuksen tapaustutkimukseni kohteeksi. Itsessään yhtenäisyydentarkistuskin kattaa kuitenkin jo kymmeniätuhansia rivejä koodeja. Analysoinnissa hyödynsin koko lähdekoodia, mutta refaktoroinnissa huomio on kiinnitetty rajattuun kohtaan tarkistusta.

5.4 Miksi arkkitehtuuria pitäisi uudistaa?

Tutkittavaa järjestelmän osaa on kehitetty lähes kaksikymmentä vuotta. Sinä aikana sitä on ollut kehittämässä lukuisia eri ohjelmoijia ympäri maailmaa. Koko järjestelmässä on tuhansien ihmisten kädenjälki. Vastuu koodista on siirtynyt useasti eri sijainteihin ja uusilla ominaisuuksilla on yleensä suurempi prioriteetti kuin vanhojen korjauksella. Teknistä velkaa on kertynyt vuosien aikana, joten korjauksia on tehtävä.

Päätavoite refaktoroinnissa on parantaa järjestelmän arkkitehtuuria ja samalla helpottaa testattavuutta. Yksi testattavuuden tavoite olisi tehdä järjestelmän komponenteista testattavia virtualisoidussa ympäristössä. Komponentit pitäisi kyetä erottamaan muista komponenteista ja suorittaa niiden toimintoja alustariippumattomasti. Tällä hetkellä parametrien testaus vaatii fyysisen testausympäristön ja usean komponentin läsnäoloa. Järjestelmässä on liikaa riippuvuuksia modulien ja lohkojen välillä, jotta komponentit voitaisiin käsitellä itsenäisinä. Testattaessa joudutaan tekemään testausta varten rajapintoja, jotka simuloivat muita ohjelmalohkoja. Muutoksia tehtäessä joudutaan kuitenkin päivittämään myös simuloivia rajapintoja. Rajapinnat ovat myös hyvin suuria, joten niiden muodostaminen ja ylläpitäminen on hyvin työlästä. Modulien koko on myös kriittinen testausta vaikeuttava piirre järjestelmässä. Ne ovat liian suuria, jotta niistä voitaisiin muodostaa yhtenäisiä kokonaisuuksia.

Niin ikään järjestelmässä käytettävä radioverkkotietokanta tekee testauksesta erittäin haastavan. Tietokanta sisältää paljon tietoa sadoissa eri tiedostoissa, ja sen rajapinta sisältää satoja eri rajapintamäärittelyjä, joten jokaisen tietokantatoiminnon simulointi vie paljon resursseja. Toistuva koodi kasvattaa ja monimutkaistaa entisestään järjestelmän suurta koodimäärää. Järjestelmään olisi hyvä tuoda myös yksikkötestit korottamaan testikattavuutta.

Tällä hetkellä konfiguraationhallintaa testataan syöttämällä XML-tiedosto ulkoisesta rajapinnasta koko skenaarion läpi radioverkkotietokantaan saakka. Sitten tietokannasta ladataan tiedot takaisin XML-muotoon ja katsotaan, palautuivatko samat arvot kuin syöttäessä. Virheellisiä arvojakin joudutaan testaamaan syöttämällä koko XML-tiedosto uudestaan. Tiedostosta muutetaan yhtä parametria ja katsotaan, mitä tapahtuu. Tällainen testaus vie paljon resursseja.

5.5 Järjestelmän koodin analysointi

Analysoitaessa yhtenäisyydentarkistuksen koodia lukujen 2 ja 3 määrittelyjä vasten, löytyy siitä useita puutteita. Ensisilmäyksellä koodista huomaa suuret modulit ja proseduurit. Suurimmat koodimodulit voivat olla yli 60000 riviä pitkiä ja pisimmät proseduurit yli 3000 riviä pitkiä. While-silmukoita hyödynnetään paljon, ja usein niitä on kaksi sisäkkäistä. Parhaimmillaan yhden proseduurin sisäkkäiset rakenteet voivat ulottua jopa 15 hierarkiatasoon. Tyyliasu ei myöskään ole yhtenäistä, mikä ilmenee esimerkiksi sisennyksien epäsäännöllisyytenä. Muita huomion arvoisia kohteita ovat nimeäminen, toistuva koodi, pitkät ehtolauseet, globaalit muuttujat ja yhtenäisyys. Näitä epäkohtia pyrin avaamaan tarkemmin tässä luvussa.

Yrityksen sisällä, ja myös alalla yleisestikin, on vakiintunut lukemattomia lyhenteitä, joiden ymmärtäminen ja sisäistäminen vie aikaa. Itse koodissakin käytetään paljon sovittuja lyhenteitä, jotka tuovat omat haasteensa järjestelmän parissa ensi kertaa työskentelevälle. Muuttujien ja prosessien nimet voivat olla jopa kolmekymmentä merkkiä pitkiä ja sisältää monia eri lyhenteitä. Ne voi olla niin ikään hyvinkin samankaltaisesti nimettyjä. Kahden pitkän muuttujan tai signaalin nimessä voi olla vain yksi ainoa ero; toisessa on kaksi peräkkäistä ”_”-merkkiä ja toisessa vain yksi, mikä tekee muuttujien erottelusta mahdotonta ilman aikaisempaa tietämystä.

Muuttujien tyyppien selvittäminen on haastavaa, sillä niitä on lukematon määrä ja niiden määrittelyt ovat alikohdassa 5.2.2 mainitussa säkissä, josta jokaisen määrittely täytyy hakea erikseen. Koodi ei siis ole itseään dokumentoivaa.

Ohjelmistosta löytyy suuria määriä toistuvaa koodia, joka voitaisiin korvata parametrisoiduilla proseduureilla. Kymmeniä koodirivejä on saatettu toistaa lukuisia kertoja vain sen takia, että yksi muuttuja on erilainen. Yhdessä kooditiedostossa voi olla satoja rivejä merkilleen toistuvaa koodia, kuten esimerkiksi erinimisiä proseduureja, jotka suorittavat täysin samat asiat.

Yhteen proseduuriin saatetaan suorittaa kutsuja lukuisia kymmeniä kertoja toisesta proseduurista. Eräässä proseduurissa toiseen proseduuriin suoritetaan lähes 800 kutsua, mikä on viidesosa proseduurin koodirivien määrästä. Näissäkin kutsuissa ainoana erona on yhden muuttujan vaihtuminen ja kutsuja voi esiintyä kymmenillä peräkkäisillä riveillä. Funktiokutsut voivat viedä ylimääräisiä resursseja, etenkin vanhemmilla C-kääntäjillä. Kutsuja voitaisiin vähentää esimerkiksi välittämällä funktiolle taulukko parametrina, joka käytäisiin silmukalla läpi funktiossa.

Koodissa hyödynnetään suuria ehtolauseita. Jotkin ehtolauseet voivat sisältää yli kymmenen ehtoa ja viedä jopa parikymmentä riviä koodia. Tällaisista ehtolauseista on vaikea saada yhdellä vilkaisulla selville, mitä kaikkea kohdassa tarkastetaan. Suurten ehtolausekkeiden eristäminen selkeästi nimettyihin proseduureihin vähentäisi huomattavasti ymmärtämiseen käytettävää aikaa. Joitain ehtoja on voitu myös kommentoida välistä pois ilman minkäänlaista tarkennusta. Ylipäätäänkin koodia kommentoidaan paljon pois. Esimerkiksi yhdestä tutkitusta 600 rivin proseduurista saataisiin yli 200 riviä pois pelkän pois kommentoidun koodin siivoamisella ja pitkien ehtolausekkeiden siirtämisellä erilliseen proseduuriin. Koodin sekaan on lisätty myös kommentteja siitä, mikä virhenumero on kyseisellä koodinpätkällä korjattu. Tällaiset kommentit vanhenevat hyvin nopeasti ja jäävät aiheuttamaan turhaa sekaannusta.

Testauksen näkökulmasta koodiin voidaan niin ikään tuoda useita parannuksia. Koodissa käytetään paljon globaaleja muuttujia. Niiden arvoa voidaan muuttaa missä tahansa kohtaa kooditiedostoa, joten on hyvin vaikea saada selville, minkä tyyppisiä tai mitä arvoja niillä voi olla. Tämän takia ne vaikeuttavat etenkin testausta, sillä arvot on alustettava tietyiksi ennen testin

ajoa, jotta testi suoriutuu halutulla tavalla. Koodissa on olemassa myös kohtia, joissa välitetään globaaleja muuttujia parametreina saman prosessin sisällä. Tällaisten parametrien käyttö on kuitenkin tarpeetonta, sillä kutsuttavalla proseduurilla on joka tapauksessa pääsy haluttuun muuttujaan. Globaalien muuttujien käyttöä voidaan vähentää hyödyntämällä enemmän tiedon välitystä parametrien avulla.

Refaktoroitavasta koodista löytyy myös epäsuorasti kutsuja toiseen ohjelmalohkoon. Tällaisten kutsujen suorittaminen hankaloittaa testausta suuresti. Niiden käyttämistä tulisi tietysti ensisijaisesti välttää, mutta aina se ei ole mahdollista. Yksi vaihtoehto kiertää kutsun aiheuttamat riippuvuudet testattaessa on käyttää eri toteutusta proseduurista ajettaessa yksikkötestejä. Esikääntäjälle, eli ennen itse ohjelmakoodin kääntämistä suoritettavalle koodin käsittelylle, voidaan määritellä erilliseen tiedostoon proseduuri, joka ei suoritakaan kutsua toiseen ohjelmalohkoon tai esimerkiksi vaikkapa tietokantaan, vaan sille voidaan antaa vain tyhjä runko ja palautusarvo tarvittaessa. Esikääntäjälle voidaan määritellä kytkin, joka ilmoittaa, suoritetaanko yksikkötestausta vai käännetäänkö ohjelma julkaisua varten. Tällöin eri tilanteisiin saadaan sopiva toteutus. Testiproseduurit kannattaisi muodostaa aina uuteen riippuvuuteen törmättäessä, jotta testirajapinnat olisivat aina ajan tasalla.

Modulien yhtenäisyyttä voitaisiin parantaa huomattavasti. Uusia ominaisuuksia on lisäilty vanhojen rinnalle, eikä missään vaiheessa ole pilkottu moduleja pienemmiksi. Pilkkomalla koodia pienempiin osiin saadaan moduleista yhtenäisempiä, hallittavampia ja helpommin siirrettäviä kokonaisuuksia. Pienemmillä moduleilla on myös yleensä huomattavasti vähemmän virheitä [McConnell, 2004]. Nykyään suurilla moduleilla on valtavasti riippuvuuksia kirjastoihin ja myös muihin ohjelmalohkoihin. Riippuvuudet moninkertaistavat kompleksisuuden testattaessa, kun funktiokutsun takana hyödynnetään uutta globaalia muuttujaa tai funktiokutsua. Pienet modulit mahdollistaisivat pilkkomisen myös korkeammalla tasolla ja jopa uusien ohjelmalohkojen luomisen.

On kuitenkin muistettava, että järjestelmä ja ympäristö asettavat rajoituksia koodille. Nykyään on totuttu käyttämään koodin kanssa kohdassa 4.3 mainittuja ohjelmistoympäristöjä. Niiden avulla on esimerkiksi helppo siirtyä funktiokutsusta funktion toteutukseen. Järjestelmässä käytetylle kielelle, TNSDL:lle, ei kuitenkaan ole tällaisia työkaluja kehitetty. Siirtymisestä liian monen proseduurin välillä tulee nopeasti työlästä ja koodiin eksyy helposti. Lisäosa Eclipse-

alustalle on olemassa, mutta se ei kuitenkaan tarjoa kovin montaa työkalua syntaksin värittämisen lisäksi, ja se toimii epävakaasti, joten sitä ei juurikaan hyödynnetä.

Myös nimeämiselle on olemassa rajoituksia. Edeltävä versionhallintajärjestelmä on vaatinut, että tiedostojen nimissä on oltava tasan kahdeksan merkkiä ja tiedostopäätteellä kolme merkkiä, joten epäselvien lyhenteiden käyttäminen on ollut välttämätöntä. Nykyään käytetään kuitenkin modernimpaa versionhallintaa, joka sallii vapaamman nimeämisen, mutta vanhat käytännöt ovat yhä käytössä. Toinen esimerkki järjestelmän asettamista rajoitteista nimeämiselle on muuttujien tyyppien nimet. Ne on rajoitettu 32 merkkiin, joten jälleen joudutaan turvautumaan lyhenteisiin.

Koodista löytyy siis useita refaktoroinnin kohteita. Järjestelmän vaatimien rajoitusten takia refaktoinnin suorittaminen vaatii kuitenkin ymmärrystä järjestelmästä ja yksikkötestien ja varmistavien testien hyödyntäminen on erittäin kriittistä. Osaan koodista ei enää juurikaan tehdä uusia ominaisuuksia, joten on hyvä myös arvioida, saako refaktoroinnista vaivan arvoista hyötyä.

5.6 Koodin refaktorointi testattavammaksi

Yksi haastavimmista tekijöistä refaktoroinnissa olivat lukemattomat erityyppiset muuttujat, joita järjestelmässä on. Käsitellyssä proseduurissa niitä oli jo 11 erilaista. Näin monen muuttujatyypin kanssa on vaikea tehdä esimerkiksi geneerisiä proseduureja. Muuttujien toiminnasta täytyy myös olla hyvin perillä. Tämä taas on vaikeaa, sillä niitä voidaan muuttaa tai olla muuttamatta muissa proseduureissa, joille ne on annettu parametreina. Toki proseduurin määrittelystä nähdään suhteellisen nopeasti, voidaanko parametria muokata, eli annetaanko parametrista vain osoitin vai kopioidaanko se. Kuitenkin ilman täydellistä käsitystä proseduurista ja sen ympäristöstä on vaikea toteuttaa kaikkia muutoksia, etenkin varmistavien testien ja yksikkötestien puuttuessa. On oltava varma siitä, etteivät tehdyt muutokset aiheuta ongelmia muualla.

Koska refaktoroitava proseduuri oli lähes 450 riviä pitkä, lähdin ensimmäisenä liikkeelle uusien proseduurien luomisella. Korvasin 15 peräkkäistä if-lauseketta yhdellä parametrisoidulla proseduurilla ja siihen tehdyillä kutsuilla, joka on kuvattu liittessä 3. Yksin tällä muutoksella saatiin proseduuria pienennettyä jo yli kolmannes. Seuraavaksi eristin joitain hieman pienempiä kokonaisuuksia kuvaavasti nimetyiksi proseduureiksi, jotka on kuvattu liittessä 4.

Alkuperäisestä proseduurissa esiteltyjen muuttujien määrää voitiin vähentää 14:sta 3:een. Näiden muutosten jälkeen alkuperäinen proseduuri oli enää 50 riviä pitkä (liite 5). Kokonaiskoodimäärä väheni kuitenkin vain 350 riviin, sillä TNSDL:n proseduurien määrittelyt vievät runsaasti tilaa. Proseduureja muodostui yhden sijasta kymmenen. Nyt koodille on abstraktiotasoja, jotka selkeyttävät sitä huomattavasti. Paloittelua olisi voinut suorittaa vieläkin enemmän, mutta on kuitenkin varottava, ettei koodin selaaminen kärsi liian syvien sisäkkäisten proseduurien takia.

Proseduurien muodostamisessa käytin apuna Eclipselle kehitettyä TNSDL-liitännäistä, jolla pystyi maalaamalla valitsemaan eristettävän koodipätkän. Se tarjosi valmiiksi välitettävät parametrit ja muodosti proseduurin automaattisesti. Tällainen työkalu nopeuttaa koodin refaktorointia huomattavasti, mutta täytyy kuitenkin muistaa suhtautua siihenkin kriittisesti. Usein koodista saattaa jäädä esimerkiksi vain parametrina annettavan muuttujan esittely asiakaskoodiin, jolloin liitännäinen tarjoaa muuttujan välitystä parametrina.

Koodin parempi nimeäminen ja siistiminen eivät suoranaisesti liity testattavuuteen, mutta auttavat varmasti testien kirjoituksessa ja koodin ymmärtämisessä. Poistinkin epäselviä lyhenteitä ja turhia muuttujia koodista. Proseduureille annettujen parametrien määrä jäi parissa tapauksessa hieman suureksi, mutta määrää saisi varmasti vielä karsittua, mikäli refaktoroijalla olisi parempi käsitys järjestelmästä ja tekniikoista.

6. Yhteenveto ja päätelmät

Hyvän ja testattavan koodin ominaisuudet ovat ohjelmoinnin alalla varsin yleisesti standardisoituneita ja käytettyjä, etenkin uutta koodia tehdessä. Ominaisuuksien soveltaminen erilaisiin järjestelmiin on ohjelmoijan itse harkittavissa. Korjauksien ja lisäyksien yhteydessä käytetään kuitenkin usein oikoteitä, jotka puolestaan kasvattavat teknistä velkaa.

Tutkitun järjestelmän koon ja käytettyjen tekniikoiden takia joitain kirjallisuudesta löytyviä suosituksia kannattaa soveltaa hyvin harkiten. Kirjallisuudessa on keskitytty suurimmaksi osaksi yleisiin kieliin, kun taas järjestelmässä on käytetty itse kehitettyjä menetelmiä, joiden kanssa ei voida hyödyntää yleisesti saatavilla olevia työkaluja. Myös esimerkiksi Martinin [2008] säännöt funktioiden pituudesta ja parametrien määrästä ovat liian tiukkoja kyseiseen järjestelmään. Optimointi on niin ikään otettava yhä huomioon arkkitehtuurillisissa ratkaisuissa, vaikkakin tehojen kasvaessa ja kääntäjien kehittyessä yhä vähenevissä määrin.

Hankalan järjestelmän refaktoroinnista tekee myös se, että refaktorointi vaatii hyvää ymmärrystä järjestelmästä. Jos ei ole työskennellyt järjestelmän tai koodin kanssa, on hyvin vaikea yrittää tehdä parannuksia. Refaktorointiin vaaditaan siis kokenut henkilö. Esimerkiksi omintakeiset nimeämiskäytännöt, erilaiset ajoympäristöt ja versiot voivat aiheuttaa hämmennystä. Vahvan ohjelmiston tuntemuksen lisäksi refaktorointi vaatii kattavan regressiotestauksen, jotta muutosten toiminnallisuus voidaan varmistaa.

Työmäärää refaktoroinnille on hyvin vaikea arvioida. Jos arvioidaan, että tarkasteltu ohjelmalohko sisältää noin 200 000 riviä koodia ja yhdessä työtunnissa voitaisiin refaktoroida noin 90 riviä (perustuen omaan suoritukseeni), kestäisi koko ohjelmalohkon refaktoroimisessa yli 2200 tuntia. Lisäksi täytyy huomioida, että refaktoroimani proseduuri ei ollut vaikeimmasta päästä, eikä aikaan sisälly vielä ollenkaan testausta.

Pyrittäessä koodin testattavuuteen, on hyvä lähteä liikkeelle yksikkötesteistä. Yksikkötestit auttavat ymmärtämään ja paikallistamaan koodissa olevia riippuvuuksia, parantavat rakennetta, sekä tietysti ilmoittavat virheistä. Yksikkötestien tekeminen on myös olennaista pyrittäessä eroon perinnekoodista, sillä ne ovat tärkeässä osassa refaktorointia. Refaktoroinnissa kannattaa aloittaa

pienistä muutoksista, jotka myöhemmin helpottavat suurempien arkkitehtuurillisten muutosten tekemistä. Yksi tehokkaimmista keinoista on muodostaa pitkistä proseduureista pienempiä kokonaisuuksia. Pienemmille proseduureille on huomattavasti helpompi kirjoittaa yksikkötestejä, kuin yhdelle valtavalle proseduurille.

Tutkittu koodi kaipaakin ensisijaisesti pilkkomista. Proseduurit ja siirtymät olisi hyvä saada vähintäänkin alle 100 rivin mittaisiksi, jotta ylläpito ja luettavuus olisivat parempaa. Pilkkominen helpottuisi huomattavasti, mikäli päästäisiin eroon suuresta määrästä muuttujatyyppejä. Jatkossa on kuitenkin arvioitava kumpi on tärkeämpää, koodin uudelleenkäyttö ja luettavuus vai muuttujatyyppien helppo päivittäminen. Lisäksi koodia täytyy siivota yleisesti, kuten poistaa turhia kommentteja ja pois kommentoitua koodia.

On kuitenkin muistettava, ettei refaktorointi ole mikään lopullinen ratkaisu perinnekoodiin. Ei voida vain olettaa, että koodi refaktoroidaan ja sen jälkeen kaikki on hyvin. Sitä tulee suorittaa koko ajan uusia ominaisuuksia lisätessä ja virheitä korjataessa tai muuten koodi väkisinkin rappeutuu. Luvussa 5 analysoitua koodia on kehitetty lähes kaksi vuosikymmentä vähäisellä ylläpidolla. Kun koodi on ensimmäisen kerran kirjoitettu, se on varmasti ollut hyvin jäsennelty ja tarkkaan harkittu kokonaisuus. Ajan myötä on kuitenkin väistämätöntä, että uudet vaatimukset muokkaavat arkkitehtuuria, joten uudelleensuunnittelu on jossain vaiheessa välttämätöntä.

Työn alkuperäinen tarkoitus oli saada ohjelmalohkoista helpommin itsenäisesti testattavia kokonaisuuksia. Suurin este tälle on, että ohjelmalohkot ovat liian suuria ja keskustelevat tiiviisti muiden kanssa. Tutkimuksen mukaan paras ratkaisu olisi muodostaa pienempiä ja yhtenäisempiä ohjelmalohkoja, joiden itsenäinen testaus ei vaatisi niin suurta ympäristön simuloimista.

Viiteluettelo

[3GPP, 1999] 3GPP, 5.x Radio Interface No. X. http://www.3gpp.org/ftp/tsg_ran/tsg_ran/TSGR_05/Docs/Pdfs/rp-99500.pdf. 2009. Checked 12.03.2014.

[Fog, 2014] Agner Fog, Optimizing Software in C++: An Optimization Guide for Windows, Linux and Mac Platforms. http://www.agner.org/optimize/optimizing_cpp.pdf. 2014. Checked 20.03.2014.

[Bavota et al., 2013] Gabriele Bavota, Andrea De Lucia, Andrian Marcus and Rocco Oliveto, Automating extract class refactoring: an improved method and its evaluation. Empirical Software Engineering, To appear in 2014.

[Beck and Diehl, 2011] Fabian Beck and Stephan Diehl, On the congruence of modularity and code coupling. In: Proc. of 19th ACM SIGSOFT Symposium and the 13th European Conference on Software Engineering 2011, 354-364.

[BlackBoxRecorder, 2014] BlackBoxRecorder, Characterization test generator for .NET. http://follesoe.github.io/BlackBoxRecorder/. Checked 04.03.2014.

[Bræk, 1996] Rolv Bræk, SDL Basics. *Computer Networks and ISDN Systems* **28**, 12 (June 1996), 1585-1602.

[Counsell et al., 2010] S. Counsell, H. Hamza and R.M. Hierons, The ‘deception’ of code smells: An empirical investigation. In: *Proc. of 32nd International Conference on Information Technology Interfaces 2010,* 683-688.

[Dantas, 2011] Francisco Dantas, Reuse vs. maintainability: revealing the impact of composition code properties. In: *Proc. of 33rd International Conference on Software Engineering 2011,* 1082-1085.

[Dogša and Batič, 2011] Tomaž Dogša and David Batič, The effectiveness of test-driven development: an industrial case study. *Software Quality Journal* **19**, 4 (Dec. 2011), 643-661.

[Duffy, 2008] Joe Duffy, *Concurrent Programming on Windows.* Addison-Wesley Professional, 2008.

[Feathers, 2004] Michael Feathers, *Working Effectively with Legacy Code.* Prentice Hall, 2004.

[Fowler et al., 1999] Marting Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[Goodliffe, 2006] Pete Goodliffe, *Code Craft: The Practice of Writing Excellent Code.* No Starch Press, Inc., 2006.

[Gorton, 2006] Ian Gorton, *Essential Software Architecture.* Springer-Verlag, 2006.

[Grenning, 2011] James Grenning, Test-Driven Development for Embedded C. Pragmatic Programmers, LLC, 2011.

[Griffith et al., 2011] Isaac Griffith, Scott Wahl and Clemente Izurieta, Evolution of legacy system comprehensibility through automated refactoring. In: *Proc. of MALETS'11,* 35-42.

[Hevery, 2008] Miško Hevery, Guide: Writing Testable Code. *http://misko.hevery.com/attachments/Guide-Writing/Testable/Code.pdf.* 2008. Checked 13.02.2014.

[Harsu, 2003] Maarit Harsu, *Ohjelmien ylläpito ja uudistaminen*. Talentum Media Oy, 2003.

[Kerninghan and Ritchie, 1988] Brian Kerninghan and Dennis Ritchie, *The C Programming Language.* Prentice-Hall, 1988.

[Kim et al., 2012] Miryung Kim, Thomas Zimmermann and Nachiappan Nagappan, A field study of refactoring challenges and benefits. In: *Proc. of 20th FSE'12,* 11.

[Klinger et al., 2011] Tim Klinger, Peri Tarr, Patrick Wagstrom and Clay Williams, An enterprise perspective on technical debt. In: *Proc. of 2nd MTD'11,* 35-38.

[Kästner et al., 2011] Christian Kästner, Sven Apel and Klaus Ostermann, The road to feature modularity?. In: *Proc. of 15th SPCL'11, 5:1-5:8.*

[Osherove, 2009] Roy Osherove, *The Art of Unit Testing: With Examples in .Net*. Manning Publications, 2009.

[Martin, 2008] Robert Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, Inc, 2008.

[McConnell, 2004] Steve McConnell, *Code Complete, Second Edition.* Microsoft Press, 2004.

[Merton and Tempero, 2006] Hayden Melton and Ewan Tempero, Identifying refactoring opportunities by identifying dependency cycles. In: *Proc. of 29th ACSC'06,* 35-41.

[Munro, 2005] Matthew Munro, Product metrics for automatic identification of "bad smell" design problems in java source-code. In: *Proc. of 11th IEEE METRICS 2005, 15.*

[Murphy-Hill, 2009] Emerson Murphy-Hill, Chris Parnin and Andrew Black, How we refactor, and how we know it. In: *Proc. of 31st ICSE '09,* 287-297.

[Lindqvist et al., 2013] Markus Lindqvist, Erkki Ruohtula, Heikki Tuominen and Markus Malmqvist, *The TNSDL Book.* Networks, Nokia, Internal report. 2013.

[Ostermann, 2011] Klaus Ostermann, Paolo Giarrusso, Christian Kästner and Tillmann Rendel, Revisiting information hiding: reflections on classical and nonclassical modularity. In: *Proc. of 25th ECOOP 2011*, 155-178.

[Pirkelbauer et al., 2010] Peter Pirkelbauer, Damian Dechev and Bjarne Stroustrup, Source code rejuvenation is not refactoring. In: *Proc. of 36th SOFSEM 2010: Theory and Practice of Computer Science,* 639-650.

[Poston et al., 2011] Robin Poston, Jignya Patel, Fatima Qadri and Bindu Varriam, Applying testability concepts to create testability guidelines. University of Memphis, Systems Testing of Excellence Program, September 2011. Available as *https://umdrive.memphis.edu/g-mis/www/memphis/step/STEP2011/step2011_p9_Poston_et_al.doc.*

[ProGuard, 2014] ProGuard, Java Optimizer and Obfuscator http://proguard.sourceforge.net/. Checked 20.03.2014.

[Ratzinger et al., 2005] Jacek Ratzinger, Michael Fischer and Harald Gall, Improving evolvability through refactoring. In: *Proc. of MSR'05,* 1-5.

[Reiss, 2013] David Reiss, Under the Hood: Dalvik patch for Facebook for Android. https://www.facebook.com/notes/facebook-engineering/under-the-hood-dalvik-patch-for-facebook-for-android/10151345597798920. 2013. Checked 30.01.2014.

[Ritchie, 2010] Peter Ritchie, *Refactoring with Microsoft Visual Studio 2010.* Packt Publishing Ltd, 2010.

[Schäfer, 2012] Max Schäfer, Refactoring tools for dynamic languages. In: *Proc. of WRT'12*, 59-62.

[Taube-Schock et al., 2011] Craig Taube-Schock, Robert Walker and Ian Witten, Can we avoid high coupling?. In: *Proc. of 25th* ECOOP 2011, 204-228.

[Wagner et al., 2006] Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner and Peter Wolstenholme, *Modeling Software with Finite State Machines: A Practical Approach.* Auerbach Publications, 2006.

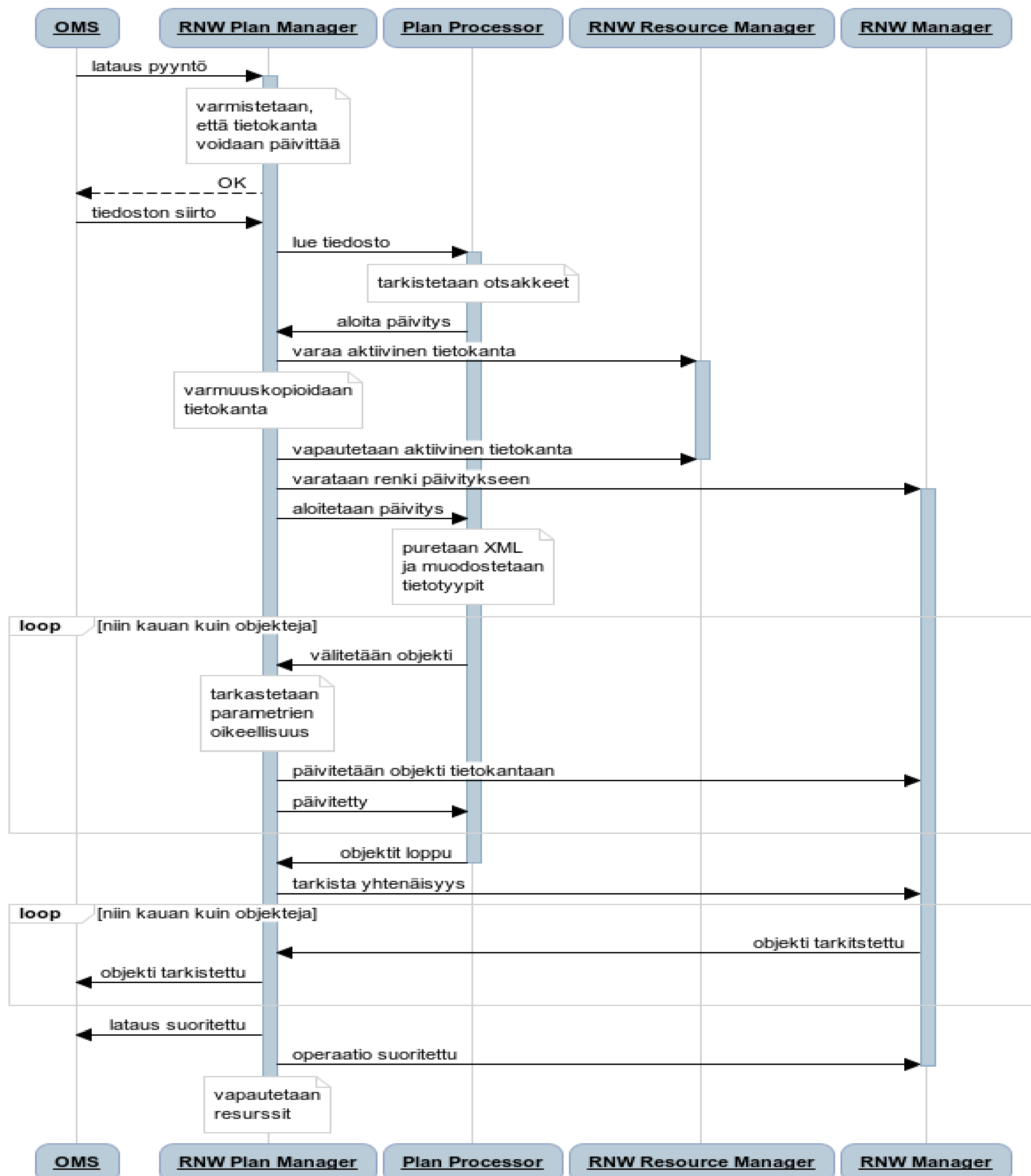
[Wong et al., 2011] Sunny Wong, Yuanfang Cai, Miryung Kim and Michael Dalton, Detecting software modularity violations. In: *Proc. of 33rd ICSE'11,* 411-420.

[Zazworka et al., 2013] Nico Zazworka, Antonio Vetro', Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman and Forrest Shull, Comparing four approaches for technical debt identification. *Software Quality Journal* **21**, 2 (2013).

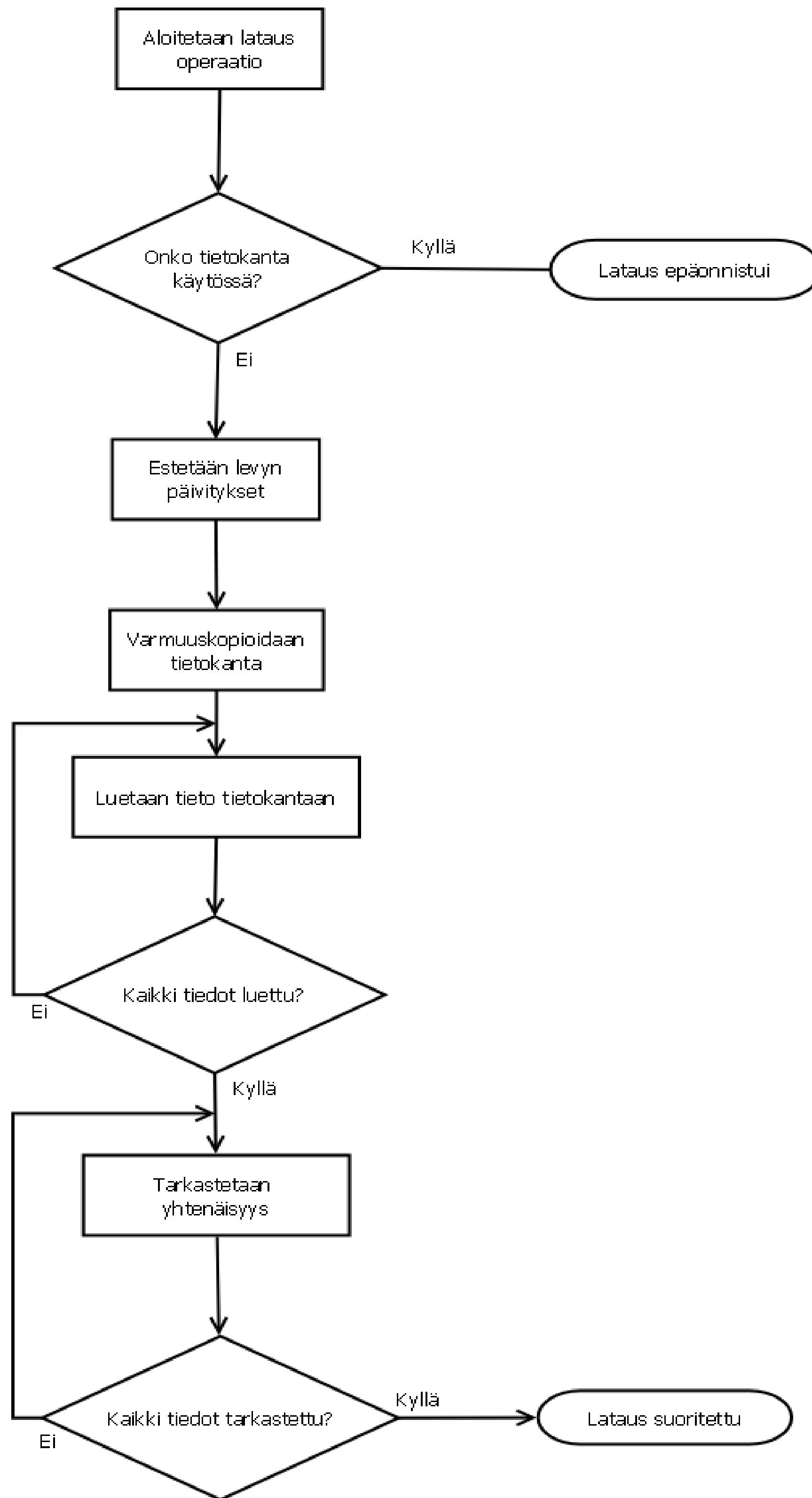
[Zibran and Roy, 2011] Minhaz Zibran and Chanchal Roy, Towards flexible code clone detection, management, and refactoring in IDE. In: *Proc. of 5th IWSC'11,* 75-76.

Liitteet

Liite 1



Liite 2



Liite 3

poistetut muuttujat (11/14)

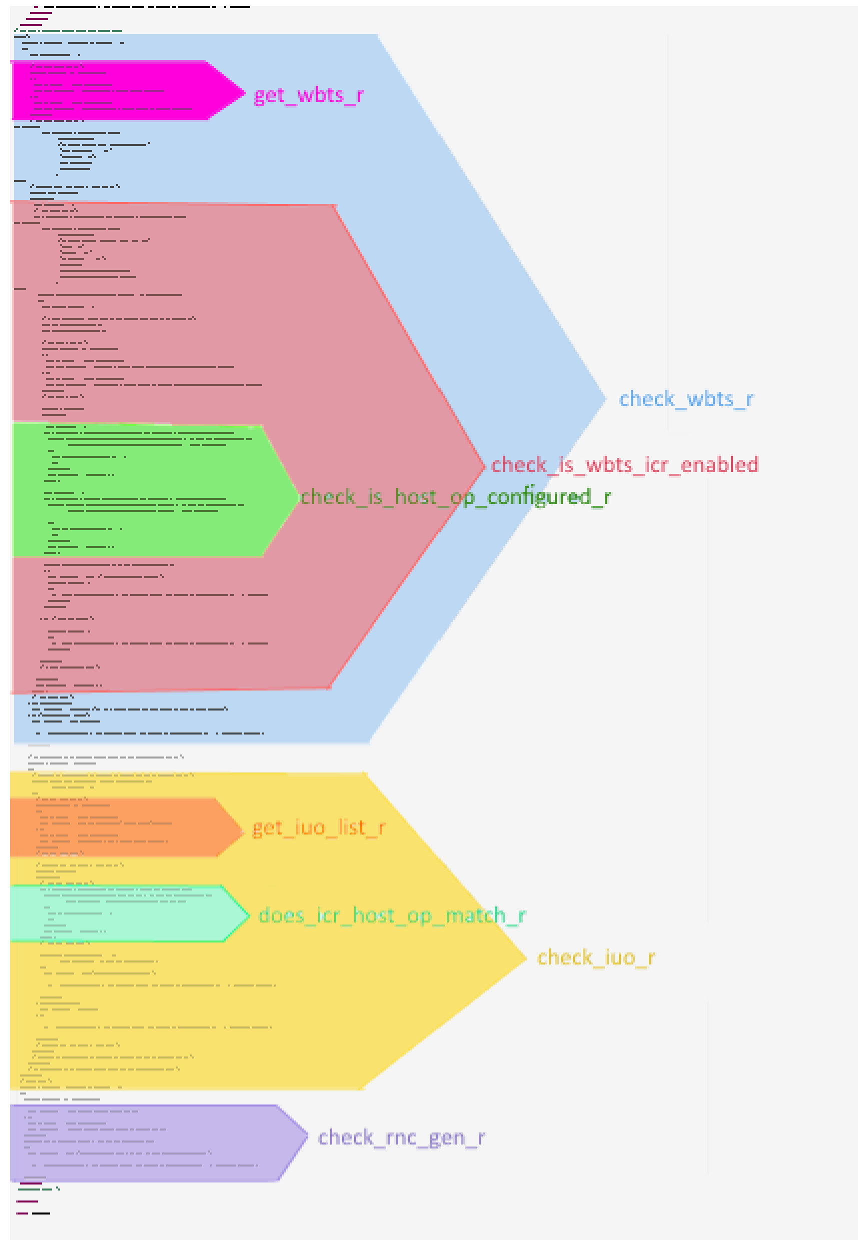
```
PROCEDURE call_rnw_db_consistency_check_r;

   FPAR
      IN/OUT return_status  error_t,
      IN is_plan_operation  bool,
      IN compare1           word,
      IN compare2           word;

   START;
      DECISION (return_status = success_ec OR is_plan_oper);
      (T):
         DECISION (compare1 >= compare2);
         (T):
            DECISION (is_plan_oper);
            (T):
               CALL send_rnw_db_cons_check_ack (rnw_rnc_obj_t_rnac_c, rna.rnac_id,
               lcr_id_t_not_def_c, 0xFFFF, return_status);
            ENDDECISION;
         ENDDECISION;
      ENDDECISION;

ENDPROCEDURE call_rnw_db_consistency_check_r;
```

Liite 4



Liite 5

```
PROCEDURE check_rna_data_validity;

  FPAR
    IN/OUT rna                rna_t,                  /* checked data */
    IN/OUT bitmask            rnw_object_bitmask_t,   /* specifies which params are actually changed */
    IN operation_type         oper_type__t,           /* defines operation type. All checks are not needed */
    IN is_plan_operation      bool,
    IN is_ray_cons_check      bool;                   /* DI58 RAN1646 */

  RETURNS
    error_t;

 DCL
  return_status               error_t      := success_ec,
  is_wbts_icr_enabled         bool         := F,
  database_rgn                rgn_t;

START;
  TASK memset( bytepointer( @db_rgn ), 0, SIZEOF( db_rgn) );

  DECISION (operation_type);
  (oper_type__t_create_c, oper_type__t_modify_c):
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.dl_max_sf.dl_max_br_sf256, rna.dl_max_sf.dl_max_br_sf128, 2232);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.dl_max_sf.dl_max_br_sf128, rna.dl_max_sf.dl_max_br_sf64, 2233);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.ul_max_sf.dl_max_br_sf256, rna.ul_max_sf.dl_max_br_sf128, 2250);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.ul_max_sf.ul_max_br_sf128, rna.ul_max_sf.ul_max_br_sf64, 2251);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.dl_max_sf.dl_max_br_sf32, rna.dl_max_sf.dl_max_br_sf16, 2257);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.dl_max_sf.dl_max_br_sf64, rna.dl_max_sf.dl_max_br_sf32, 2258);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.ul_max_sf.ul_max_br_sf32, rna.ul_max_sf.ul_max_br_sf16, 2255);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.ul_max_sf.ul_max_br_sf16, rna.ul_max_sf.ul_max_br_sf8, 2262);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.ul_max_sf.ul_max_br_sf64, rna.ul_max_sf.ul_max_br_sf32, 2256);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.auto_acinc_resthtwp+1, rna.auto_ac_reslev_updint, 2419);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.auto_acdec_resthtwp+1, rna.auto_ac_reslev_updint, 2420);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.auto_acdec_resthnrtq+1, rna.auto_ac_reslev_updint, 2422);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.auto_acinc_resthnrtq+1, rna.auto_ac_reslev_updint, 2421);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.auto_acdec_resthtwp+1, rna.auto_acinc_resthtwp, 2423);
    CALL call_rnw_db_consistency_check_r (return_status, is_plan_operation, rna.auto_acdec_resthnrtq+1, rna.auto_acinc_resthnrtq, 2424);

    DECISION (return_status = success_ec OR is_plan_operation);
    (T):
        TASK is_wbts_icr_enabled := check_wbts_r (is_plan_operation, is_ray_cons_check, return_status, rna);
        CALL check_iuo_r (return_status, bitmask, is_wbts_icr_enabled, is_plan_operation, is_ray_cons_check, rna);
    ENDDECISION;

    DECISION (return_status = success_ec OR is_plan_operation);
    (T):
      CALL check_rnc_gen_r (is_plan_operation, is_ray_cons_check, return_status, db_rgn, rna);
    ENDDECISION;

  ENDDECISION;
  RETURN return_status;
ENDPROCEDURE check_rna_data;
```

parametrit

palautusarvo

muuttujat