Hannes Nieminen

Laadunvalvonnan helpottaminen automatisoinnilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

19.5.2014

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Hannes Nieminen Laadunvalvonnan helpottaminen automatisoinnilla 36 sivua + 4 liitettä 19.5.2014
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander Projektipäällikkö Timo Gröhn

Työ tehtiin rakennusalan suunnittelun ja konsultoinnin, sekä ohjelmistokehityksen asiantuntijayritykselle Granlund Oy:lle, jossa työskentelee yhteensä noin 500 talotekniikan sekä kiinteistö-, energia- ja ohjelmistoalan ammattilaista. Työn tarkoituksena oli parantaa vielä suurelta osalta kehitysvaiheessa olevan Granlund Designer -verkkosovelluksen lähdekoodin laadunvalvontaa ja ylläpitoa. Lisäksi helpottaa katselmointia sekä automatisoida mahdollisimman paljon näihin liittyviä toimenpiteitä.

Suurena osana työssä oli NDepend- ja ReSharper-laadunvalvontatyökalujen käyttöönotto. Työssä kartoitettiin, miten sovelluksen laadunvalvontaa voidaan parantaa, mitkä ovat kehitysryhmän toivomia muutoksia, mitä asioita sovellukset mahdollistavat, valittiin sovelluksista halutut toiminnallisuudet sekä tehtiin käyttöönotto asennuksia, asetuksien säätöä ja kehittäjien perehdyttämistä myöten. Työssä käsiteltiin myös laatua käsitteenä ja sen merkitystä sovelluskehityksessä. Lähdekoodin laadun saavuttamisessa tärkeitä asioita ovat ylläpidettävyys, joustavuus, tehokkuus, modulaarisuus, uudelleenkäytettävyys, luettavuus, testattavuus ja vikasietoisuus. Edellä mainitut sovellukset mahdollistavat paljon erilaisia mittareita, joilla voidaan automaattisesti seurata koodin laatua ja yhdenmukaisuutta, sekä valvoa, että asiat tehdään ennalta määriteltyjen ohjeiden ja sovelluskehitysryhmän käytäntöjen mukaisesti. Sovelluksilla voidaan myös automatisoida raportteja valittujen parametrien mukaisesti, joilla voidaan pitää yllä käsitystä sovelluksen tilasta helposti.

Työssä päästiin haluttuihin tavoitteisiin NDependin ja ReSharperin osalta ja lopputulokseen voidaan olla tyytyväisiä. Työkalut helpottavat huomattavasti virheiden havaitsemista ja laadun ylläpitämistä haluttujen sääntöjen mukaisesti. NDepend luo raportin aina kun muutoksia viedään versiohallintaan ja kääntöpalvelin tekee uuden käännöksen. ReSharper taas on jatkuvasti käytössä jokaisen kehittäjän työympäristössä antamassa korjausehdotuksia ja parantamassa kehitystyötä. Katselmoinnin osalta ei kuitenkaan päästy aivan toivottuun lopputulokseen käyttämällä NDepend tai ReSharper sovelluksia, mitä myös epäiltiin työtä aloitettaessa. Haluttiin kuitenkin määritellä, josko toiminnallisuuden olisi saanut näitä työkaluja käyttäen toteutettua. Lopuksi päätettiin, että tulevaisuudessa tutkitaan Review Assistant -sovellusta, jonka pitäisi soveltua tarkoitukseen hyvin.

Avainsanat	lähdekoodi, laadunparannus, laatu, työkalut, ylläpito, analyysi, työkalu, automaatio, NDepend, ReSharper

Author(s) Title Number of Pages Date	Hannes Nieminen Improving Code Quality Assurance with Automation 36 pages + 4 appendices 19 May 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Timo Gröhn, Project Manager Simo Silander, Senior Lecturer

The purpose of this Bachelor's Thesis was to research improving the code quality assurance and code review process of Granlund Designer web application that is still heavily under development. The study was conducted at the request of Granlund Oy, an expert designing, consulting and software development company with over 500 employees.

Deployment of NDepend and ReSharper quality assurance tools were in a key part. During this study it was determined how to improve the quality assurance process, what changes the development team wished and what these tools enabled. The tools were also deployed, configured and introduced to the development team. Quality as a term and what it means in software development specifically was also researched. To achieve high source code quality important issues to focus on are maintainability, flexibility, efficiency, modularity, reusability, readability, testability and resilience. The tools mentioned above have a lot of meters that can be used to monitor these factors and provide a view of the code quality and consistency. The tools can also be used to generate automated customizable reports of the state of the program.

The desired result was achieved during the study. NDepend and ReSharper were deployed and provide help in monitoring and improving the code quality. NDepend generates a report every time a developer pushes new changes to the revision control and the continuous integration does a new build. While ReSharper is continuously in use on every developer's workstation, it informs about mistakes and provides easy fixes and improvements. New code review practices were decided and since NDepend and ReSharper do not provide anything for code reviewing research on a review tool called Review Assistant was considered as a future step.

Keywords	source code, quality assurance, quality, tools, analysis, automation, NDepend, ReSharper

Sisällys

Lyhenteet

GD	Granlund Designer -sovellus.
GM	Granlund Manager -sovellus.
IDE	Integrated development environment. Sovelluskehitysympäristö, joka sisältää kaiken tarpeellisen sovelluskehittäjälle. Yleensä sisältää lähdekoodi editorin, debuggerin, kääntäjän ja sovelluksen ajamisen.
VS	Microsoft Visual Studio IDE.
TFS	Microsoft Team Foundation Server. Versionhallintajärjestelmä, joka on käytössä GD-kehitysryhmällä.
.NET	.NET Framework. Microsoftin kehittämä .NET-sovelluskehys. Pääasiassa verkkosovellus kehitykseen Microsoft Windows -ympäristössä.
ORM	Object-relational mapping. Oliomallin mukaisen esityksen kuvaus relaatiomallin mukaiseksi esitykseksi.
EF	Microsoft Entity Framework. Avoimen lähdekoodin ORM-alusta .NET-sovelluskehykseen.
LINQ	Language-Integrated Query. .NET-sovelluskehyksen komponentti kyselyjen muodostamiseen.
LOC	Lines of code. Lähdekoodin rivimäärä. Käytetään usein yhtenä mittareista arvioitaessa sovellusta.
MVC	Model-View-Controller-sovelluskehitysmalli. Käytössä GD:ssa.
SRP	Single Responsibility Principle -arkkitehtuuri ratkaisu sovelluskehityksessä.
PDB	Visual Studio IDE:n käyttämä program database-tiedostomuoto, johon tallennetaan sovelluksen debug- ja projektitietoja.

IL	NDependissä käytetty Intermediate Language viittaa generoidusta koodista laskettuihin kutsujen määriin mittareissa. Esimerkiksi ”for” tekee kaksi ja ”foreach” tekee kolme kutsua.
IntelliSense	Visual Studiosta löytyvä Intelligent code completion -ominaisuus eli älykäs koodin automaattinen täydennys.

1 Johdanto

Työ tehdään Granlund Oy -yritykselle. Kyseessä on rakennusalan suunnittelun ja konsultoinnin sekä ohjelmistokehityksen asiantuntijayritys, jossa työskentelee yhteensä noin 500 talotekniikan sekä kiinteistö-, energia- ja ohjelmistoalan ammattilaista. Granlundilla on kokemusta alalta yli 50 vuoden ajalta. Yrityksen ohjelmistokehitysosaston toiminta on siis vain pieni, mutta myös tärkeä ja avainasemassa oleva osa yrityksen toimintaa. Yritykseltä löytyy muita liiketoiminnan osa-alueita tukevia ja mahdollistavia sovelluksia, joista tunnetuimpana Granlund Manager.

Tässä työssä kuitenkin käsitellään viimeisintä ja vielä suurelta osalta kehitysvaiheessa olevaa Granlund Designer (GD) -sovellusta. Työn tarkoituksena olisi parantaa Granlund Designer -sovelluksen lähdekoodin laadunvalvontaa ja -ylläpitoa, helpottaa katselmointia sekä automatisoida mahdollisimman paljon näihin liittyviä toimenpiteitä.

Kyseessä on noin kaksi vuotta kehityksessä ollut rakennusalan arkkitehdeille kohdistettu .NET-verkkosovellus. Sovellusta kehitetään yrityksen sisällä sekä omaan että asiakkaiden käyttöön. Sovelluksen parissa työskentelee ammattitaitoinen, koodilaatutietoinen kehitysryhmä, ja laadun ylläpitämiseen tehdään aktiivisesti toimenpiteitä. Parantamisen varaa kuitenkin riittää varsinkin automatisoinnin osalta. Koodikatselmointi on tällä hetkellä keskittynyt lähinnä sovellusarkkitehdille, joka työllistää yhtä henkilöä turhan paljon ja esimerkiksi tätä haluttaisiin hajauttaa.

Yrityksen toisen sovelluskehitysryhmän Granlund Manager -projektia varten on hankittu sovelluksia laadunhallintaa helpottamaan ja automatisoimaan. Näitä ovat NDepend, ReSharper ja Review Assistant. Yrityksen sisäisen yhdenmukaisuuden ja lisenssien tähden halutaan hyödyntää samoja työkaluja myös muissa projekteissa kuten GD:ssä. Sovelluksien valinta ja lisenssien ostaminen on siis jo tehty, mutta käyttöön otettavia ominaisuuksia ei ole kartoitettu ja itse käyttöönottoa ei ole myöskään tehty. Käyttöönotto päätettiin suorittaa ensin GD:lle ja sen jälkeen tehdä sama GM-puolelle.

Sovellukset mahdollistavat erilaisia mittareita, joilla voidaan automaattisesti seurata koodin laatua ja yhdenmukaisuutta sekä valvoa, että asiat tehdään ennalta määriteltyjen ohjeiden ja sovelluskehitysryhmän käytäntöjen mukaisesti. Sovelluksilla

voidaan myös automatisoida raportteja valittujen parametrien mukaisesti, joilla voidaan pitää yllä käsitystä sovelluksen tilasta helposti. Suunnitteilla on myös ns. ilmoitustaulu, jossa näkyy reaaliaikaisia tilastoja sovelluksesta ja viimeisimmistä versionhallintamuutoksista sekä jatkuvan integraation käännöksistä ja testituloksista. Näin kehittäjät pysyisivät ajan tasalla muutoksista ja ongelmatapauksista, sekä työn tulokset näkyisivät selkeämmin.

Työn tarkoituksena on siis kartoittaa kehitysryhmän toivomia muutoksia, sovelluksien antamia ominaisuuksia, valita sovelluksista halutut toiminnallisuudet sekä tehdä käyttöönotto asennuksineen ja asetuksineen.

2 Granlund Designer

Granlund Designer on rakennusalan arkkitehdeille kohdistettu .NET MVC -verkkosovellus rakennusprojektien suunnittelua ja hallintaa varten. Sovellusta kehitetään yrityksen sisäiseen käyttöön sekä asiakkaille myytäväksi palveluksi. Se on vielä kehitysvaiheessa, eikä sitä ole vielä julkaistu tuotantoon. Uusimman version kehitys aloitettiin tyhjästä uusilla teknologioilla, työkaluilla ja arkkitehtuurilla noin kaksi vuotta sitten. Tämä päätös perustui siihen, että haluttiin siirtyä teknologian kannalta ajan tasalle ja parantaa sovelluksen suorituskykyä sekä laatua.

Kehitysryhmällä on käytössä Visual Studio 2013 (VS) -IDE-kehitysympäristö, jolla kehitys tapahtuu. Versionhallintajärjestelmänä toimii Microsoftin Team Foundation Server (TFS), johon on suora integraatio VS-ympäristössä. Sovellus käyttää hyvin paljon tietokantoja, joiden pohjalla toimii Microsoft SQL Server. Tietokantoja hyödynnetään sovelluksessa Microsoft Entity Framework (EF) ORM-rajapintaa käyttäen, joka vähentää huomattavasti kyselyihin vaadittua koodirivimäärää ja helpottaa kehittäjien työtä kyselyiden tekemisen osalta. Kehityskielinä toimii back-end eli palvelinpuolella C# ja front-end eli käyttöliittymäpuolella Javascript, sekä Jquery-kirjasto, jolla laajennetaan Javascript-toiminnallisuutta. Käyttöliittymien tekemiseen käytetään myös Telerikin komponenttia nimeltä Kendo UI, jonka päälle on rakennettu yhdenmukaiset omat komponentit, joita sovelluskehittäjien on helppo hyödyntää. Näkymät toteutetaan C# Razor -syntaksilla.

GD-kehitysryhmän koko on 7 henkilöä, ja kehitysmenetelmänä käytetään Scrum-tekniikkaa kuukauden pituisilla iteraatioilla eli sykleillä. Suurin osa työntekijöistä on ollut projektissa mukana sen alusta alkaen, joten heillä on kattava käsitys sovelluksen historiasta, tämänhetkisestä tilasta sekä tulevaisuudesta. Laadun parantamista ja ylläpitoa pidetään tärkeänä ja sen eteen tehdään jatkuvasti toimenpiteitä, mutta parantamisen varaa kuitenkin on.

Sovelluksen arkkitehtuuriin tehtiin muutos vuoden alkupuolella. Siirto vanhasta uuteen arkkitehtuuriin on edelleen kesken, ja sitä suoritetaan normaalien työtehtävien ohella tarpeen mukaan järkeväksi tulkituissa tapauksissa. Tämän päätöksen pohjana oli sovelluskehityksen aikana opittu uusi tieto ja halu yksinkertaistaa ja parantaa rakennetta kehityksen selkeyttämiseksi. Muutokseen ollaan oltu erittäin tyytyväisiä.

Yksikkötestaus kuuluu myös projektin käytäntöihin. Palvelinpuolella C#-koodi testataan käyttäen Visual Studion sisältämää yksikkötestaustoiminnallisuutta. Käytäntönä on ajaa testit ennen muutoksien versionhallintajärjestelmään laittamista. Lisäksi muutosten sisäänlaiton jälkeen jatkuvan integraation palvelin suorittaa sovelluksen kääntämisen sekä ajaa testit palvelimella. Jos kääntö tai testit epäonnistuvat, tulee siitä automatisoitu ilmoitus sovelluskehittäjälle sekä sovelluksesta vastaavalle arkkitehdille ja projektipäällikölle. Käyttöliittymäpuolen Javascript-toiminnallisuuden testaukseen käytetään Javascriptille tehtyä Qunit-yksikkötestaussovelluskehystä. Sovelluksen testikattavuus paranee jatkuvasti normaalin kehityksen ohella. Lisäksi kehitysryhmäämme kuuluu testaaja, joka käy kaikki uudet ja muuttuneet ominaisuudet läpi ennen niiden julkaisua tuotantoon. Näin saadaan varmistettua, että kaikki toimii oikein ja laatu pysyy hyvänä.

Tällä hetkellä ReSharperin lisäksi ei käytetä muita työkaluja, jotka valvoisivat koodin laatua, kehitysryhmän käytäntöjä tai auttaisivat automatisoimaan virheiden korjausta tai katselmointia. ReSharper on reaaliaikainen koodinanalysointityökalu, joka on jo asennettu suurimmalle osalle kehittäjistä. Se on käytössä vain vakioasetuksilla, joka ei vastaa kehitysryhmän tarpeita eikä välttämättä ohjaa kehittäjää oikeisiin ratkaisuihin. ReSharperia käsitellään tässä työssä lisää myöhemmin. Koodikatselmointi on tällä hetkellä keskittynyt vain sovellusarkkitehdille, joka työllistää yhtä henkilöä turhan paljon ja siksi sitä haluttaisiin hajauttaa.

Sovelluksen laatua pidetään tärkeänä ja sitä halutaan parantaa sekä ylläpitää tehokkaammin kuin tällä hetkellä pystytään. Tarkoituksena on siis kartoittaa, miten kehitysryhmä haluaa nähdä sovelluksen laadun kehittyvän tulevaisuudessa ja miten tavoitteeseen voidaan päästä. Tähän päästään määrittelemällä halutut tavoitteet ja tutkimalla, mitä valituilla sovelluksilla voidaan tehdä. Lopuksi valitaan käyttöön otettavat ominaisuudet ja tehdään käyttöönotto.

3 Laatu käsitteenä

3.1 Laatu yleisesti

Termiä laatu käytetään yleensä ilman erillistä adjektiivia kuvaamaan hyvää laatua. Monet eivät pidä tarpeellisena määritellä, millainen laatu on kyseessä. Voidaan siis sanoa, että tuote on laadukas, joka tarkoittaa hyvää laatua. Laatu voidaan kuitenkin myös määritellä adjektiivilla kuten huono, hyvä tai erinomainen. [1, s. 20-26.]

Laadun mittaus voidaan tehdä ennalta sovittujen määritelmien ja standardien perusteella, joka tekee tuotteen tai palvelun laatutason määrittelyn melko suoraviivaiseksi. Nämä perustuvat yleensä vaatimuksiin, ominaispiirteisiin ja asteeseen. Vaatimukset tarkoittavat tuotteen tai palvelun vähintään toteutettavia ominaisuuksia, ominaispiirteet ovat verrattavissa kykenevyyteen ja aste tarkoittaa arvosanaa. Adjektiiveilla määriteltyjen laatutasojen määrittely on puolestaan hyvin vaikeaa ja epämääräistä. On epäselvää missä menee eri tasojen raja ja kuka määrittää rajat sekä tuotteen saaman tason. [1, s. 20-26.]

Tuotteen tai palvelun määritelmiin voi kuulua esimerkiksi

- komponentit, materiaalit, mitat ja testaukseen liittyvät tiedot
- käyttötarkoitus tai käyttökohde
- rajoitteet
- valmistusprosessi

- turvallisuusvaatimukset. [1, s. 20-26.]

Näitä määritelmiä mittareina käyttämällä voidaan määrittää tuotteen tai palvelun laatu huomattavasti helpommin. [1, s. 20-26.]

Tietyn osaamistason tai laadun valvontaan voidaan käyttää esimerkiksi sertifikaatteja. Voidaan esimerkiksi määritellä, että yrityksen työntekijöillä pitää olla tietyt sertifikaatit suoritettuna voidakseen tuottaa tarpeeksi laadukasta koodia. Sovelluksen laadun varmistaminen taas voidaan toteuttaa ulkopuolisella katselmoinnilla, jolla saadaan puolueeton ammattilaisen määrittelemä arvio. [1, s. 20-26.]

Nykyhetkenä sovelluskehityksessä suurella osalla yrityksistä ei ole todennettavissa olevaa tietoa heidän saavuttamastaan laatutasosta. Monilla kehitysryhmillä ei ole selkeitä laatutavoitteita, eikä erillistä laadunvarmistus- ja testausryhmää. Näissä tapauksissa kehitysryhmä hoitaa siis laadunvarmistuksen, jolloin se jää vähemmälle ja ei takaa korkeaa laatua. [1, s. 20-26.]

Monesti voidaan tulkita, että tuotteen valmistajalle laatu tarkoittaa tuotteen olevan määritelmien ja standardien mukainen. Palvelun tarjoajalle laatu taas tarkoittaa määräaikojen kunnioittamista ja palvelun toimittamista määritelmien ja standardien mukaisesti. [1, s. 20-26.]

3.2 Laatu sovelluskehityksessä

Myös sovelluskehityksessä laatua voidaan mitata monilla erilaisilla mittareilla ja eri henkilöiden kannalta. On olemassa sääntöjä, standardeja, tyylejä esimerkiksi tiedostojen ja muuttujien nimeämiseen sekä koodin ulkoasuun. Sovellus voidaan tulkita esimerkiksi paljon virheitä sisältäväksi ja epävakaaksi tai lähes virheettömäksi ja vakaaksi.

Asiakkaan eli loppukäyttäjän kannalta tuote on laadukas, kun siinä ei ole virheitä, se toimii luotettavasti, se on helppokäyttöinen ja toimii, kuten on luvattu. Asiakkaat eivät usein ole kiinnostuneita itse koodin tai toteutuksen laadusta, kunhan sovellus toimii, kuten on luvattu. Siksi tilataan usein tuote, jossa ei ymmärretä tai haluta rahallisista syistä ottaa huomioon sovelluksen laadunvarmistusta ja tietoturvaa, joka saattaa

lopulta osua omaan nilkkaan virheellisyyden, huonon päivitettävyyden tai puutteellisen tietoturvan takia. Ajatellaan esimerkkinä tilannetta, jossa säästetään sovelluksen tietoturvassa ja halutaan vain minimitoiminnallisuus mahdollisimman halvalla. Rahaa säästettäessä ei kuitenkaan oteta huomioon mahdollisia vahinkoja ja haitallisia vaikutuksia tuotteen imagoon. Huono tietoturva mahdollistaa väärinkäytön ja esimerkiksi tietomurrot. Tämä saattaa johtaa yrityksen arvostuksen laskemiseen ja tulla kalliimmaksi kuin tietoturvaan alun perin panostaminen olisi tullut. Kyseessä on lähes uhkapelaamiseen verrattavissa oleva tilanne, jossa asiakkaan pitää osata päättää, panostetaanko laadunvarmistukseen ja tietoturvaan ennalta vai jätetäänkö se huonolle tasolle ja toivotaan, että suuria tietomurtoja ei tapahdu. [1, s. 20-26.]

3.2.1 Laadun mittareita

Sovelluskehittäjien kannalta laatua voidaan taas mitata yhdenmukaisella koodaustyylillä, jossa noudatetaan hyviksi todettuja standardeja ja sääntöjä. Tämä parantaa koodin luettavuutta, ymmärrettävyyttä ja selkeyttä. On myös helppoa siirtyä työskentelemään toisen henkilön tekemään sovelluksen osaan, jos koodi on aina samanlaista ja helposti ymmärrettävissä. On myös kannattavaa suunnitella sovelluksen arkkitehtuuri hyvin helpon sovelluskehityksen, modulaarisuuden, uudelleenkäytettävyyden ja päivittämisen kannalta. [1, s. 20-26, 309-322.]

Sovelluskehityksessä laadun saavuttamisessa on tärkeää

- ylläpidettävyys – Sovellusarkkitehtuuri ja toteutus on suunniteltu niin, että ominaisuuksia on helppo ylläpitää, muokata, poistaa ja lisätä. Jotta koodi olisi ylläpidettävää, on sen oltava helposti luettavissa ja ymmärrettävissä. Siksi on suositeltavaa käyttää kehitysryhmässä standardin mukaista tai vähintään yhdenmukaista kehitystyyliä ja nimeämissääntöjä.

- joustavuus – Joustavuus tarkoittaa mukautuvuutta uusiin tilanteisiin ja käyttötarkoituksiin, mihin sovellusta ei ollut alun perin tarkoitettu. Joustavuus on saavutettavissa kovakoodauksen välttämisellä ja sovelluksen parametrisoinnilla.

- tehokkuus – Tehokkuudella tarkoitetaan mahdollisimman vähäistä resurssien käyttöä ja nopeata suoritusaikaa. Sovellukset ajetaan palvelimilla tai

työasemilla, joissa on rajoitettu määrä muistia ja prosessoritehoa, jotka halutaan hyödyntää mahdollisimman tehokkaasti. Tähän päästään hyödyntämällä uusimpia ja nopeiksi, sekä kevyiksi todettuja teknologioita ja ratkaisuja, sekä noudattamalla standardien mukaisia tai hyviksi todettuja ratkaisuja. Tämä kuuluu hyvään sovellusarkkitehtuurin suunnitteluun.

- modulaarisuus – Modulaarisuudella tarkoitetaan sovelluksen jakamista moduuleihin, joista sovelluskokonaisuus muodostuu. Esimerkiksi yksi toiminnallisuus voi olla yksi moduuli. Näin pystytään helposti ja vaivattomasti lisäämään, poistamaan ja muokkaamaan yksittäisiä ominaisuuksia isossa sovelluksessa.

- uudelleenkäytettävyys – Sovelluskehityksessä tullaan monesti tilanteeseen, jossa yhtä toiminnallisuutta halutaan käyttää useammasta paikasta. Jos esimerkiksi kaksi moduulia tarvitsee samaa ominaisuutta, on kannattavaa siirtää ominaisuus niin sanotulle yleiselle alueelle, että se on käytettävissä kummassakin moduulissa. Näin vältetään saman koodin uudelleenkirjoittamista. Toiminnallisuuden kirjoittaminen kannattaa tehdä mahdollisimman yleismuotoisesti, jotta se on helposti käytettävissä eri tarkoitusperiin.

- luettavuus – Tuotetun koodin pitää olla helposti luettavaa ja ymmärrettävää ylläpidettävyyden takia. Tähän päästään käyttämällä kehitystiimin yhteisiä, jonkin standardin mukaisia tai muuten hyviksi todettuja sääntöjä koodin muotoilussa sekä nimeämissäännöissä.

- testattavuus – Sovellus on testattavissa, jos jokainen sen osa tai yksittäinen toiminto on erikseen testattavissa. Automaattisilla testeillä pystytään huomaamaan jokin rikki mennyt sovelluksen osa helposti ja paikallistaa. Näin voidaan korjata virhe nopeasti. Ilman automatisoituja testejä virhe voisi olla kauan huomaamatta. [1, s. 38-41; 2, s. 241-257.]

3.2.2 Laatuun liittyviä termejä

Sovelluksen tekijät ovat ihmisiä ja ihmiset tekevät usein virheitä. Sovellusta kehittäessä tehdään siis myös virheitä, jotka yleensä näkyvät väärin toimivana tai virheilmoituksia antavana sovelluksena. Näitä virheitä kutsutaan yleensä ”bugeiksi”, ja ne johtuvat

virheellisestä logiikasta sovelluksen koodissa. Suurin osa bugeista karsiutuu pois jo kehitysvaiheessa, kun kehittäjä ”debuggaa” eli läpikäy koodia ja toiminnallisuutta tekemisensä ohella ja sen jälkeen. Uusille ominaisuuksille tehdään yleensä myös yksikkötestejä, joita tehdessä virheet yleensä löytyvät. Jos bugi syntyy koodiin testien tekemisen jälkeen, se voidaan havaita testejä ajettaessa seuraavan kerran, joka on yleensä sidottu versionhallintaan viennin yhteyteen. Lopuksi kehitysryhmällä on suositeltavaa olla vielä testaaja tai testaajia, jotka käyvät toiminnallisuuden läpi seuraten alkuperäisiä vaatimuksia sekä määritelmiä ja raportoivat tarvittaessa puutteista sekä virheistä. [1, s. 41-43.]

Vikasietoisuudella tarkoitetaan virheiden mahdollisuuden huomioon ottamista, jossa koodiin tehdään vikasietoisia ratkaisuja, jotka estävät virheen tapahtuessa koko sovelluksen käytön keskeytymistä. On otettava esimerkiksi huomioon tilanteet missä verkkoyhteydet tai palvelimet eivät toimi odotetusti, joka voi aiheuttaa koko sovelluksen kaatumisen, jos sovellusta ei ole tehty vikasietoiseksi. [1, s. 41-43.]

Puute on sovelluksesta puuttuva toiminnallisuus tai ominaisuus, joka alun perin piti löytyä sovelluksesta. Tämä on osittain verrattavissa bugiin, mutta puute ei estä sovelluksen toimintaa vaan tällöin sovellus vain poikkeaa alkuperäisistä vaatimuksista, eikä toimi täsmälleen halutulla tavalla. [1, s. 41-43.]

4 Laadunvarmistus sovelluskehityksessä

Tässä luvussa käydään läpi markkinoilla olevia ratkaisuja ja sovellusvaihtoehtoja laadunvarmistukseen sovelluskehityksessä. Sovellusten koodin laadun, tietoturvan ja lisenssialkuperän analysointiin on markkinoilla monia eri työkaluja. Koodin määrän, laadun, testikattavuuden ja arkkitehtuurin analysointiin löytyy monia analysointisovelluksia kuten

- CodeIt.Right
- CodeRush
- JustCode

- NDepend
- ReSharper
- Kalistick.

Osa edellä mainituista sovelluksista on reaaliajassa korjausehdotuksia ja automatisoituja korjaustoimenpiteitä mahdollistavia työkaluja, kuten ReSharper. Osa taas on staattisen analyysin työkaluja, kuten NDepend, joka analysoi lähdekoodia kokonaisuutena. Lisäksi on olemassa työkaluja, jotka mahdollistavat kummankin toiminnallisuuden samanaikaisesti, kuten CodeIt.Right. Tästä voidaankin todeta, että CodeIt.Right on hyvältä vaikuttava vaihtoehto, sillä se sisältää toiminnallisuuden kummaltakin puolelta [3.]. Yrityksemme toinen sovelluskehitysryhmä on kuitenkin hankkinut edellä mainituista sovelluksista ReSharper- ja NDepend-lisenssit ja ryhmien yhdenmukaisen toiminnan johdosta päätimme käyttää samoja työkaluja.

4.1 Staattisen analyysin työkalu NDepend

NDepend mahdollistaa koodin staattisen analyysin, raporttien ja statistiikan helpon generoinnin ja analysointimahdollisuudet. Se sisältää vakiona yli 200 sääntöä ja erilaisia mittareita, joilla voidaan mitata koodin laatua. Jo olemassa olevia sääntöjä voi muokata ja on myös mahdollista luoda uusia sääntöjä käyttäen saatavilla olevia mittareita. Analyysiraporttien luomisen voi automatisoida TFS:sään viennin yhteyteen. Jokaisen versiohallintaan laitetun muutoksen jälkeen sovellus siis käännetään, testit ajetaan ja NDepend-raportti luodaan. Kaikki on siis mahdollista automatisoida ja sitoa normaaliin ryhmän kehitystoimintaan, mikä tekee raporttien luonnista vaivatonta eikä työaikaa ei mene hukkaan.

4.1.1 Asennus

Lisenssimäärän ja -tarpeiden takia NDepend-työasemaversio asennettiin vain yhdelle työasemalle, jossa sen testaus ja käyttöönotto suoritettiin. Työasemaversion asentaneelle kaikki ominaisuudet ovat käytettävissä suoraan Visual Studion sisältä NDepend-lisäosaa hyödyntäen.

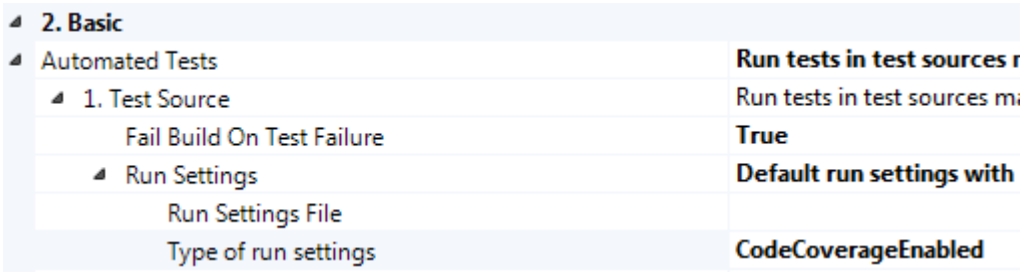
Asennuksen jälkeen on luotava NDepend-projekti ja liitettävä se analysoitavaan Visual Studio -toteutukseen, jonka jälkeen raportin luominen onnistuu yhdellä napin painalluksella. Vakiona käytössä on yli 200 sääntöä, joita voi helposti kytkeä päälle ja pois tai muokata halutessaan. Lisäksi voi tehdä omia sääntöjä halujensa mukaan.

NDepend-palvelinversio laitettiin puolestaan palvelimelle, joksi valitsimme kääntöpalvelimemme, sillä NDependin käsittelemät tiedot löytyvät kyseiseltä palvelimelta ja NDependin toiminta sidotaan jatkuvanintegraation kääntöprosessiin. Palvelinpuoli ei vaadi asennusta, vaan zip-paketti vain puretaan palvelimelle haluttuun kansioon. Itse käyttöönotto vaatii NDependin ajamisen sitomisen jollain tavalla kääntöpalvelimen kääntöprosessiin, johon löytyy monia mutta valitettavasti vain epävirallisia ratkaisuja.

```
1  $ps = new-object System.Diagnostics.Process
2  $ps.StartInfo.Filename = "C:\NDepend\NDepend.Console.exe"
3  $ps.StartInfo.Arguments = [string]::Format("{0}/Raisu4/Raisu4.ndproj", $Env:TF_BUILD_SOURCESDIRECTORY)
4  $ps.start()
5  $ps.WaitForExit()
```

Kuva 1. NDependin ajamiseen käytetty PowerShell-skripti

Lopulliseksi toteutukseksi päädyttiin ratkaisuun, jossa toteutettiin yksinkertainen PowerShell-skripti (Kuva 1), joka sijoitettiin VS:n toteutuksen juurihakemistoon versiohallinnassa. Tämä skripti laitettiin TFS:n jatkuvan integraation kääntöasetuksiin ajettavaksi testien ajon jälkeen. Skriptin ainut toiminta oli NDependin ajaminen projektitiedostoa käyttäen oikeasta sijainnista.



Kuva 2. Koodin kattavuustiedoston luomisen määrittäminen päälle TFS-kääntöasetuksista

Lisäksi on huomioitava, että raportin luomisessa testikattavuuden analysointi vaatii coverage- eli kattavuustiedoston, jota ei vakiokäännösasetuksilla luoda. Se on määriteltävissä luotavaksi kääntöpalvelimen asetuksista kuvan Kuva 2 lailla.

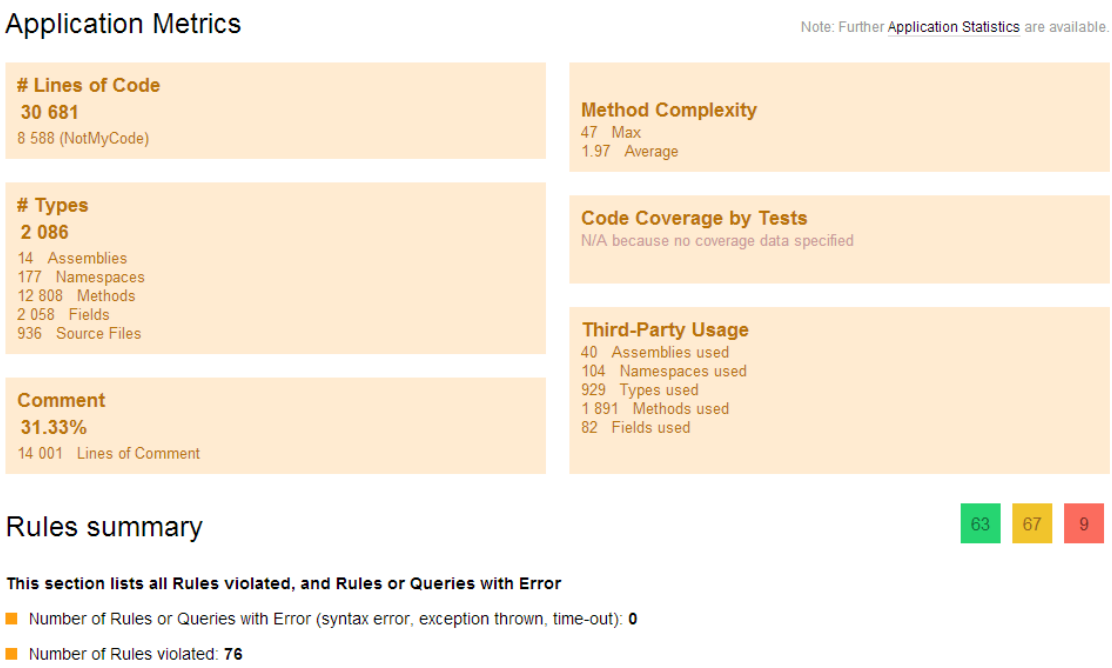
Tarkemmat ohjeet asetuksen vaihtamiseen ovat liitteessä 1. On myös määriteltävä NDepend-projektin asetuksiin sijainti, mistä coverage-tiedosto löytyy. Tämän jälkeen raporttia luodessa otetaan myös aina huomioon testikattavuus ja testeihin liittyvät analysointisäännöt.

NDepend-raportin luonti sidottiin siis jatkuvan integraation kääntöprosessiin ja uusin raportti on nyt luettavissa sisäverkosta selaimella kääntöpalvelimen ja projektin nimellä eli osoitteesta http://palvelimennimi/projektinnimi. Tässä tapauksessa siis http://grahki-build01/GD.

NDepend-käyttöä suunniteltiin alustavasti myös koodikatselmointia helpottamaan ”Review”-merkintöjen tekemiseen ja näyttämiseen, mikäli se olisi mahdollista. Ideana oli, että kuka tahansa kehittäjistä voisi merkata koodipätkän, metodin tai luokan ”// Review: tekstiä” tyyppisesti katselmoitavaksi. Tämä ei kuitenkaan osoittautunut mahdolliseksi, sillä mukautettuja sääntöjä tehdessä ei voi analysoida kommenttirivien tekstisisältöä, joten kaikkien review-merkintöjen hakevaa sääntöä ei voi toteuttaa.

4.1.2 Raporttien analysointi

Heti asennuksen jälkeen otettu raportti ennen kattavuustiedoston luomista antaa hieman kuvaa, mitä työkalulla voidaan tehdä, mukaan lukematta testeihin liittyvää statistiikkaa ja kattavuutta. Raporttia voi analysoida, joko automaattisesti luodussa Html- ja JavaScript-verkkosivumuodossa, tai suoraan Visual Studion lisäosan kautta. Lisäosa vaatii kuitenkin lisenssin toimiakseen, joten kannattavampaa on pitää tämä vain yhdellä työasemalla, sillä raporttien lukeminen onnistuu myös ilman lisäosaa suoraan selaimesta. Selain- ja lisäosaraportin erona ovat lähinnä käyttöliittymän ulkoasu ja ainoastaan lisäosa mahdollistaa interaktiiviset ominaisuudet kuten Dependency Graph, Dependency Matrix, Treemap Metric View ja Abstractness vs Instability, joita voidaan käyttää tarkempaan analysointiin. Edellä mainittujen ominaisuuksien toimintaa ja tulkitsemista käydään läpi seuraavaksi.

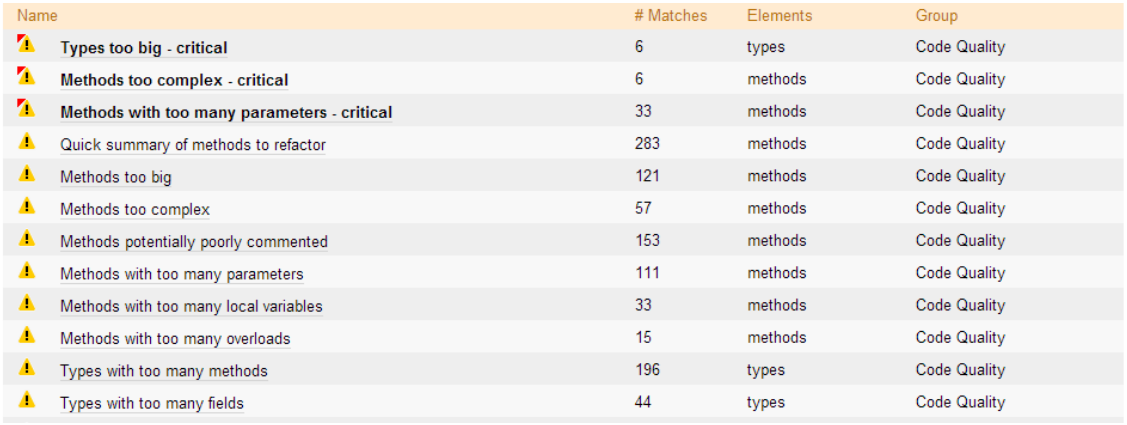


Kuva 3. NDepend-raportti 1. Sovelluksen mittareita ja yhteenveto virheistä

Verkkosivuraportin pääsivulla oleva listaus projektin yleistiedoista (kuva 3) antaa hyvin kuvan projektin laajuudesta ja tilasta. Siitä voidaan nähdä, että projektin omaa koodia on 30 681 riviä. Projektissa käytettyjen ulkopuolisten kirjastojen sekä automaattisesti luodun koodin yhteisrivimäärä on kerrottu erikseen 8 588 (NotMyCode).

NDependin käyttämä rivimäärä ei ole fyysinen vaan looginen. Sitä ei siis lasketa suoraan lähdekooditiedostoista, joka antaisi epätarkan ja vertailukelvottoman arvon. NDepend laskee rivien määrän PDB-tiedostoista käyttäen logiikkaa, jonka ansiosta laskettu arvo ei ole kielikohtainen ja on näin verrattavissa muilla kielillä kirjoitettuihin tiedostoihin. Ainoastaan koodin ajamiseen vaikuttava koodi lasketaan eli rajapintoja eikä abstrakteja metodeja oteta mukaan laskuun. Myöskään koodin ulkoasulla ei ole merkitystä, joten tottumukset ja käytännöt kuten aaltosulkujen sijainti eivät vaikuta saatuun arvoon. [4.]

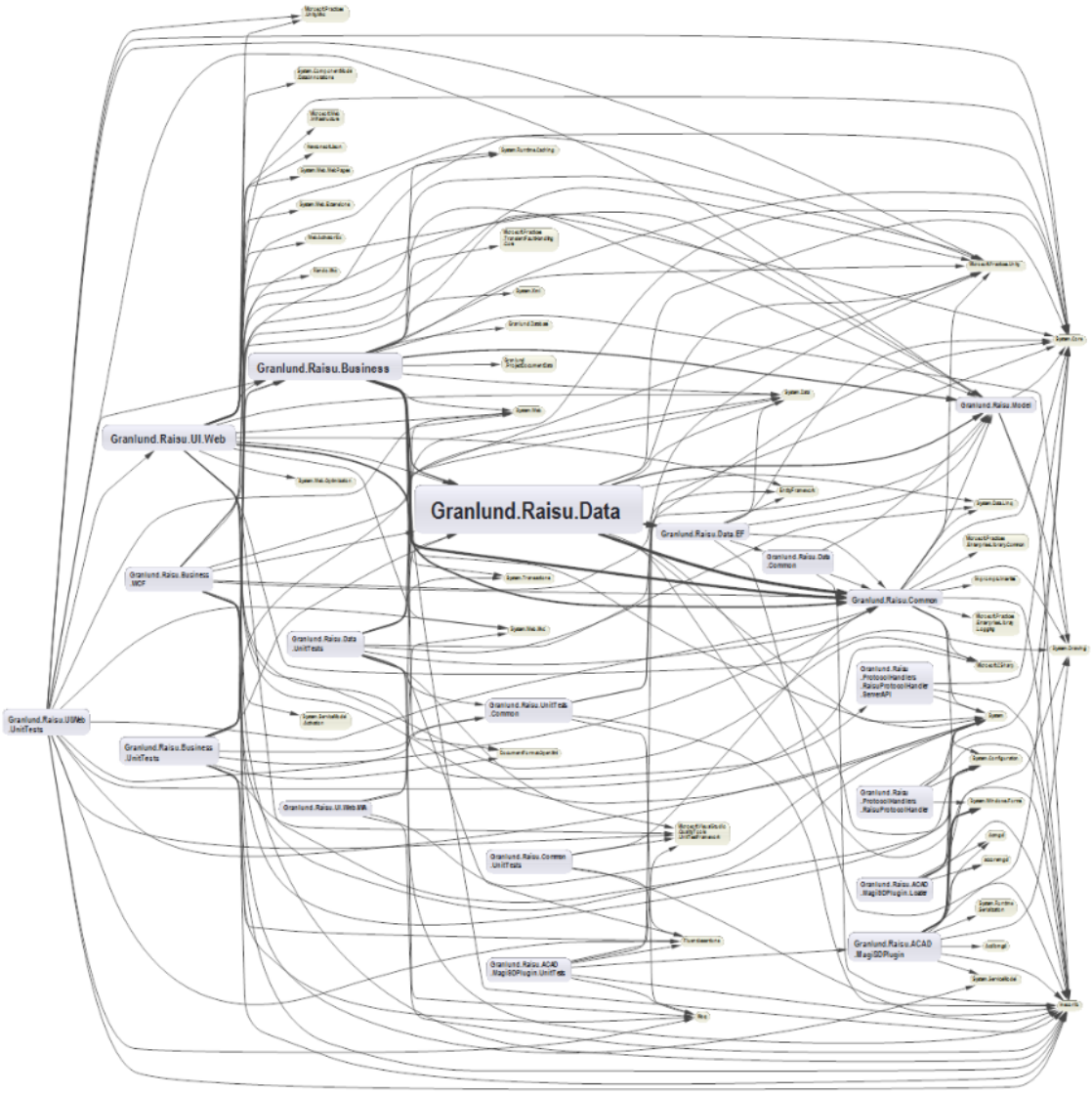
Voidaan myös havaita, että kommenttien määrä on hyvin matala eli vain 31,33 %. Ja kuten myös aiemmin todettiin, testien kattavuutta ei voida ensimmäisessä raportissa vielä nähdä, sillä kattavuustiedostoa ei ole määritelty.



Name	# Matches	Elements	Group
Types too big - critical	6	types	Code Quality
Methods too complex - critical	6	methods	Code Quality
Methods with too many parameters - critical	33	methods	Code Quality
Quick summary of methods to refactor	283	methods	Code Quality
Methods too big	121	methods	Code Quality
Methods too complex	57	methods	Code Quality
Methods potentially poorly commented	153	methods	Code Quality
Methods with too many parameters	111	methods	Code Quality
Methods with too many local variables	33	methods	Code Quality
Methods with too many overloads	15	methods	Code Quality
Types with too many methods	196	types	Code Quality
Types with too many fields	44	types	Code Quality

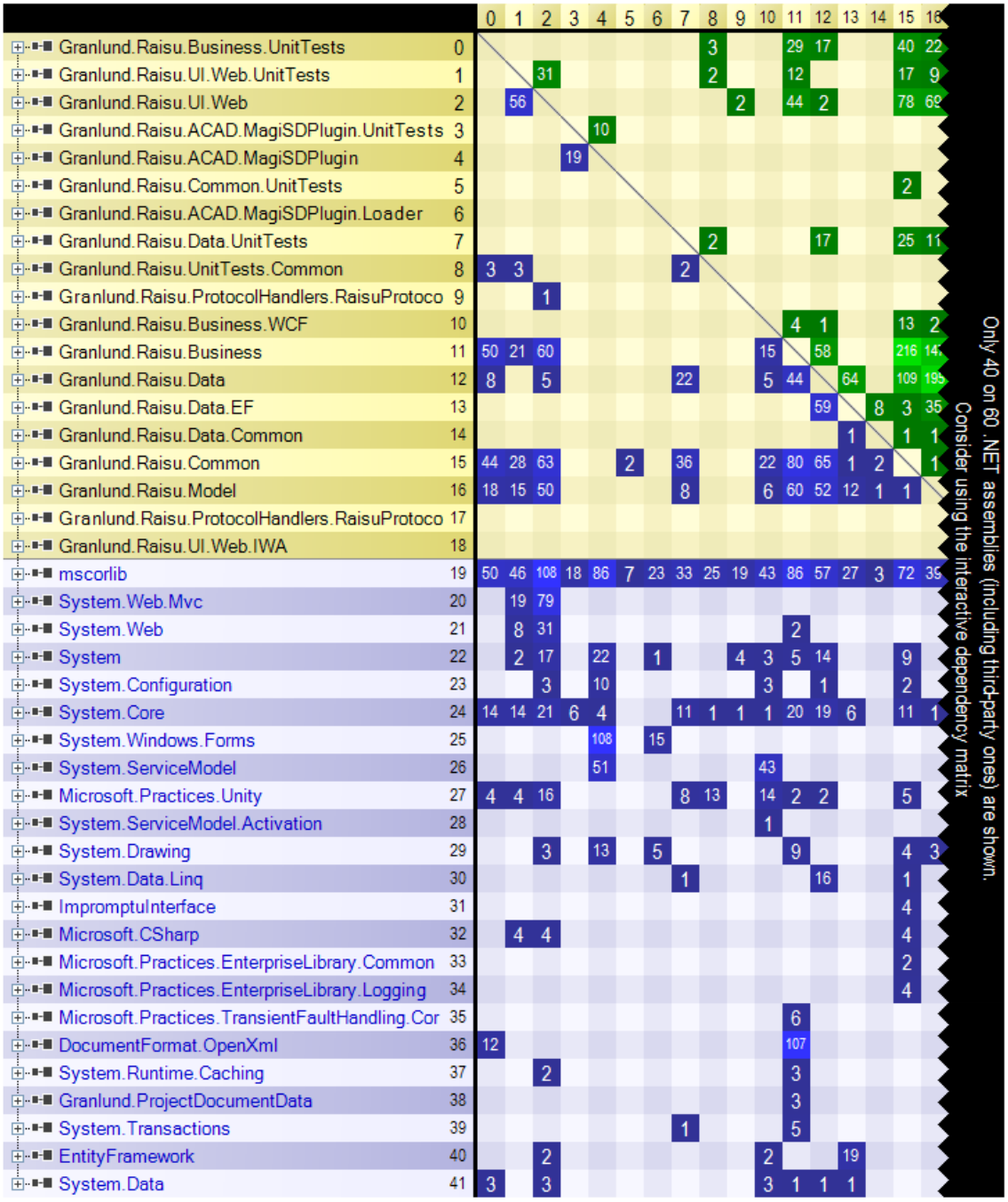
Kuva 4. Otos rikotuista säännöistä

Raportin pääsivulla voidaan myös nähdä yhteenveto rikotuista säännöistä yllä olevan kuvan 4 mukaisesti. Raporttia tehdessä sääntöjä ei ollut vielä muokattu, joten saadut tulokset ovat vakiosäännöillä saatu. Tämä ei kuitenkaan anna vielä hyvää kuvaa sovelluksen tilasta, sillä säännöt on suositeltavaa muokattava GD:n tavoitteiden mukaisiksi.



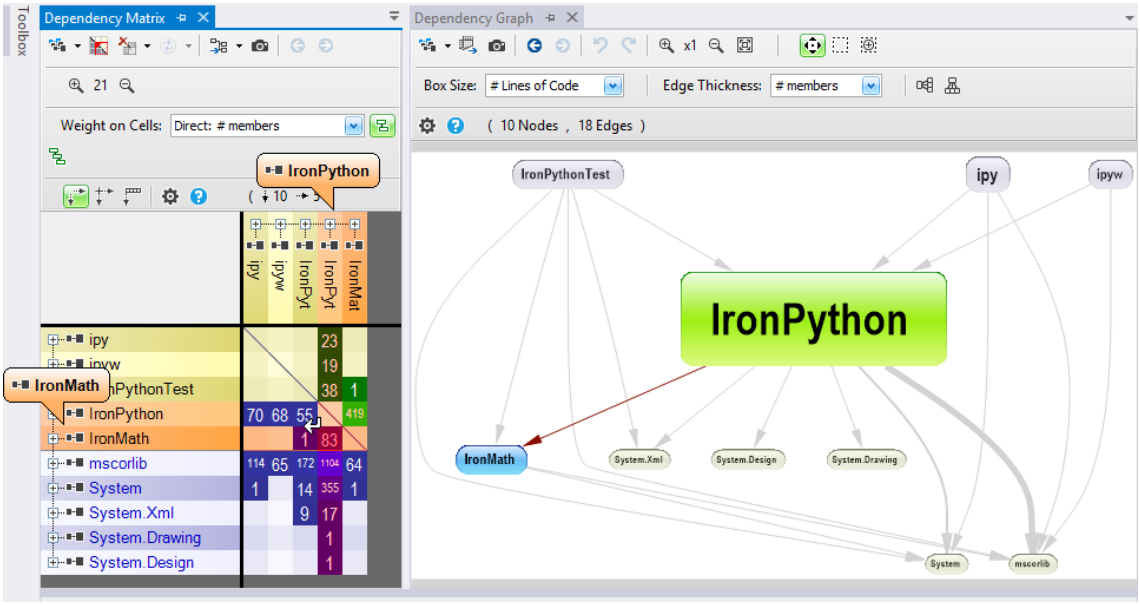
Kuva 5. Riippuvuuskaavio (Dependency Graph)

Kuvassa 5 voidaan nähdä automaattisesti luodun riippuvuuskaavion verkkoraporttiversio. NDependin työasemaversion interaktiivisella riippuvuuskaaviolla voi kuitenkin tutkia rakennetta huomattavasti tarkemmin. Tarkkailemaan pääsee luokkien periytymistä ja kutsujen kulkemia polkuja visuaalisesti. Kuten kuvasta voidaan huomata, on isommissa sovelluksissa lähes mahdotonta hyödyntää riippuvuuskaaviota, sillä siitä tulee nopeasti iso ja vaikeasti luettava. NDepend tuo tähän kuitenkin ratkaisuna heidän kehittämän uuden riippuvuusmatriisi (Dependency Matrix) kuvaajan, joka esitellään seuraavaksi. [5.]



Kuva 6. Riippuvuusmatriisi (Dependency Matrix)

NDependin riippuvuusrakennematriisi eli DSM (Kuva 6) on kompakti tapa kuvata ja selailla komponenttien välisiä riippuvuuksia. Se eroaa normaalista luokkakaaviosta, jota yleensä käytetään. Se mahdollistaa myös ison sovelluksen rakenteen tutkimisen selkeämmin kuin normaali luokkakaavio. [5.]



Kuva 7. Riippuvuusmatriisin ja riippuvuuskaavion ero

Matriisin otsikot kuvastavat kaavion laatikoita ja matriisin arvolliset solut kuvastavat nuolia, kuten yllä olevasta kuvasta 7 voidaan nähdä. DSM ei ole yhtä intuitiivinen kuin normaali luokkakaavio, mutta se on suunniteltu isojen kaavioiden näyttämiseen, sillä kun normaalia kaaviota käytettäessä ylitetään noin 20 luokkaa, on lähes mahdotonta sisäistää kokonaisuutta tai löytää haluamaansa tietoa. DSM on siis hyödyllinen ainakin isojen kaavioiden ja sovelluksien analysoinnissa. [5.]

DSM:n arvollisissa soluissa olevat numerot kuvastavat liitännäisyyden voimakkuutta eli kuinka monta metodia, muuttujaa, tyyppiä tai nimiavaruutta on liitännäisyydessä. Kaikkiin liitännäisyyksiin voi mennä sisälle valitsemalla solun ja tutkia sitä tarkemmin. [5.]

Matriisin avulla voidaan myös havaita erilaisia rakennekuvioita helposti vain nopeasti silmäilemällä kuvaa 6. Näitä ovat

- Kerrosrakenne (layered structure / acyclic structure). Kun matriisin vasemmalla alhaalla oleva kolmio muodostuu kaikista sinisistä soluista, ja ylhäällä oikealla oleva kolmio muodostuu kaikista vihreistä soluista, on rakenne täydellinen kerrosrakenne. Tällöin rakenteessa ei ole riippuvuussyklejä. Voidaan siis tulkita kuvasta (Kuva 6), että GD:llä on täydellinen kerrosrakenne. [5.]

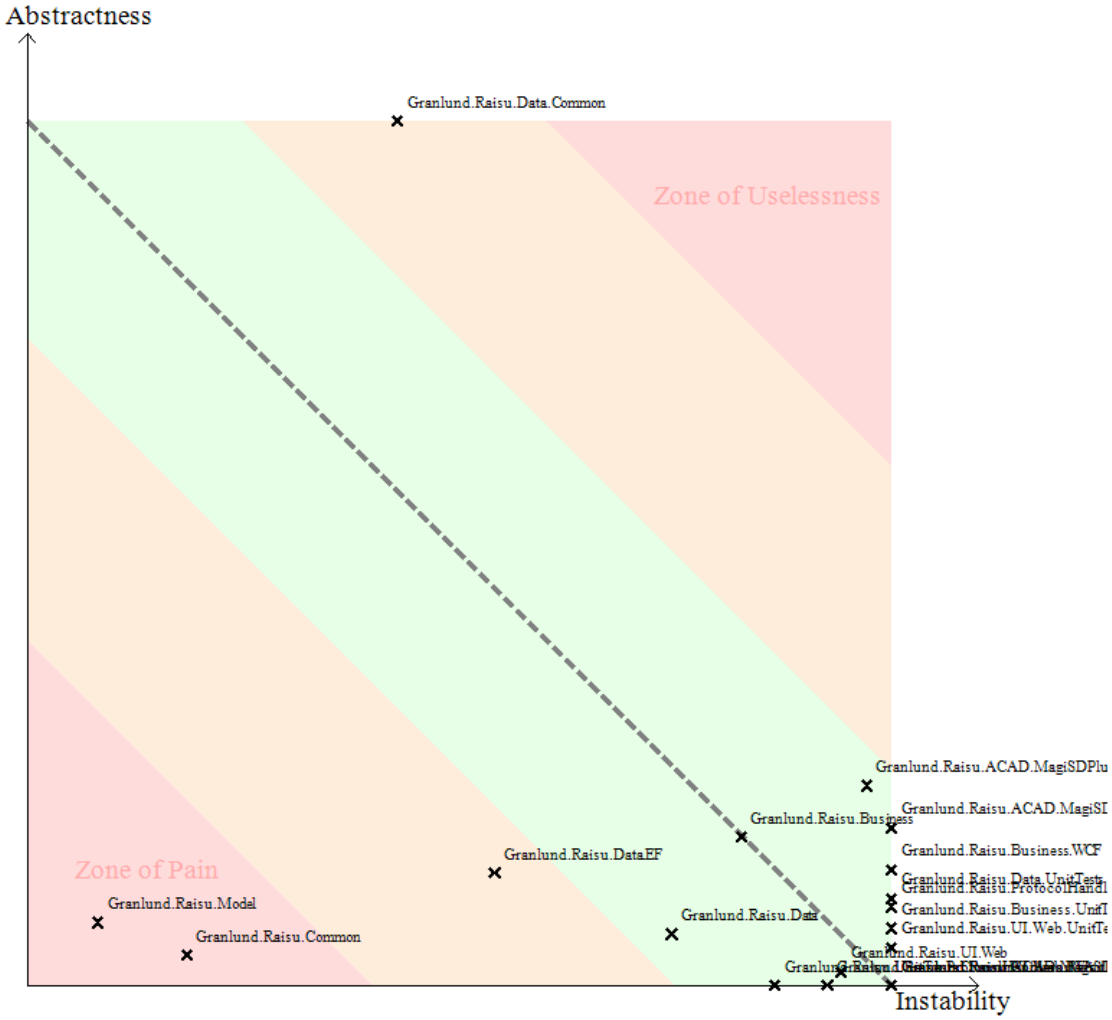
- Riippuvuussyklit (eng. dependency cycle) on merkattu DSM:ssä neliönmuotoisella punaisella reunaviivalla. Näitä ei GD:llä kuvasta 6 päätellen ole. [5.]

- Korkea yhteenkuuluvuus – vähäinen kytkentä (eng. high cohesion - low coupling). Yleinen idea nykypäivänä, jossa on komponentin sisällä korkea yhteenkuuluvuus, mutta komponenttien välillä vähäinen kytkettävyys. Yhteenkuuluvuus tarkoittaa siis riippuvuutta toistensa välillä. Viistossa matriisin leikkaavan viivan ympärille muodostuva symmetrinen neliö viittaa korkeaan yhteenkuuluvuuteen. [5; 6.]

- Liikaa vastuuta (eng. too many responsibilities). SRP eli yhden vastuun periaate määrittelee, että luokalla ei pitäisi olla enempää kuin yksi syy muuttua. Luokan ei siis pitäisi olla käytössä liian monessa paikassa. Tällaiset luokat näkyvät DSM:ssä niin, että sarakkeella on paljon sinisiä soluja ja rivillä paljon vihreitä soluja. [5.]

- Suositut luokat (eng. popular code elements) eivät ole huono asia ja niitä on aina sovelluksessa. Ne voidaan havaita DSM:stä sarakkeina, joissa on paljon vihreitä soluja ja riveinä missä on paljon sinisiä soluja. GD:ssä suosituimmat luokat löytyvät kuvasta 6 katsottuna siis Common- ja Model-nimiavaruuksien alta, joista Common on vanhan arkkitehtuurin Modelia vastaava nimiavaruus, joka on poistumassa. Model sisältää luokkia, joita käytetään melkein kaikkialla sovelluksessa tiedon säilömiseen ja käsittelyyn. [5.]

- Toisistaan riippuvaisuus (eng. mutually dependent). Kahden luokan välinen riippuvaisuus voidaan nähdä valitsemalla yksittäinen solu tutkittavaksi. Luokka on riippuvainen toisesta, jos se käyttää toista luokkaa. Luokat ovat siis toisistaan riippuvaisia, jos ne käyttävät toisiansa. [5.]



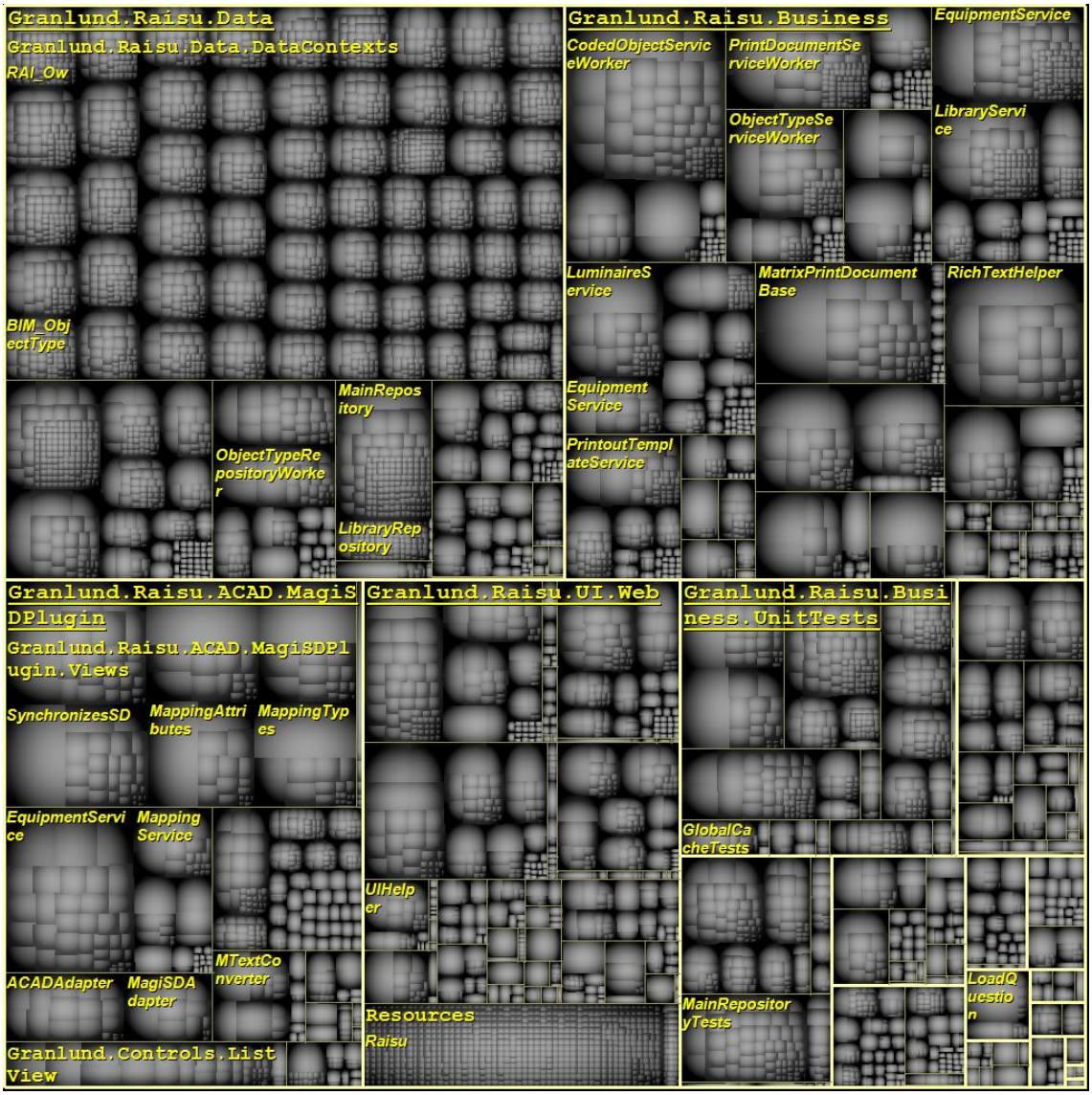
Kuva 8. Abstractness vs Instability

Abstractness vs Instability -kuvaaja (Kuva 8) antaa yleiskuvan sovelluksen tilasta. Yksinkertaisuudessaan punainen alue tarkoittaa huonoa ja vihreä alue tarkoittaa hyvää. Pystysuunnassa mitataan kuinka abstrakti (eng. abstractness) nimiavaruus on. Sivuttaissuunnassa mitataan epävakautta (eng. instability). [7.]

Abstraktiudella tarkoitetaan sovelluksen joustavuutta muutoksille. Jos sovellus on toteutettu abstrakteja luokkia ja rajapintoja käyttäen, sitä on huomattavasti helpompaa muokata ja laajentaa. [4; 7; 8.]

Epävakaudella ei tarkoiteta sovelluksen kaatumisen todennäköisyyttä, vaan sen muokkauksen sietokykyä eli kuinka paljon sovellusta voidaan muokata hajottamatta sitä. Kuinka moni sovelluksen osa on siis riippuvainen kyseisestä rajapinnasta, joka rajoittaa rajapinnan helppoa muuttamista. [4; 7; 8.]

GD:n kuvaajasta voidaan havaita, että suurin osa nimiavaruuksista sijoittuu vihreälle alueelle ja epävakauden puolelle. Tämä tarkoittaa siis sitä, että sovelluksen muokkaaminen on haastavaa, sillä muutoksien tekeminen voi aiheuttaa muutostöitä ja ongelmia monella sovelluksen alueella. Myös muutama nimiavaruus sijoittuu ”Zone of Pain”-alueelle. Nämä kuitenkin ovat ainoat nimiavaruudet, jotka sisältävät DTO-luokkia, jotka ovat pelkkiä luokkamäärittelyjä, joita käytetään ympäri sovellusta. Tämä selittää, miksi ne sijoittuvat sinne. Näistä voidaan siis olla välittämättä, sillä ne eivät aiheuta mitään haittaa.



Kuva 9. Puukartta, jossa mittarina on metodien rivimäärä

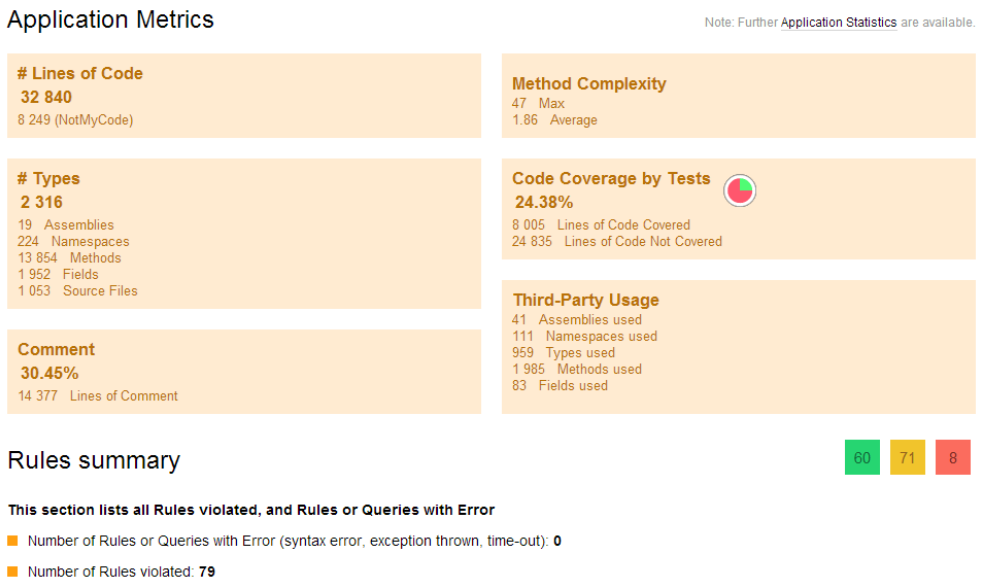
Puukartta (eng. treemap) on suunniteltu tiedon esittämiseen puumaisessa rakenteessa käyttämällä sisäkkäisiä suorakulmioita. Hierarkia muodostuu seuraavista asioista

- .NET kokonaisuudet sisältävät nimiavaruudet
- nimiavaruudet sisältävät tyypit
- tyypit sisältävät metodit ja muuttujat. [9.]

Puukartassa suorakulmiot vastaavat koodielementtejä ja tiedot ovat tarkasteltavissa eri tasoilla, joita ovat kokonaisuudet, nimiavaruus, tyyppi, metodi, muuttuja. Muodostettavaan karttaan voidaan myös valita koodimittari (eng. code metric), jossa vaihtoehtoina voidaan käyttää saatavilla olevia mittareita kuten koodin rivimäärä. CQLinq kyselyjä muokkaamalla muuttuneet arvot näkyvät puukartassa eri värillä ja näin voidaan verrata kyselyn eroavaisuutta uusilla arvoilla. [9.]

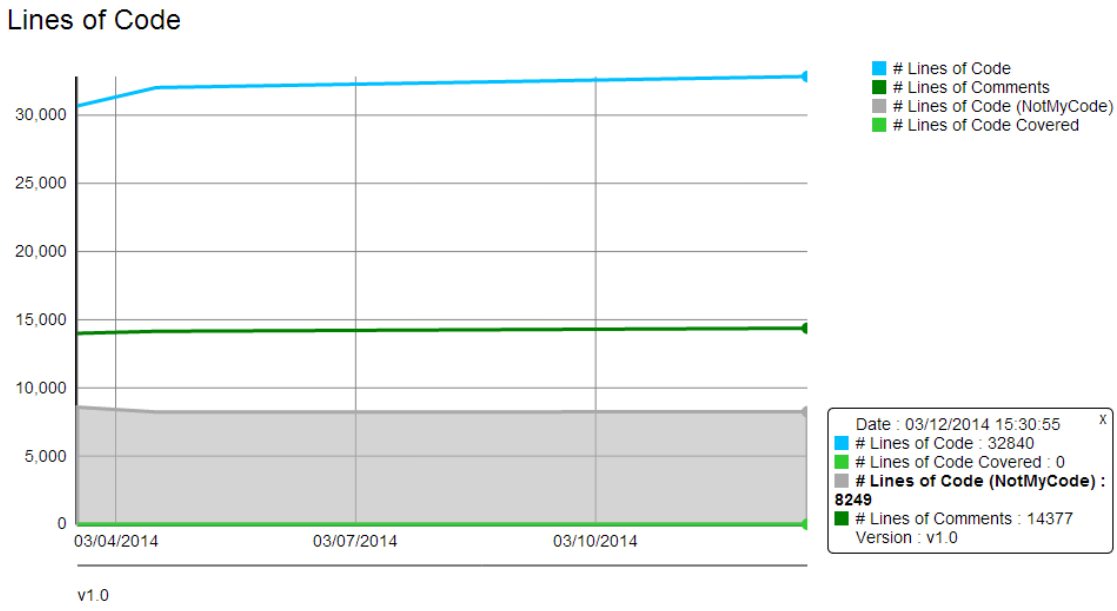
Edellä on kuva 9 puukartasta. Tasoksi on valittu metodi ja mittariksi Lines of Code eli looginen rivien määrä. Tässä tapauksessa siis jokainen suorakulmio vastaa yhtä metodia ja suorakulmion koko kertoo metodin pituuden rivimäärän perusteella. Kartasta voidaan helposti siis tulkita, mitkä osiot sovelluksesta ovat suurimpia ja missä on pisimmät metodit. Mittarina voidaan myös käyttää esimerkiksi IL Cyclomatic Complexity arvoa, jolloin suorakulmion koko viittaisi metodien monimutkaisuuteen. Edellä mainitun mittarin toimintaa avataan seuraavassa kappaleessa hieman enemmän. [9.]

GD:n kuvaajasta (Kuva 9) ei tällä hetkellä saa ihan täydellistä kuvaa sovelluksesta, sillä arkkitehtuurimuutos on vielä kesken ja vanhan arkkitehtuurin osat näkyvät kuvaajassa suuressa osassa. Kuvaajasta voidaan kuitenkin nähdä, että data eli tietokantakerros, business eli logiikkakerros ja ui, eli käyttöliittymäkerros ovat suurimmat sovelluksen osat. Vaihtamalla muihin mittareihin, kuten metodien monimutkaisuuteen, voidaan helposti nähdä lisää tietoa.



Kuva 10. Raportti 2. Sovelluksen mittareita ja yhteenveto virheistä

Uudessa raportissa (Kuva 10) voidaan nähdä myös testien kattavuus, koska kattavuustiedosto on säädetty. Tästä nähdään, että GD:n testien kattavuus on vajaan 25 %:n luokkaa, joka on todella matala. On otettava kuitenkin huomioon keskeneräinen arkkitehtuurimuutos, joka vaikuttaa saatuihin tuloksiin paljon. Suuri osa testeistä on vielä tekemättä tai siirtämättä uudelle puolelle ja vanhan puolen luokat sekä testit ovat poistamatta.



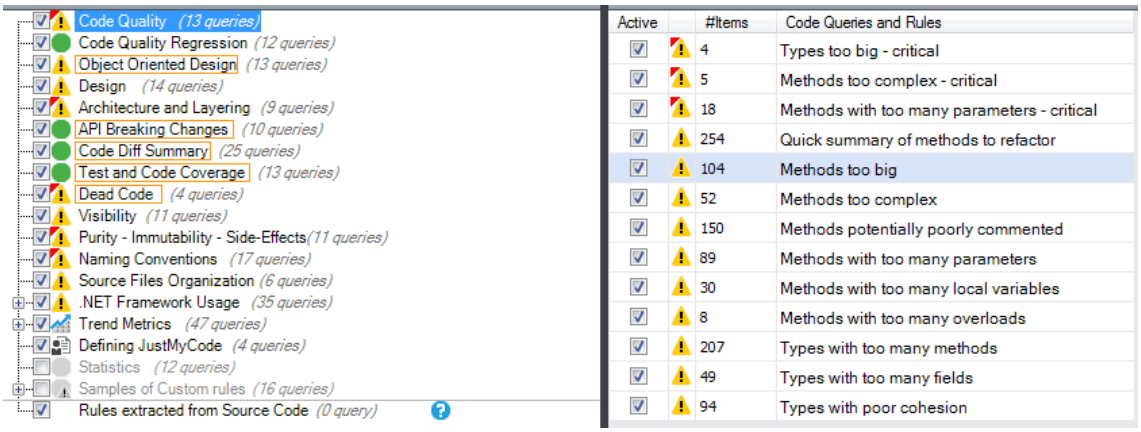
Kuva 11. Kuvaaja, jossa mittarina on rivimäärä

NDepend luo myös automaattisesti kuvaajia eri raporttien välillä tapahtuneista muutoksista käyttäen oletustrendimittareita. Näitä voidaan muokata ja lisätä samalla tavoin kuin muitakin sääntöjä.

Kuvan 11 kuvaajasta voidaan tulkita, että rivimäärä nousee hitaasti. Tämä johtuu siitä, että keskeneräiset arkkitehtuurimuutokset aiheuttavat paljon poistettua koodia, sekä toteutuksien parantelua, joka myös lyhentää koodin pituutta. Muokattujen sääntöjen voimaan ottamisen jälkeen virheiden määrä laskee huomattavasti ja alkaa näyttää tulevaisuudessa hyödyllisempää tietoa.

4.1.3 Sääntöjen valitseminen, muokkaaminen ja luonti

NDepend tarjoaa yli 200 sääntöä suoraan käyttöönotossa. Näitä sääntöjä on helppo muokata haluamikseen ja lisätä uusia sääntöjä omia käyttötapauksia varten kuvan 12 valikon kautta. Osa oletussäännöistä ei sovi suoraan GD:n tarpeisiin ja siksi niistä on valittava käyttöön tietyt ja muokata ne sopiviksi. [4.]



Kuva 12. NDepend-sääntöjen muokkaus ja valinta Visual Studio -lisäosassa

```
// <Name>Methods too big</Name>
warnif count > 0 from m in JustMyCode.Methods where
   m.NbLinesOfCode > 30
   // We've commented # IL Instructions, because with LINQ syntax, a few lines of code can compile to hundreds of IL instructions.
   // || m.NbILInstructions > 200
   orderby m.NbLinesOfCode descending,
           m.NbILInstructions descending
select new { m, m.NbLinesOfCode, m.NbILInstructions }

// Methods where NbLinesOfCode > 30 or NbILInstructions > 200
// are extremely complex and should be split in smaller methods.
// See the definition of the NbLinesOfCode metric here
// http://www.ndepend.com/Metrics.aspx#NbLinesOfCode
```

Kuva 13. ”Liian pitkä metodi” -säännön CQLinq-kysely

Säännöt tehdään NDependin Code Query LINQ (CQLinq) -kyselykielellä, joka vastaa LINQ-kieltä. Esimerkki yhdestä yksinkertaisesta säännöstä voidaan nähdä yllä olevasta kuvasta Kuva 13. Haluttua maksimimetodin pituutta voidaan siis muokata helposti tässä tapauksessa muuttamalla NbLinesOfCode > 30 arvoa. [10.] Seuraavassa taulukossa käytetty IL eli Intermediate Language viittaa generoidusta koodista laskettuihin kutsujen määriin mittareissa. Esimerkiksi ”for” tekee kaksi ja ”foreach” tekee kolme kutsua.

Taulukko 1. Säännöissä käytettyjen mittarien nimet ja selitykset

Mittari	**Selitys**
NbLinesOfCode	Kertoo rivien määrän. Arvon mittaus vaatii PDB tiedostot. Lukee loogisen rivien määrän, ei fyysistä. Tällöin koodin tyyli ja kieli ei vaikuta saatuun arvoon eli arvo on vertailukelpoinen muiden arvojen kanssa. [11]
CyclomaticComplexity	Complexity viittaa metodin monimutkaisuuteen, jota mitataan seuraavasti: 1 + seuraavien toimintojen määrä metodissa (if | while | for | foreach | case | default | continue | goto | && | || | catch | ternary operator ?: | ??) [11] [12]
ILCyclomaticComplexity	Complexity viittaa metodin monimutkaisuuteen, jota mitataan seuraavasti: 1 + kutsuttujen toimintojen * toiminnon käyttämien IL-kutsujen määrä. Yli arvo 20 on vaikea ymmärtää, yli 40 todella vaikea ymmärtää. [11] [12]
ILNestingDepth	Arvo lasketaan IL-koodista. Kertoo kapseloitujen kontekstien määrän metodissa.
NbParameters	Kertoo metodin parametrien lukumäärän. Ref ja out parametrit lasketaan myös mukaan.
NbVariables	Kertoo muuttujien määrän metodin sisällä.
NbOverloads	Kertoo metodin ylikuormitusten määrän.

Moni mittari perustuu vain yksinkertaisten määrien, kuten parametrien tai muuttujien laskemiseen, mutta joidenkin mittarien arvo perustuu vaativampaan logiikkaan. Esimerkiksi koodin laatuosiosta löytyvä sääntö ”Types with poor cohesion” käyttää mittareita LCOM ja LCOMHS. LCOM eli Lack of Cohesion Of Methods antaa arvon luokan yhtenäisyydelle väliltä 0-1. Luokalla ei pitäisi olla enempää kuin yksi syy muuttua, jolloin luokka on yhtenäinen. LCOM HS:n arvo on välillä 0-2, ja jos arvo ylittää

yhden, on se tulkittava huolestuttavana. Kyseiset mittarit ovat yleisesti käytettyjä koodin analysointiin ja lasketaan seuraavilla kaavoilla:

$$LCOM = 1 - (\frac{(Sum(MF))}{M*F}) \quad (1)$$

$$LCOM\ HS = \frac{(M-(Sum(MF)))}{F} * (M-1), \quad (2)$$

joissa

M on metodien määrä luokassa (mukaan luetaan staattiset ja instanssimetodit, konstruktorit, get- ja set-metodit, eventit, sekä lisää- ja poistametodit)

F on muuttujien määrä luokassa

MF on metodien määrä luokassa, jotka käyttävät tiettyä muuttujaa

Sum(MF) on kaikkien luokassa olevien muuttujien MF summa

Luokka on siis yhtenäinen, jos kaikki metodit käyttävät kaikkia muuttujia ja LCOM ja LCOMHS arvo on 0. [4; 12.]

Sääntöjen muokkauksessa päätettiin, että turhaksi koetut säännöt otetaan pois käytöstä ja hyviksi todetut, joita ei tarvitse muokata jätetään käyttöön oman ryhmänsä alle. Lisäksi luotiin GD-ryhmä, jonne kopioitiin kaikki säännöt, joita muokattiin. Alkuperäisen ryhmän alla oleva sääntö taas otettiin pois käytöstä. Myös kokonaan uudet säännöt sijoitettiin GD-ryhmän alle, mutta lisäksi niille annettiin ”GD - ” -prefiksi tunnistuksen helpottamiseksi. Tarkemmin sääntöjä voi tarkastella liitteistä 2 ja 3, joissa on lueteltu asetukset ja niiden muokatut arvot tai kysely yksityiskohtaisesti.

Alla esimerkkejä muutamista muokatuista ja lisätyistä säännöistä

- GD - Obsolete types

  Sääntö etsii kaikki tarpeettomat tyypit. Näin voidaan pitää kirjaa vanhentuneen koodin määrästä ja sen muutoksista.

```
// <Name>Obsolete types</Name>
warnif count > 0
from t in JustMyCode.Types where
t.IsObsolete
select t
```

Kuva 14. Käyttämättömät tyypit NDepend-sääntö

- Methods too complex

```
// <Name>Methods too complex</Name>
warnif count > 0 from m in JustMyCode.Methods where
  m.ILCyclomaticComplexity > 30 &&
  m.ILNestingDepth > 5
  orderby m.ILCyclomaticComplexity descending,
          m.ILNestingDepth descending
select new { m, m.ILCyclomaticComplexity, m.ILNestingDepth }
```

Kuva 15. Liian monimutkaiset metodit NDepend-sääntö

	#Items	Code Queries and Rules
⚠	33	GD - Obsolete types
⚠	6	Types too big
⚠	13	Methods too complex
⚠	19	Controller actions with too many parameters - critical
⚠	90	Controller actions with too many parameters
⚠	27	Methods with too many parameters
⚠	107	Quick summary of methods to refactor
⚠	12	Methods too big
⚠	42	Types with too many methods
⚠	18	Types with too many fields
⚠	482	Don't use obsolete types, methods or fields
⚠	1	Potentially dead Types
⚠	19	Potentially dead Methods (excluding UnitTests)
⚠	17	Potentially dead Fields
⚠	8	Interface name should begin with a 'I'
⚠	93	Methods name should begin with an Upper character
⚠	98	Avoid methods with name too long (excluding UnitTests)
⚠	11	Avoid types with name too long
⚠	10	Avoid fields with name too long
⚠	63	Namespace name should correspond to file location
⚠	35	Avoid defining multiple types in a source file
⚠	10	GD - Check that all namespaces begin with Granlund.Raisu

Kuva 16. GD-ryhmän NDepend-säännöt

Suurin osa sääntöihin tehdyistä muutoksista tuli koodin laatuun liittyviin sääntöihin. Kuvasta 16 voidaan nähdä kaikki GD-ryhmän säännöt, joiden nimistä voi melko helposti päätellä, mikä niiden tarkoitus on. Kriittiset säännöt on todettu tärkeimmiksi ja ne on merkitty punaisella merkillä. Ne näkyvät raportissa eriteltynä ja ovat täten helpommin huomattavissa. Taulukosta 2 voidaan nähdä kahden satunnaisen koodin laatuun liittyvän säännön alkuperäiset ja lopulliset ehdot, jotka GD:hen otetaan käyttöön. Voidaan siis havaita pieniä muutoksia sääntöjen ehtoihin. Suurimmassa osassa sääntöjä, joissa tarkistetaan rivimääriä, haluttiin nostaa ehdon rajaa huomattavasti korkeammalle, sillä GD:n rivimäärät ovat melko suuria verrattuna oletussääntöjen arvoihin. Liitteestä 2 voidaan nähdä kaikki kategorian alle liittyvät säännöt ja niihin tehdyt muutokset, jos sääntöjä halutaan tutkia tarkemmin esimerkiksi toteutusmielessä.

Taulukko 2. Kaksi esimerkkiä koodin laatua mittaavista säännöistä ja niiden muutetuista ehdoista

Sääntö	**Alkuperäiset ehdot**	**Lopulliset ehdot** (muuttuneet = kursivoitu)
Methods too complex	ILCyclomaticComplexity > 40 && ILNestingDepth > 5	*ILCyclomaticComplexity > 30 &&* ILNestingDepth > 5
Methods too big	NbLinesOfCode > 40	*NbLinesOfCode > 100*

4.2 Jatkuvan analyysin työkalu ReSharper

ReSharper on Visual Studio -kehitysympäristöön integroitu lisäosa jatkuvan koodin analysointiin kehittäjän työn auttamiseksi. Se mahdollistaa esimerkiksi virheiden helpon havaitsemisen ja automatisoidun koodin muotoilun sekä koodin automatisoidun luomisen. Jos ReSharper havaitsee, että koodin voisi kirjoittaa optimaalisemmin tai siistimmin, se mahdollistaa vapaaehtoisen automaattisen korjauksen yhdellä napin painalluksella. Se osaa myös muotoilla koodin tiettyjen muokattavien tyylisääntöjen mukaisesti yhdellä napin painalluksella, jolla koodin ulkoasu saadaan pidettyä helposti yhdenmukaisena ja eri henkilöt saadaan tekemään samojen tyylisääntöjen mukaista koodia. ReSharper mahdollistaa myös helpon koodin luomisen kuten foreach, for, switch ja vastaavien rakenteiden luomisen, joka helpottaa ja nopeuttaa kehittäjän

työntekoa huomattavasti. Tavoitteena on luoda sopivat säännöt GD:lle ja mahdollisesti myös sisällyttää sääntöihin samoja piirteitä kuin NDependissä.

4.2.1 Asennus, asetuksien tekeminen ja jakaminen

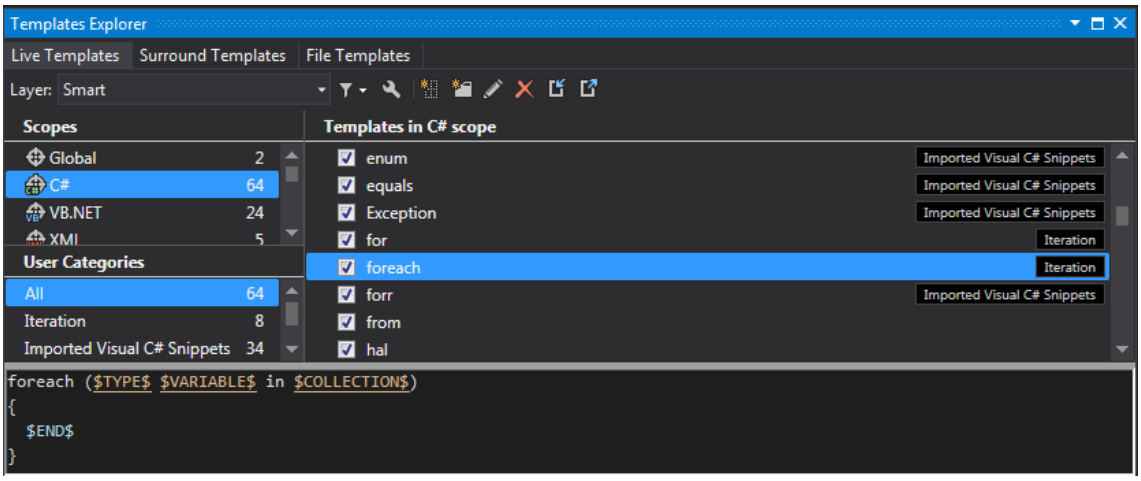
ReSharper on ollut jo jonkin aikaa käytössä sovelluskehitysryhmällämme, joten se on kaikilla asennettuna. Kaikilla on kuitenkin käytössä vain vakioasetukset ja ryhmälle haluttaisiin projektikohtaiset asetukset. ReSharperille voi määritellä projektikohtaiset ja henkilökohtaiset asetukset, mutta meidän tarkoitukseen soveltuvat parhaiten pelkät projektikohtaiset. Asetuksia voi muokata suoraan Visual Studion sisältä ReSharperin asetukset osiosta. Ne tallentuvat Raisu4.sln.DotSettings-tiedostoon toteutuksen juurihakemistoon. Tämä asetustiedosto voidaan viedä versionhallintajärjestelmään ja näin asetukset saadaan jaettua kaikkien kehittäjien välillä. Jatkossa siis myös asetuksiin tehtävät muutokset synkronoituvat helposti kaikille. [13.]

Asetuksia luodessa haluttiin yhdenmukaistaa lähdekoodin ulkoasua, ottaa pois joitakin perusominaisuuksiin kuuluvia ilmoituksia, joista ei ole hyötyä kehitysryhmällemme, sekä lisätä joitakin asioita näytettäväksi virheenä, jotta niitä ei jätettäisi helposti huomioimatta. Koodin ulkoasu on usein erilainen eri kehittäjillä, sillä sille ei ole yhtä oikeaa tyyliä. Jokaisella kielellä on yleistynyt niin sanottu standardiulkoasu, jonka lisäksi on mielipiteisiin perustuvia erilaisia ratkaisuja. ReSharper vastaa tähän mahdollistamalla ulkoasuun liittyvien sääntöjen tarkan muokkauksen kielikohtaisesti. Asioita, kuten aaltosulkujen sijainti, voidaan säätää eri tyylivaihtoehtojen kuten BSD, K&R, Whitesmith ja GNU välillä. Esimerkiksi C#-kielelle yleistyneeseen tyyliin kuuluu BSD, jossa aaltosulut sijoitetaan omille riveilleen eikä samalle riville esimerkiksi if-ehdon kanssa. JavaScriptille yleistyneeseen tyyliin kuuluu taasen K&R-tyyli, jossa aaltosulku sijoittuu samalle riville if-ehdon kanssa. Muita säädettäviä asetuksia ovat esimerkiksi sisennyksien, välien, rivivaihtojen ja nimeämistyylien säännöt, joita on yhteensä satoja. Yhteensä GD:lle tuli monia kymmeniä muutoksia, joita voi tarkemmin tarkastella liitteestä 4.

4.2.2 Mallien luominen

ReSharper mahdollistaa koodin luomisen helposti avainsanoilla esim. foreach + tab luo foreach-rakenteen käyttäen valmista mallia. IntelliSense mahdollistaa valmiiseen

malliin arvojen valitsemisen helposti. Jokaisella kielellä kuten C#, JavaScript, Razor ja CSS on omat mallinsa, joita voi luoda ja muokata helposti ReSharperin Templates Explorer ikkunan kautta, joka voidaan nähdä yllä olevassa kuvassa 17.



Kuva 17. ReSharper-mallien luominen

Valmiita hyödyllisiä malleja kielille

- C#

  #region, for, forr (for reverse), foreach, do, while, if, else, switch, form, using, try

- JavaScript

  function, for, forin (samanlainen kuin C# foreach), if, else, switch, while, do, try

Itse tehtyjä malleja kielille

- JavaScript
  - ajax (jQuery $.ajax) -malli nimellä ”ajax”
  - document ready -malli nimellä “dr”

```
<script type="text/javascript">
    $(document).ready(function () {
        $END$
    });
</script>
```

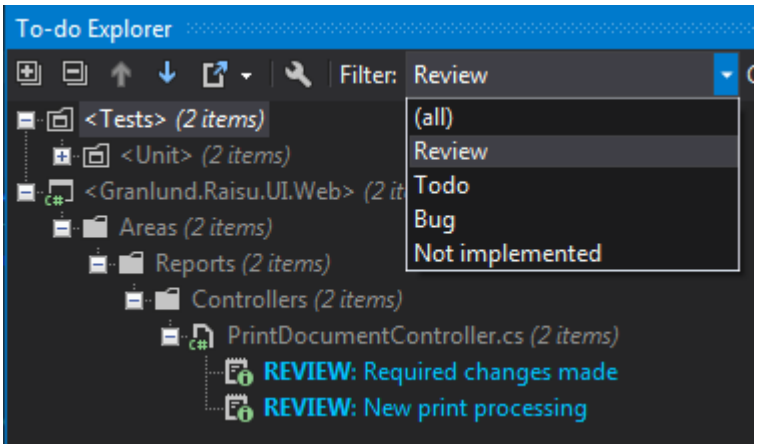
- JavaScript ja näkymät (View .NET Razor syntaksilla) yhteiset
    - wizard Init -malli nimellä “winit”
    - wizard open -malli nimellä “wopen”
    - dialog open -malli nimellä ”dopen”
- Näkymät
    - link -malli nimellä ”link”.

Päätimme toteuttaa yllälistatut mallit. JavaScriptin ja näkymien yhteiset mallit ovat samoihin tarkoituksiin, mutta luovat kielelle tarkoitetun oman toteutuksen kielestä riippuen. Mallit, jotka liittyvät wizardiin tai dialogiin, sekä link ovat sovelluksen omille komponenteille tehtyjä malleja, joilla niiden käyttö on helpompaa ja nopeampaa. Malleja luodessa on kuitenkin muistettava ylläpidon hankaluus komponenttien muuttuessa, joten vain oikeasti tarvittavia ja helposti ylläpidettäviä malleja kannattaa tehdä.

4.2.3 Katselmointi toiminnallisuuden toteuttamisen tutkiminen

ReSharperin To-do Explorer, joka voidaan nähdä kuvassa 18, mahdollistaa avainsanojen luomisen ja muokkaamisen, millä koodia voidaan merkata eri tarkoituksiin. Näitä ovat vakiona ”Todo” eli jokin jäljellä oleva tehtävä, ”Bug” eli virhe ja ”Not implemented” eli vielä toteuttamaton ominaisuus. Harkitsimme myös avainsanan ”Review” lisäämistä, joka mahdollistaisi haluttaessa helpon katselmointimerkintöjen tekemisen kaikille kehittäjille. Nämä kaikki voidaan listata helposti käyttäen To-do Exploreria. Lisäksi listaa voidaan rajoittaa avainsanojen perusteella, kuten kuvasta voidaan nähdä. Avainsanojen väriä voidaan vaihtaa To-do Explorerin listalla, mutta

koodissa kaikki näkyvät samalla vaalean sinisellä värillä kuin ”TODO”, joka hieman rajoittaa tunnistamista.



Kuva 18. To-do listaus

Avainsanoja käytetään koodissa seuraavasti

- // TODO: tekstiä,
- // BUG: tekstiä
- ja // REVIEW: tekstiä.

Päätimme kuitenkin jättää tämän ominaisuuden vielä ottamatta käyttöön, sillä se ei vastannut aivan haluttua toiminnallisuutta. Suunnitelmissa on tutkia muita paremmin käyttötarkoitukseen soveltuvia vaihtoehtoja tulevaisuudessa. Näihin luetaan mukaan Review Assistant, joka on suunniteltu mahdollisesti enemmän siihen, mitä toivottiin.

4.3 Muita laadun ylläpitoon huomioon otettavia toimenpiteitä ja työkaluja

Tässä osiossa käydään läpi lyhyesti muita tärkeitä laadun ylläpidon ja parantamisen toimenpiteitä sekä työkaluja. Osiossa analysoidaan myös, mitkä näistä olisivat GD:lle varteenotettavia toimenpiteitä, jotka vielä puuttuvat tai jo kuuluvat käytäntöihimme.

4.3.1 Tietoturvallisuuden analysointi

Tämän luvun tiedot perustuvat suurelta osalta Agile Finland ry:n järjestämän tietoturvaillan 19.2.2014 esityksiin ja siellä käytyihin keskusteluihin. Tapahtumaan osallistui monien eri yritysten tietoturvavastaavia ja muita aiheesta kiinnostuneita sovelluskehitysalan työntekijöitä. Esitelmiä tapahtumassa pitivät Intel ja F-Secure.

Tietoturva tulkitaan usein erillisenä osana sovelluksessa ja se jätetään myös usein joko kokonaan suunnittelematta ja toteuttamatta tai se toteutetaan vasta, kun joudutaan ongelmiin. Tämä voidaan kuitenkin tulkita myös suunniteltuna riskinä, vaikka ajattelutapana tämä ei ole kovinkaan suosittu ammattilaissovelluskehittäjien keskuudessa eikä eettisesti oikein, sillä normaalisti loppukäyttäjä kuitenkin olettaa tietojensa olevan turvassa. Tietoturvan suunnittelu ja toteuttaminen vie aikaa ja rahaa. Mikäli julkaisija tai palveluntarjoaja tulkitsee tietoturvan tarpeettomaksi sovelluksessaan, voidaan tehdä riskipäätös ja jättää se toteuttamatta. Tällä säästetään kehityskustannuksissa ja riskeerataan sovelluksen ja yrityksen imago. Jos tulevaisuudessa joudutaan pahoittelemaan tietomurtoja tai väärinkäyttöjä toteuttamattoman tietoturvan takia, voi yrityksen ja sovelluksen imago kärsiä.

Tietoturvallisuuden analysointiin löytyy myös valmiita työkaluja, jotka havaitsevat tunnettuja ja yleisiä haavoittuvuuksia sovelluksen toteutuksessa. Niillä voidaan parantaa ainakin hieman sovelluksen tietoturvaa. Markkinoilta löytyy muutamia sovellusvaihtoehtoja, jotka mahdollistavat jonkin tason automaattisen tietoturvan analysoinnin. On kuitenkin muistettava, että sovelluksen tietoturva pitäisi tulkita oleellisena osana sovellusta ja suunnitella sekä toteuttaa sovelluksen kehityksen aikana, eikä vasta kun sovellus on jo tehty. Jos tietoturvaa ei ole otettu huomioon sovelluksen kehitysvaiheessa, tulee sen parantaminen kalliiksi jälkikäteen. Pitää myös ottaa huomioon, että nämä analysointisovellukset eivät ole läheskään kaikenkattavia oikeaan tietoturva-analyysiin. Onkin suositeltavaa, että sovelluskehitysryhmästä löytyy vähintään yksi tietoturvaan erikoistunut kehittäjä tai projektin suunnittelu, ja toteutusvaiheessa käytetään ulkopuolista konsulttia.

GD:llä ei ole tietoturvaan erikoistunutta kehittäjää, mutta kuitenkin ammattitaitoinen kehitysryhmä ja sovellusarkkitehti. Tietoturvaan voitaisiin kuitenkin panostaa huomattavasti enemmän.

4.3.2 Koodikatselmointi

Katselmointi voidaan määritellä monella eri tavalla, eikä sillä ole aina samaa merkitystä. Se kuitenkin perustuu siihen, että esimerkiksi tietyn ominaisuuden tekemisen jälkeen lähdekoodi katselmoidaan eli käydään läpi laadunvarmistussyistä [14, s. 152-157.]. Asioita, joihin katselmoivan henkilön kannattaa kiinnittää huomiota, ovat arkkitehtuuri-, rakenne-, tyyli- ja logiikkavirheet, uudelleenkäytettävyys, ylläpidettävyys, monimutkaisuus sekä tehokkuus. Kyseessä on siis oleellinen laadunvarmistustoimenpide. Katselmointi voidaan suorittaa esimerkiksi toisen kehittäjän toimesta. Kehittäjä voi tehdä sen itse, tai se voidaan tehdä ulkopuolisen toimijan toimesta. Yhtenä vaihtoehtona on myös niin sanottu ”one-on-one”-menettely, jossa koodin tekijä läpikäy koodinsa toisen kehittäjän kanssa. Oman koodin läpikäyminen ja ”one-on-one” -käytäntö voidaan tulkita epävirallisiksi, sillä koodin alkuperäinen tekijä ei usein huomaa tekemiään virheitä yhtä helposti kuin ulkopuolinen henkilö. Niin sanottu virallinen katselmointi voidaan tehdä kehitysryhmän toisen jäsenen tai jäsenten toimesta, sillä uusi perspektiivi lisää virheiden havaitsemisen mahdollisuutta. Katselmoinnin voi myös ulkoistaa tällaista palvelua tarjoavalle yritykselle, mutta sovelluksen ja koodipohjan tuntemus ovat tällöin rajoitteina, jotka saattavat mahdollisesti hidastaa ja heikentää prosessin laatua. [2, s. 385-400.]

GD:ssa on ollut käytäntönä, että kaikki uudet ominaisuudet katselmoidaan arkkitehdin toimesta. Tämän jälkeen tehdään refaktorointi eli korjaukset, jos sellaisia tulkitaan tarpeellisiksi. Refaktoroinnilla tarkoitetaan koodin rakenteen muuttamista, esimerkiksi arkkitehtuurillisesti, ilman että toiminnallisuus muuttuu. Termiä kuitenkin käytetään monesti väärin myös tapauksissa, joissa itse sovelluksen toiminnallisuus muuttuu [15.]. Tämän tehtävän vain yhdelle henkilölle jättäminen kuitenkin työllistää yhtä henkilöä turhaan. Haluttaisiin siis jokin käytäntö, jossa kuormaa jaetaan muillekin sovelluskehittäjille, sekä jokin katselmointiprosessia helpottava työkalu.

Alkuperäinen suunnitelma oli tutkia NDepend- ja ReSharper-ominaisuuksia, jos ne mahdollistaisivat halutun toiminnallisuuden. Tämä ei kuitenkaan onnistunut NDependillä, ja ReSharperilla se ei onnistunut aivan halutulla tavalla. Tulevaisuudessa siis luultavasti tutkitaan toisen sovelluskehitysryhmän käyttämää Review Assistant -työkalua ja sen antamia mahdollisuuksia. Markkinoilla on varmasti myös muita samaan tarkoitukseen suunniteltuja työkaluja, jos Review Assistant ei sovellu käyttötarkoitukseemme tarpeeksi hyvin. Jotta katselmoinnin kuormaa saataisiin

kuitenkin jaettua, voisi GD:lle suositella otettavaksi käyttöön ”one-on-one”-käytäntöä, jolloin jokainen kehittäjistä pääsee osallistumaan katselmointiin, mutta ei joudu yksin läpikäymään toisen henkilön tekemää koodia, joka olisi erittäin hidasta ja työlästä. Tässä on kuitenkin otettava huomioon, että kaikkia virheitä ei välttämättä löydetä tekijän perspektiivin takia. Siksi olisi suositeltavaa tehdä myös katselmointeja pelkän toisen kehittäjän toimesta, mutta suuremmalla aikavälillä. Käytäntönä voisi siis olla, että kun uusi ominaisuus valmistuu ja todetaan testauksella toimivaksi, luodaan ”one-on-one”-katselmointitehtävä, johon valitaan joku kehittäjistä katselmointipariksi ominaisuuden tekijälle. Kehittäjät käyvät koodin läpi tekijän ohjaamana ja tekevät merkinnät tarvittavista parannuksista, jonka jälkeen kehittäjä tekee tarpeelliset korjaukset ja tehtävä suljetaan. Lisäksi olisi pidemmän aikavälin viralliset katselmoinnit, joihin tekijä itse ei osallistu, joiden ajankohta riippuu tehdyksi saatujen tehtävien sen hetkisestä määrästä.

5 Tulokset

Työssä toteutettiin NDepend-staattisen analyysin työkalun asennus, sekä GD:lle sopivat säännöt. NDepend-raportin luonti sidottiin TFS:n jatkuvan integraation kääntömäärittelyihin. Lopputuloksena on raportti jokaisen versiohallintajärjestelmään vietävän muutoksen jälkeen, johon on päästävissä käsiksi verkkoselaimella yrityksen sisäverkosta. Raportin avulla voidaan seurata sovelluksen nykyistä tilaa, tapahtuneita muutoksia ja mihin ollaan menossa. Tulevaisuudessa on suunnitelmissa toteuttaa vielä reaaliaikainen tiedotustaulu, joka näyttää mukautetun raportin sovelluskehitysryhmän huoneen seinällä. Taululle tullaan myös luultavasti lisäämään lokitiedoista kerättyjä virhetilastoja sekä TFS-käännöstietoja.

```
foreach (var objectHierarchyId in codedObjectHierarchyIds)
{
    ServiceFacade.Equipment.EquipmentHierarchyService.ActivateEquipments(codedObjectHierarchyIds, isActive: true);
}
```

Kuva 19. Esimerkki ReSharperilla löydetystä virheestä

Työssä toteutettiin myös jatkuvan analyysin työkalun eli ReSharperin käyttöönotto, GD:lle sopivien sääntöjen määrittely ja niiden automatisoitu jakaminen kehitysryhmän jäsenten välillä. Käyttöönotossa havaittiin lähemmäs 100 virhettä koodissa, jotka korjattiin saman tien. Osa virheistä oli lähes merkityksettömiä, mutta löytyi myös paljon

virheitä, jotka vaikuttivat suuresti palvelimen kuormitukseen ja suorituskykyyn. Hyvänä esimerkkinä tällaisesta oli niinkin yksinkertainen tilanne kuin kuvasta 19 voidaan nähdä, jossa foreach toistorakenteessa tehtiin tietokantapäivityskutsu käyttämällä toistorakenteen muuttujan sijaan muuttujana toistorakenteen listaa. Näin siis tehtiin listan pituuden verran kertoja sama päivitysoperaatio kaikilla listan sisällä olevilla arvoilla. Jos siis listassa oli 100 kohdetta, niin tehtiin 100 kertaa 100 kohteen päivitys. Vaikka koodi siis kääntyikin ja tuntui toimivan oikein oli virheenä käyttämätön muuttuja ja virheellinen logiikka. Toteutuksen kannalta tässä tapauksessa on myös tietenkin suositeltavampaa päivittää koko listan sisältö kerralla yhdellä kutsulla eikä toistorakenteen avulla monella yksittäisellä kutsulla.

Nyt ReSharper tarkkailee automaattisesti ja ilmoittaa, jos virheitä kuten käyttämättömiä muuttujia löytyy. Se ilmoittaa vapaaehtoisista korjaustoimenpiteistä ja parannusehdotuksista koodin joukossa. Lisäksi ReSharper mahdollistaa helpon koodin ulkonäön siistimisen sääntöjen mukaisesti yhdellä pikanäppäimen painalluksella. Näin koodi pysyy yhdenmukaisena ja siistinä eri kehittäjien välillä ja ulkonäköön ei tarvitse käyttää kehittäjän aikaa.

Työssä päästiin haluttuihin tavoitteisiin NDependin ja ReSharperin osalta ja lopputulokseen voidaan olla tyytyväisiä. Työkalut helpottavat huomattavasti virheiden havaitsemista ja laadun ylläpitämistä haluttujen sääntöjen mukaisesti. Harkittaessa kyseessä olevien sovelluksien käyttöönottoa sovelluskehitysprojekteissa kannattaa ottaa huomioon projektin jäsenten kiinnostuksen ja panostuksen taso laadunparantamiseen. Jos ryhmästä ei siis löydy ketään, joka hyödyntäisi NDependin luomia raportteja ja tietoja, on työkalu lähes tarpeeton. Suurin osa luodusta informaatiosta vaatii analysointia ja ymmärrystä varsinkin graafisien näkymien osalta. NDependin antamat tulokset myös näkyvät vasta jälkikädessä, sillä staattinen analyysi alkaa antaa tehokkaita tuloksia vasta muutaman mittauspisteen jälkeen. NDependin sopivuus kehitysryhmälle ja sen antamien tuloksien vakuuttavuus ei siis ole yhtä itsestään selvää kuin ReSharperin. ReSharperia voidaan taas suositella hyvänä ratkaisuna, ja melkeinpä jopa välttämättömänä työkaluna kaikille projekteille. Sen käyttö ei vaadi lähes ollenkaan perehtymistä, ja se antaa tuloksia jatkuvassa kehityksessä kokoajan. Kummallekin työkalulle on kuitenkin suositeltavaa tehdä tarkat projektikohtaiset säännöt, jotta työkaluista saadaan kaikki irti eikä tuoda turhia tietoja näkyviin häiritsemään kehittäjää.

Katselmoinnin osalta ei kuitenkaan päästy aivan toivottuun lopputulokseen käyttämällä NDepend- tai ReSharper-sovelluksia, jota myös epäiltiin työtä aloitettaessa. Haluttiin kuitenkin määritellä, jos toiminnallisuuden olisi saanut näitä työkaluja käyttäen toteutettua, sillä vähemmän sovelluksia on aina parempi lisenssien, käytön opiskelun ja yksinkertaisuuden takia. Kuitenkin päätettiin, että tulevaisuudessa tutkitaan toisen kehitysryhmän käyttämää Review Assistant -sovellusta, jonka pitäisi soveltua tarkoitukseen hyvin.

6 Yhteenveto

Työssä tutkittiin Granlund Designer -sovelluksen lähdekoodin laadun parantamista ja siihen liittyviä menetelmiä sekä työkaluja. Alussa käsiteltiin laatua käsitteenä, sen merkitystä sovelluskehityksessä ja määriteltiin, miten sitä voidaan mitata erilaisilla mittareilla. Pääosassa olivat NDepend- ja ReSharper-työkalujen antamat mahdollisuudet sekä niiden käyttöönoton suorittaminen ja projektille sopivien asetuksien luominen. Lisäksi tarkkailtiin muita laadunparantamiseen liittyviä menetelmiä, jotka voisivat olla hyviä ottaa käyttöön projektissa tai joita jo käytetään ja voidaan todeta päteviksi. Lopuksi käsiteltiin myös katselmointiin liittyviä käytäntöjä ja prosessin helpottamista sovelluksilla.

Työssä päästiin haluttuihin tavoitteisiin NDependin ja ReSharperin osalta ja lopputulokseen voidaan olla tyytyväisiä. Työkalut helpottavat huomattavasti virheiden havaitsemista ja laadun ylläpitämistä haluttujen sääntöjen mukaisesti. NDepend luo raportin aina, kun muutoksia viedään versiohallintaan ja kääntöpalvelin tekee uuden käännöksen. ReSharper taas on jatkuvasti käytössä jokaisen kehittäjän työympäristössä antamassa korjausehdotuksia ja parantamassa kehitystyötä. Katselmoinnin osalta ei kuitenkaan päästy aivan toivottuun lopputulokseen käyttämällä NDepend- tai ReSharper-sovelluksia. Lopuksi suunniteltiin, että tulevaisuudessa tutkitaan Review Assistant -sovellusta, jonka pitäisi soveltua tarkoitukseen hyvin.

Lähteet

1. Chemuturi, Murali. 2010. Mastering Software Quality Assurance : Best Practices, Tools and Technique for Software Developers. Ft. Lauderdale, FL, USA: J. Ross Publishing Inc.
2. Goodliffe, Pete. 2006. Code Craft : The Practice of Writing Excellent Code. San Francisco, CA, USA: No Starch Press, Incorporated.
3. Bangare, Pallavi S.; More, Pooja; Bangare, Sunil L.; Upadhye, Ashish; Zambad, Pooja. 2010. Re-Evaluation of Visual Studio 2010 Add-ins For Coding Guidance. Verkkodokumentti. <http://www.ijitee.org/attachments/File/v2i4/D0492032413.pdf>. Luettu 27.4.2014.
4. NDepend. Metrics. Verkkodokumentti. <http://www.ndepend.com/Metrics.aspx>. Luettu 4.8.2014.
5. NDepend. Dependency Structure Matrix. Verkkodokumentti. <http://www.ndepend.com/doc_matrix.aspx>. Luettu 4.4.2014.
6. CodeBetter. Code metrics on Coupling, Dead Code, Design flaws and Re-engineering. Verkkodokumentti. <http://codebetter.com/patricksmacchia/2008/02/15/code-metrics-on-coupling-dead-code-design-flaws-and-re-engineering/>. Luettu 1.4.2014.
7. Stackoverflow. What is Abstractness vs. Instability Graph?. Verkkodokumentti. <http://stackoverflow.com/questions/1031135/what-is-abstractness-vs-instability-graph>. Luettu 27.4.2014.
8. Martin, Robert. OO Design Quality Metrics An Analysis of Dependencies. Verkkodokumentti. <http://www.objectmentor.com/resources/articles/oodmetrc.pdf>. Luettu 9.4.2014.
9. NDepend. Visualizing Code Metrics with Treemap. Verkkodokumentti. http://www.ndepend.com/Doc_Treemap.aspx. Luettu 8.4.2014.
10. NDepend. CQLinq Features. Verkkodokumentti. <http://www.ndepend.com/Doc_CQLinq_Features.aspx>. Luettu 8.4.2014.
11. Debbarma, Mrinal Kanti; Debbarma, Swapan; Debbarma, Nikhil; Chakma, Kunal; Jamatia, Anupam. 2013. A Review and Analysis of Software Complexity Metrics in Structural Testing. <http://www.ijcce.org/papers/154-K271.pdf>. Luettu. 27.4.2014.
12. Hanselman, Scott; Cauldwell, Patrick; Cell, Stuart. NDepend Code Metrics. Verkkodokumentti. <http://www.hanselman.com/blog/content/binary/NDepend%20metrics%20placemats%201.1.pdf>. Luettu 10.4.2014.
13. JetBrains. ReSharper Customization Guide. Verkkodokumentti. <http://confluence.jetbrains.com/display/NETCOM/ReSharper+Customization+Guide>. Luettu 13.2.2014.
14. Tomayko, Jim; Hazzan, Orit. 2004. Human Aspects of Software Engineering. Herndon, VA, USA: Charles River Media / Cengage Learning.
15. Ritchie, Peter. 2010. Refactoring with Microsoft Visual Studio 2010. Olton, Birmingham, GBR: Packt Publishing Ltd.

TFS kääntöasetuksissa kattavuustiedoston luomisen määrittely

Kattavuustiedoston luomisen määrittelevä asetus löytyy seuraavasti: Visual Studio, Team Explorer Home, Builds, valitaan oikealla hiirenpainikkeella normaali jatkuvan integraation kääntömääritys, joka tässä tapauksessa on Raisu4ContinuousIntegration ja valitaan ”Edit Build Definition…”. Process kategorian alta on valittava Automated Tests, Test Source, Run Settings, Type of run settings: CodeCoverageEnabled.

NDepend GD-ryhmän säännöt

GD:lle erikseen luodut säännöt GD-prefiksillä

- GD – Obsolete types

```
// <Name>GD - Obsolete types</Name>
warnif count > 0
from t in JustMyCode.Types where
t.IsObsolete
select t
```

- GD – Check that all namespaces begin with Granlund.Raisu

```
// <Name>GD - Check that all namespaces begin with Granlund.Raisu</Name>
warnif count > 0 from n in Application.Namespaces where
  !n.NameLike (@"^Granlund.Raisu")
select new { n, n.NbLinesOfCode }
```

Seuraavassa taulukossa on loput GD ryhmään kuuluvat säännöt

Sääntö (kriittinen)	**Lisätiedot** (tarvittaessa)	**Alkuperäiset ehdot**	**Lopulliset ehdot** (muuttuneet = kursivoitu)
Types too big	Oli alun perin kriittinen.	NbLinesOfCode > 500	NbLinesOfCode > 500
Methods too complex		ILCyclomaticComplexity > 40 && ILNestingDepth > 5	*ILCyclomaticComplexity > 30 &&* ILNestingDepth > 5
Controller actions with too many parameters - critical	Rajoitettu kontrollerimetodeihin	NbParameters > 3	NbParameters > 3
Controller actions with too many parameters	Rajoitettu kontrollerimetodeihin	NbParameters > 5	*NbParameters > 8*
Methods with too		NbParameters > 5	*NbParameters > 8*

many parameters			
Quick summary of methods to refactor	UnitTestit rajattu ulkopuolelle	NbLinesOfCode > 30 || CyclomaticComplexity > 20 || ILCyclomaticComplexity > 50 || ILNestingDepth > 5 || NbParameters > 5 || NbVariables > 8 || NbOverloads > 6	*NbLinesOfCode > 100* || CyclomaticComplexity > 20 || *ILCyclomaticComplexity > 40* || ILNestingDepth > 5 || *NbParameters > 8* || *NbVariables > 20* || NbOverloads > 6
Don't use obsolete types, methods or fields			
Potentially dead Types			
Potentially dead Methods (excluding UnitTests)	UnitTestit rajattu ulkopuolelle		
Potentially dead Fields			
Interface name should begin with a 'I'			
Methods name should begin with an Upper character			
Methods too big		NbLinesOfCode > 40	*NbLinesOfCode > 100*
Methods too complex		CyclomaticComplexity > 20 || ILCyclomaticComplexity > 40 || ILNestingDepth > 5	Ei muutoksia
Types with too many methods		Methods.Count() > 20	*Methods.Count() > 50*
Types with too many fields		Fields.Count() > 20 && !IsEnumeration	*Fields.Count() > 30 &&* !IsEnumeration

NDepend-säännöt

Alla on listattuna vakioryhmien säännöt, jotka otettiin pois käytöstä. Pitää muistaa ottaa huomioon, että osa säännöistä on siirretty GD ryhmän alle.

- Code Quality (4/13). Seuraavat **jätettiin käyttöön**:
    - Methods potentially poorly commented
    - Methods with too many local variables
    - Methods with too many overloads
    - Types with poor cohesion
- Code Quality Regression (12/12)
- Object Oriented Design (10/13)
    - Class with no descendant should be sealed if possible
    - Non-static classes should be instantiated or turned to static
    - Methods should be declared static if possible
- Design (12/14)
    - Avoid namespaces with few types
    - Don’t use obsolete types, methods or fields
- Architecture and Layering (8/9)
    - Avoid partitioning the code base through many small library Assemblies

- API Breaking Changes (10/10)
- Code Diff Summary (25/25)
- Test and Code Coverage (13/13)
- Dead Code (1/4)
    - Wrong usage of IsNotDeadCodeAttribute
- Visibility (8/11)
    - Methods that could have a lower visibility
    - Types that could have a lower visibility
    - Fields that could have a lower visibility
- Purity – Immutability – Side-Effects (10/11)
    - Property Getters should be immutable
- Naming Conventions (10/17)
    - Instance fields should be prefixed with a ‘m_’
    - Static fields should be prefixed with a ‘s_’
    - Interface name should begin with a ‘I’
    - Methods name should begin with an Upper character
    - Avoid types with name too long
    - Avoid methods with name too long

- Avoid fields with name too long

- Source Files Organization (4/6)
  - Avoid defining multiple types in a source file
  - Namespace name should correspond to file location

ReSharper-asetukset

Environment

- General
    - Show tips on startup -> otetaan pois käytöstä

Code Inspection

- Settings
    - Edit Items to Skip
        - kendo.common.css
        - kendo.themebuilder.css
        - jquery.wizard.js
        - kendo.aspnetmvc.js
        - ProjectDocumentData(folder)
- Generated Code
    - Generated file masks
        - *.map
- Inspection Severity
    - C#

        - Class is never instantiated -> Non-private accessibility -> Do not show
        - Unused local variable -> Error
        - Virtual no inheritance -> Do not show
    - JS
        - Unused local variable or function -> Error

Code Editing

- Code Cleanup
    - Luotiin GD-profiili ao. asetuksilla ja valittiin se ”silent clean-up” profiiliksi
        - HTML, C#, JavaScript, CSS ja XML
            - Reformat code -> käyttöön
        - C#
            - Use auto-property, if possible -> käyttöön
            - Make field read-only, if possible -> käyttöön
            - Optimize ‘using’ directives -> käyttöön
        - JavaScript
            - Terminate statements -> käyttöön
- C#

- Naming Style
  - Static fields (private) -> lowerCamelCase[_lowerCamelCase]
  - Instance fields (private) -> lowerCamelCase[_lowerCamelCase]
  - Advanced settings… -> Abbreviations -> lisättiin UI
- Formatting Style
  - Braces Layout
    - Force Braces -> kaikkiin Add braces
  - Blank Lines
    - Preserve Existing Formatting
      - Keep max blank lines in declarations -> 1
      - Keep max blank lines in code -> 1
  - Line Breaks and Wrapping
    - Line Wrapping
      - Right margin (columns) -> 180
      - Wrap LINQ expressions -> Chop always
    - Other
      - Place singleline accessor attribute on same line -> pois käytöstä

- Spaces
  - Around Operators
    - Multiplicative operators (*,/,%) -> käyttöön
  - Other
    - Within single-line initialize braces -> käyttöön
- Other
  - Align Multiline Constructs
    - LINQ query -> käyttöön
- Namespace Imports -> Not Removed
  - System
  - System.Collections.Generic
  - System.Linq

- HTML
  - Other
    - JavaScript templates MIME types -> lisättiin “text/x-kendo-template”

- JavaScript
  - JavaScript Naming Style

        - Abbreviations -> lisättiin UI
    - Formatting Style
        - Line Breaks
            - Blank lines
                - Max blank lines in code -> 1
                - Max blank lines between declarations -> 1
            - Braces Layout
                - Force braces
                    - Braces in statements -> Add braces
                - Case statement
                    - Indent “case” from “switch” -> käyttöön
- TypeScript
    - Spaces
        - Lambdas
            - Before ‘=>’ -> käyttöön
- CSS
    - Inspections
        - Check browser compatibility

- IE9+
- Opera -> pois käytöstä
- Safari -> pois käytöstä

- Formatting Style
  - Code Layout
    - Properties -> On separate lines

- XML
  - Formatting Style
    - Line Wrapping (general)
      - Right margin (columns) -> 180
    - Around tags
      - Maximum blank lines between tags -> 1

- XML Doc Comments
  - Formatting Style
    - Line Wrapping (general)
      - Right margin (columns) -> 180